Práctica II PROLOG

Víctor Mora Afonso. alu3966.
Inteligencia Artificial. 9 de enero de 2012.

Ejercicios sobre Unificación

- **Indique si los siguientes pares de términos unifican entre sí. En caso de que unifiquen, indique a qué valores se ligan las variables.**

p(a) y p(A)

```
5 ?- p(a) = p(A) .
9 ?- A = a.
```

p(j(j(j(j(j))))) y p(j(j(j(j))))

```
10 ?- p(j(j(j(j(j))))) = p(j(j(j(j)))).
false.
```

p(j(j(j(j(j))))) y p(j(j(j(X))))

```
11 ?- p(j(j(j(j(j))))) = p(j(j(j(X)))).
X = j(j).
```

q(_,A,_) y q(32,37,12)

```
12 ?- q(_,A,_) = q(32,37,12).
A = 37.
```

z(A,p(X),z(A,X),k(Y)) y z(q(X),p(Y),z(q(z(H))),k(z(3)))

```
13 ?- z(A,p(X),z(A,X),k(Y)) = z(q(X),p(Y),z(q(z(H))),k(z(3))).
false.
```

z(A,p(X),z(A,X),k(Y)) y z(q(X),p(Y),z(q(z(H)),z(H)),k(z(3)))

```
14 ?- z(A,p(X),z(A,X),k(Y)) = z(q(X),p(Y),z(q(z(H)),z(H)),k(z(3))).
A = q(z(3)),
X = z(3),
Y = z(3),
H = 3.
```

- **A continuación, ejecute las siguientes secuencias de objetivos en el top-level shell y observe las ligaduras de las variables:**

f(X) = f(Y).

```
1 ?- f(X) = f(Y).
```

X = Y.

X = 12, f(X) = f(Y).

4 ?- X=12,f(X) = f(Y).
X = 12,
Y = 12.

f(X) = f(Y), X = 12.

2 ?- X=12, f(X) = f(Y).
X = 12,
Y = 12.

f(X) = f(Y), Y = 12.

5 ?- f(X) = f(Y), Y=12.
X = 12,
Y = 12.

X = Y, Y = Z, X = H, Z = J, X = 1.

4 ?- X=Y, Y=Z, X=H, Z=J, X=1.
X = 1,
Y = 1,
Z = 1,
H = 1,
J = 1.

X = 1.

5 ?- X =1.
X = 1.

1 = X.

6 ?- 1 = X.
X = 1.

Ejercicios sobre predicados

- **A continuación indicamos las soluciones de tres predicados (el orden es significativo):**

p(5,2) tiene éxito.
p(7,1) tiene éxito.
q(1,3) tiene éxito.
z(3,1) tiene éxito.
z(3,7) tiene éxito.
No hay más soluciones que las anteriores.

Indique los pasos de ejecución para la secuencia p(A,B),q(B,C),z(C,A).

```
1 ?- p(A,B),q(B,C),z(C,A).
A = 7,
B = 1,
C = 3.
```

Defina el predicado sumar_dos/2 que toma un número en el primer argumento y retorna en el segundo argumento el primero sumado a dos. ¿Cuáles son los modos de uso permitidos para dicho predicado?.

```
sumar_dos(A,B) :- B is +(A,2).
```

Ejercicios de Listas

1. Define un predicado PROLOG suma(X, L1, L2) que unifique la lista L2 con el resultado de sumar X a los enteros en las posiciones pares de L1.

```
suma(X,[],[]).
suma(X,[Y|R],[Y|B]) :-
        suma_par(X,R,B).

suma_par(X,[],[]).
suma_par(X,[Y|R],[A|B]) :-
        A is +(X,Y),
        suma(X,R,B).

?- ['suma.pl'].
% suma.pl compiled 0.00 sec, 1,380 bytes
true.

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

Lo que se ha hecho para obtener el resultado es crear dos predicados suma que se van llamando el uno al otro de forma alternada a medida que se va recorriendo la lista inicial. En suma_par se añadirá a la lista final la cabeza de la lista que llega como entrada sumándole X, mientras que en el predicado suma no se sumará. Haciendo llamadas alternativas a cada uno de estos predicados se consigue que se sume X sólo a los elementos pares.

2. Define un predicado PROLOG elimina(N, L1, L2) que unifique L2 con la lista que resulta de eliminar todos los números menores que N de la lista L1.

```
elimina(_N,[],[]).
elimina(N,[Y|R],[Y|W]) :-
   Y >= N,
   elimina(N,R,W).

elimina(N,[Y|R],W) :-
   Y < N,
   elimina(N,R,W).

?- ['elimina.pl'].
% elimina.pl compiled 0.00 sec, 1,160 bytes
true.

?- elimina(4,[1,6,2,7,3,3,9,1],L).
L = [6, 7, 9] ;
false.
```

Es necesario definir tres preficados elimina, en función de cómo sea la lista de entrada:
- Si la lista está vacía, se devolverá una lista vacía. El valor de N es indiferente.
- Si la cabeza de la lista que se pasa como parámetro es mayor o igual que N, esta formará parte de la lista final, se pondrá a la cabeza y se procederá a calcular el resto de la lista final a partir del resto de la orginal (eliminando aquellos elementos menores que N).
- Si la cabeza de la lista es menor que N, esta no estará en la lista final, por lo tanto no se incluye y se procede a calcular el resto de la lista final a partir del cuerpo de la lista inicial (eliminando aquellos elementos menores que N).

3. Define el predicado PROLOG potencia(X, N, R) que unifique R con la Nésima potencia de X.

```
potencia(_,0,1).
potencia(X,N,Z) :- N > 0, M is N - 1, potencia(X,M,C), Z is C*X.

?- ['potencia.pl'].
% potencia.pl compiled 0.00 sec, 1,060 bytes
true.

?- potencia(2,8,Z).
Z = 256 ;
false.
```

Primero se empieza definiendo el caso trivial, es decir, cualquier número elevado a 0 es 1. A partir de aquí se puede calcular cualquier potencia de forma recursiva decrementando el valor de N de uno en uno hasta llegar al caso base para ahora construir el resultado, que se obtiene multiplicando X por X^(n-1).

4. Define un el predicado PROLOG simplifica(L1, L2) que unifique L2 a la lista L1 con una única aparición de los elementos repetidos consecutivos de L1.

?-simplifica([a, b, b, b, c], L)
L=[a, b, c]

```
simplifica([],[]).
simplifica([X|R],NR) :-
  member(X,R),
  simplifica(R,NR).

simplifica([X|R],[X|NR]) :-
  not(member(X,R)),
  simplifica(R,NR).

?- ['simplifica.pl'].
% simplifica.pl compiled 0.00 sec, 1,252 bytes
true.

?- simplifica([1,2,3,2,1,6],L).
L = [3, 2, 1, 6] ;
false.
```

Si la lista a simplificar está vacía, no hay nada que simplificar, se devuelve la lista vacía. Ahora, cuando la lista tiene elementos, lo que se hace es ver si el primer elemento está contenido en el resto de la lista, si es así, este no formará parte de la lista final y se pasará a procesar el resto de elementos de la lista inicial. Por el contrario, si la cabecera de la lista no está incluido en el resto de la lista, entonces sí que formará parte de la lista final.
Para poder hacer este ejercicio se usa el predicado member() ya definido en prolog que determina si un elemento está contenido en una lista o no.

5. En matemáticas, la sucesión de Fibonacci es la siguiente sucesión infinita de números naturales: 0, 1, 1, 2, 3, 5, 8, ... La sucesión inicia con 0 y 1, y a partir de ahí cada elemento es la suma de los dos anteriores. Definir el predicado PROLOG fibonacci(X, N) que unifique N al número N-1 ésimo de la sucesión de fibonacci.

```
fibonacci(0,0).
fibonacci(1,1).
fibonacci(N,X) :-
  N > 0,
  N1 is N - 1,
  N2 is N - 2,
```

```
  fibonacci(N1,A),
  fibonacci(N2,B),
  X is A + B.

?- ['fibonacci.pl'].
% fibonacci.pl compiled 0.00 sec, 1,172 bytes
true.

?- fibonacci(10,X).
X = 55 ;
false.
```

Primero se crean los predicados de los casos triviales, es decir, el primer elemento de la serie de Fibonacci es 0 y el segundo es 1. Ahora, siempre que N sea mayor que 0, para calcular un elemento de la serie que no sea trivial, lo que se hace es calcular el elemento anterior (N-1) y el anterior del anterior (N-2); el elemento N-ésimo será la suma de estos dos.
La forma en la que se ha hecho el algoritmo es un tanto ineficiente, pues se repiten muchos cálculos. Sería interesante desarrollarlo de una manera que se almacenen estos cálculos y se puedan reutilizar en un futuro (memoization).

6. Define un predicado PROLOG modulo(M, N, X) que unifique X con M mod N sabiendo que este valor es M si M es menor que N. En caso contrario, será el mismo valor que el módulo de M - N.

```
modulo(0,X,X).
modulo(_X,1,0).
modulo(X,X,0).

modulo(M,N,X) :-
  N > 0,
  M > N,
  A is M-N,
  modulo(A,N,X).

modulo(M,N,X) :-
  M < N,
  X is M.

?- ['modulo.pl'].
% modulo.pl compiled 0.00 sec, 1,360 bytes
true.

?- modulo(10,3,X).
X = 1 ;
```

false.

Primero se definen los casos triviales: el módulo de dividir cero por cualquier número (menos el cero) es ese mismo número; el módulo de dividir cualquier número por uno es cero; el módulo resultante de dividir cualquier número por sí mismo es cero.
A la hora de definir el predicado, se comprueba primero que N es mayor que 0 para evitar problemas de división por cero. Si M es menor que N, el resultado será M. En caso contrario, habrá que calcular el módulo. Para ello no hay que dividir ya que la división entera no son más que restas, lo que se hace es restar N a M y calcular el modulo de este nuevo número recursivamente hasta que se alcanza la condición de parada (M < N).

7. Define el predicado PROLOG antecedente(LAnt, L) que sea válido si la lista LAnt unifica con una sublista de L con los primeros elementos de ella:

?- antecedente([1, 2,3], [1, 2, 3, 4, 5, 6])
true.

```
antecedente([],_L).
antecedente([X|R],[X|F]) :-
  antecedente(R,F).

?- ['antecedente.pl'].
% antecedente.pl compiled 0.00 sec, 960 bytes
true.

?- antecedente([1,4,5],[1,4,5,6,7]).
true.
```

Si LAnt es una lista vacía, independientemente del valor de L ésta siempre será un subconjunto de L. Si LAnt no es una lista vacía, para determinar que es una sublista válida su cabecera ha de coincidir con la de L y la cabecera del resto de LAnt y de L también han de coincidir y asi sucesivamente hasta que LAnt se vacíe. Cualquier otro caso dará falso.

Ejercicio de Acumuladores

Una fábrica de bicicletas necesita mantener el inventario de los componentes que fabrica. Estos están clasificados en: o piezas básicas: llanta, radio, manillar, engranaje, tornillo, marco trasero, tuerca, horquilla. o piezas ensambladas:

bici: rueda, rueda, marco.
rueda: radio, llanta, centro.
marco: marco trasero, marco delantero.
centro: engranaje, eje.
eje: tornillo, tuercas.

marco delantero: horquilla, manillar.

El predicado piezas(X, L), unifica L con la lista de todas las piezas básicas que hacen falta para fabricar X.

```
piezas(X, Ac, [X|Ac]) :- basica(X).
piezas(X, Ac, L) :- ensamblada(X, ListPiezas),
piezaslistaAc(ListPiezas, Ac, L).
piezaslistaAc([], Ac, Ac).
piezaslistaAc([X|T], Ac, L) :- piezas(X, Ac, PiezasCabeza),
piezaslistaAc(T, PiezasCabeza, L).
```

Ejemplo extraído de Programación en PROLOG, Clocksin, W. F. y Mellish, C. S.Ed. Gustavo Gili, S.A., 1987. Lo correcto es definir un predicado que no tiene entre sus argumentos ningún acumulador y cuyos objetivos se encargan de inicializarlos al valor adecuado. De esta manera el usuario no tiene que preocuparse de realizar esta tarea. En los ejemplos siguientes:

```
suma(X, N) :- sumaAc(X, 0, N).
prodEscalar(L1, L2, Resultado) :- prodEscalar(L1, L2, 0,
Resultado).
longitud(L, N) :- longitudAc(L, 0, N).
piezas(X, L) :- piezas(X, [], L).
```

Cuando el acumulador es una lista, la única forma de actualizarlo es incorporando elementos por la cabeza de la lista, por lo que el orden de elementos en la lista es inverso al orden en el que se construye la solución. Trabajando con listas se puede usar como alternativa a los acumuladores el predicado append(L1, L2, L) que instancia L a la lista resultante de concatenar
las listas L1 y L2. En el caso del ejemplo de la fábrica de bicicletas:

```
piezasNew(X, [X]) :- basica(X).
piezasNew(X, L) :- ensamblada(X, ListPiezas),
piezaslista(ListPiezas, L). piezaslista([], []).
piezaslista([X|T], L) :- piezasNew(X, PiezasCabeza),
piezaslista(T, PiezasCola),
append(PiezasCabeza, PiezasCola, L).
```

En este caso cada lista se recorre dos veces, una para listar los componentes y otra para construir la unión de las 2 listas. Esta solución es menos eficiente.

Se procede a realizar una ejecución del código propuesto en el enunciado. Primero hay que definir cuáles son las piezas básicas y también de qué pieza están compuestas las piezas compuestas (a través del predicado ensamblada).

```
basica(llanta).
basica(radio).
basica(manillar).
basica(engranaje).
basica(tornillo).
basica(trasero).
basica(tuercas).
basica(horquilla).

ensamblada(bici,[rueda,rueda,marco]).
ensamblada(rueda,[radio,llanta,centro]).
ensamblada(marco,[trasero,delantero]).
ensamblada(centro,[engranaje,eje]).
ensamblada(eje,[tornillo,tuercas]).
ensamblada(delantero,[horquilla,manillar]).

piezas(X, Ac, [X|Ac]) :- basica(X).
piezas(X, Ac, L) :- ensamblada(X, ListPiezas),
                    piezaslistaAc(ListPiezas, Ac, L).

piezaslistaAc([], Ac, Ac).
piezaslistaAc([X|T], Ac, L) :- piezas(X, Ac, PiezasCabeza),
                               piezaslistaAc(T, PiezasCabeza, L).


piezasNew(X, [X]) :- basica(X).
piezasNew(X, L) :- ensamblada(X, ListPiezas),
                   piezaslista(ListPiezas, L).

piezaslista([], []).
piezaslista([X|T], L) :- piezasNew(X, PiezasCabeza),
                         piezaslista(T,PiezasCola),
                         append(PiezasCabeza, PiezasCola, L).



?- ['acumuladores.pl'].
% acumuladores.pl compiled 0.00 sec, 3,916 bytes
true.

?- piezas(bici,[],L).
L = [manillar, horquilla, trasero, tuercas, tornillo, engranaje, llanta, radio, tuercas|...] ;
false.
```

```
?- piezasNew(bici,L).
L = [radio, llanta, engranaje, tornillo, tuercas, radio, llanta, engranaje, tornillo|...] ;
false.
```

Ejercicio del 8-puzzle del tutorial

http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

Estudiar este ejemplo y realizar los ejercicios 5.2.1. y 5.2.3.

5.2.1. Añadir estadísticas a 'solve', es decir, devolver también el número de nodos expandidos y el número de nodos restantes en 'Open' (al final de una búsqueda satisfactoria).

Dado que el código del 8puzzle es demasiado largo como para copiarlo en este documento (el archivo original está adjunto), sólo se copiarán las partes del documento que hayan sido modificadas para poder realizar los ejercicios.

Para añadir el conjunto de estadísticas, lo primero que hay que hacer es entender la implementación realizada del algoritmo de búsqueda A y el conjunto de funciones desarrolladas para obtener una representación del 8puzzle que este algoritmo pueda entender y resolver.*

```
search([State#_#_#Soln|Rest], Soln,Expandidos,EnAbierta) :- goal(State),
      size(Rest,EnAbierta), Expandidos is 0.
search([B|R],S,Expandidos,EnAbierta) :- expand(B,Children),
                insert_all(Children,R,Open),
                search(Open,S,Expandidos1,EnAbierta),
                      size(Children,C),
                      Expandidos is Expandidos1 + C.
```

Cuando se alcanza una solución, el tamaño de la lista Open será el número de nodos que quedaban en abierta en el momento que se llego a la solución. Para calcularlo se crea el predicado 'size', que determina el número de elementos de una lista.

```
size([],0).
size([_|T],N) :- size(T,N1), N is N1+1.
```

Para calcular el número de nodos expandidos la cosa no es tan trivial. Para realizar el cálculo se parte de la solución y se vuelve hacia atrás ya que este algoritmo A funciona de forma recursiva. De esta manera, cuando se alcanza la solución se supondrá que el número de estados expandidos es cero (se inicializa la variable). Ahora el número de nodos que se*

expanden en un nivel será la suma de los que se expandieron en el nivel anterior más el número de hijos que tiene el nodo actual. Estos hijos se crean en el predicado 'expand' y serán almacenados en la lista Children, por lo que todo se reduce a calcular el tamaño de esta lista usando de nuevo el predicado 'size'.

```
solve(State,Soln,Expandidos,EnAbierta) :- f_function(State,0,F),
                search([State#0#F#[]],S,Expandidos,EnAbierta), reverse(S,Soln),
                    size(Soln,Longitud),
                    nl, write('Longitud de la solucion: '), write(Longitud), nl.
```

Finalmente se ha añadido una última estadística, la longitud de la solución. Para ello simplemente hay que calcular la longitud de la lista Soln que contendrá el conjunto de movimientos que llevan desde el estado inicial hasta el objetivo.

5.2.3. Modificar el programa para que use f(n) = g(n) + p_fcn(n). Estas búsquedas darán como resultado soluciones óptimas (si existen). Usando las estadísticas del ejercicio 5.2.1 hacer comparaciones práctica entre las soluciones usando h(n) = p_fcn(n) +3*s_fcn(n) y h(n) = p_fcn(n).

```
    %%% the heuristic function
h_function(Puzz,H) :- p_fcn(Puzz,H),
```

Modificar la función heurística es sencillo, simplemente hay que eliminar el cálculo de s_fcn y la suma que se hacía. Ahora se calcula p_fcn y se unifica H con el valor devuelto por este predicado.
Anteriormente estaba de la siguiente manera:

```
    %%% the heuristic function
h_function(Puzz,H) :- p_fcn(Puzz,P),
                      s_fcn(Puzz,S),
                      H is P + 3*S.
```

Ahora simplemente hay que comparar el rendimiento de cada una de las funciones heurísticas utilizando las estadísticas creadas en el ejercicio anterior. El estado objetivo es goal(1/2/3/4/5/6/7/8/0).

*Primero se utiliza como heurística h(n) = p_fcn(n)+3*s_fcn(n).*

Problema	*Longitud solución*	*Longitud Open*	*Estados expandidos*
413285706	*7*	*122974*	*808*
123406758	*2*	*87*	*2240*

103426758	*3*	*89*	*2243*
123460758	*3*	*119*	*4220*

Ahora se utiliza como heurística h(n) = p_fcn(n)

Problema	*Longitud solución*	*Longitud Open*	*Estados expandidos*
413285706	*7*	*92*	*345*
123406758	*2*	*25*	*62*
103426758	*3*	*27*	*69*
123460758	*3*	*36*	*114*

La primera heurística, que es la suma de la distancia total de Manhattan más la secuencia de Nilsson, no es admisible y por tanto, no tiene porqué conducir a la solución óptima y, como se observa en las tablas tarda más (y ocupa más espacio) en llegar al objetivo.
La segunda heurístca (distancia total de Manhattan) sí que es una heurística admisible y, por tanto, llevará a una solución óptima y como se ve en las tablas ofrece mejores resultados que la otra heurística.