Programación Lógica

Evaluación 2011
Soluciones

Ejercicio 2 [20 puntos]

a) Indique si las siguientes afirmaciones son verdaderas o falsas. Justifique brevemente.

i. Si un conjunto de expresiones tiene un mgu, este es único.

Falso.

{f(x),f(y)}

{x/y}, {y/x} y {x/z,y/z} son mgu.

ii. Dados un programa lógico y un objetivo, podemos construir un único árbol SLD.

Falso.

Depende de la regla de computación utilizada.
(ver 2.c)

iii. Dado un programa lógico y un objetivo, se cumple que toda respuesta correcta es una respuesta computada.

Falso.

Sean el programa

p(X).

y el objetivo

<- p(X)

{X/3} es una respuesta correcta y no es una respuesta computada.

b) Para los siguientes pares de términos dar un mgu, si existe.

i. { p(f(a,x), g(y), z), p(z, w, f(v, y)), p(u,g(f(a,b)),t) }

{x / f(a,b), z / f(a,f(a,b)), v / a, y / f(a,b), w / g(f(a,b)), u / f(a,f(a,b)),
t / f(a,f(a,b)) }

ii. { a(f(y), w, g(z)), a(x, y, x) }

No unifica.

iii. { q([X|Xs]), q([a, b, c|Ys] }

X = a
Xs = [b,c|Ys]

c) Sea el programa P:

```
prog(X,Y) <- pred1(X), pred2(Y).
prog(X,X).
pred1(a).
pred1(b).
pred2(X).
```

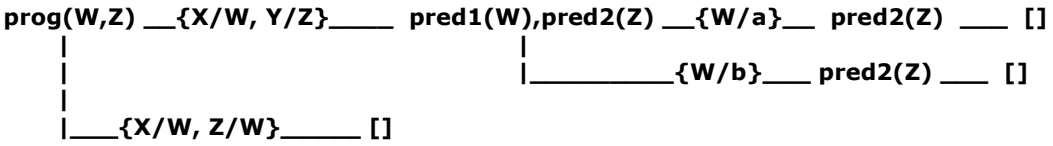
i. Dé una respuesta correcta que no sea una respuesta computada, si existe alguna, para P ∪ { <- prog(X,b) }.

Las respuestas computadas son:

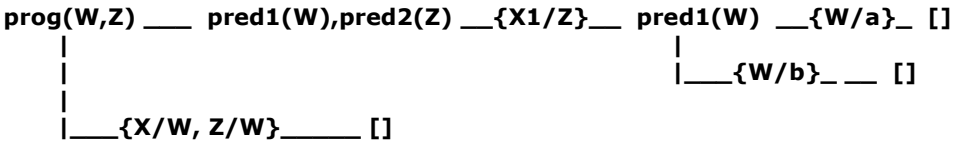
{X/a}, {X/b}, {X/b}

No existe una respuesta correcta que no sea computada.

ii. Construya un árbol SLD para P ∪ { <- prog(W,Z) }, con regla de computación primer átomo del objetivo.



iii. Construya un árbol SLD para P ∪ { <- prog(W,Z) }, con regla de computación último átomo del objetivo.



Ejercicio 3 [20 puntos]

Considere el siguiente programa:

```
p(X)    :- X > 2, !, a(X, X1), p(X1).
p(X)    :- X > 0, b(X).
p(0).
a(X, Y) :- X > 2, Y is X - 1.
a(X, Y) :- Y is X - 4.
b(3).
b(1).
```

a) Diga cuáles de las siguientes consultas dan true: ?- p(0), ?- p(1), ?- p(2), ?- p(3), ?- p(4).

b) Considere las siguientes variantes del programa:

i.
```
p(X) :- X > 2, !, a(X, X1), p(X1).
p(X) :- X > 0, b(X).
p(0).
a(X, Y) :- X > 2, !, Y is X - 1.
a(X, Y) :- Y is X - 4.
b(3).
b(1).
```

ii.
```
p(X) :- X > 2, a(X, X1), p(X1).
p(X) :- X > 0, !, b(X).
p(0).
a(X, Y) :- X > 2, Y is X - 1.
a(X, Y) :- Y is X - 4.
b(3).
b(1).
```

¿Cuáles de las consultas dadas en a) dan true para cada variante?

c) Defina cut rojo y cut verde. Indique, para cada uno de los cuts de las tres variantes del programa dado, si se trata de cuts verdes o rojos. Justifique.

Solución

a) Dan true:
?- p(0)
?- p(1)
?- p(4).

b)

i. Son solución:
?- p(0)
?- p(1)

ii. Son solución:
?- p(0)
?- p(1)
?- p(3)
?- p(4)

c)
El cut rojo cambia el significado del programa.
El cut verde no cambia el significado del programa.

Los cut de la primera cláusula de 'p' (en a y en b.i) y la primera cláusula de 'a' (en b.i) son rojos.
Sin esos cut el programa da más soluciones.

El cut de la segunda cláusula de 'p' (en b.ii) es verde.
Si no ponemos ningún cut, las soluciones del programa son las mismas que las que se obtienen para la variante b.ii.

Ejercicio 4 [20 puntos]

a) Considere el siguiente programa R para Prolog puro, proposicional:

```
p :- a,b.
a :- b.
a :- r.
b :- p.
b :- s.
r.
s.
```

i. ¿p es consecuencia lógica de R?
ii. Explique por qué no se obtiene una respuesta 'true' al plantear la consulta ← p.

b) Escriba un meta-intérprete para Prolog proposicional que detecte la repetición de un átomo previamente reducido, evitando así entrar en loop. El meta-intérprete, cada vez que está por entrar en loop, escribe un mensaje con el texto 'cuidado: loop" y el nombre del predicado.
El predicado a implementar es loop_solve(+Obj,+Historia), en la invocación inicial Historia es la lista vacía.

c) Muestre los mensajes escritos ante la consulta loop_solve(p,[]). ¿Es exitosa esta consulta?

d) ¿Cuál es la conducta de su meta-intérprete ante la consulta p(0), si el programa que toma como entrada es:

p(X) :- p(s(X)).

Solución

a.
i. Sí, ya que existe una refutación SLD para p.
ii. El intérprete de Prolog entra en una rama infinita :
p—a,b—b,b—p,b—a,b,b—b,b,b—p,b,b …
antes de encontrar una solución

b.
% loop_solve(+Objetivo,+Historia) ☐ Resuelve Objetivo evitando entrar en loop y emitiendo un mensaje si eso se está por producir, siendo el objetivo y el programa proposicionales. Historia contiene los objetivos intentandos hasta el momento.

```
loop_solve(true,_):- !.
loop_solve((A,B),Hist):-
                !,
                loop_solve(A,Hist),
                loop_solve(B,Hist).
loop_solve(A,Hist) :-
                member(A,Hist),
                writeln('cuidado: loop  '),
                writeln(A),
                !,
                fail.
loop_solve(A,Hist):-
                clause(A,B),
                loop_solve(B,[A|Hist]).
```

c. Consulta: p

```
cuidado: loop
p

cuidado: loop
p
```

La consulta es exitosa.

d.
Este programa es claramente un loop. Pero la lista Historia contiene términos de la forma p(0), p(s(0)), p(s(s(0))), …, de modo que no se detecta la repetición del objetivo.
O sea, el meta-intérprete no detecta este loop y entra él mismo en loop.

Ejercicio 5 **[25 puntos]**

Se piden varios predicados que permiten resolver un sudoku simple (que se define más adelante). Para todo el ejercicio, se asume que M es una matriz N X N, representada del siguiente modo: [Fila1, Fila2, ..., FilaN]. Cada fila es una lista de N elementos.

Escriba los predicados siguientes:

fila_completa_valida(+N, ?F)	<-	F es una lista con valores entre 1 y N sin repetidos. Se asume que al ser invocado este predicado, F es una lista con algunos elementos instanciados y otros sin instanciar.
col_i_esima(+M, +I, -C)	<-	C es la columna I-ésima de M.
sudoku_simple(+N, ?M)	<-	M es una matriz N X N que contiene números de 1 a N. Los números no se repiten en una misma fila ni en una misma columna. M puede estar completamente instanciada o puede contener valores instanciados y variables.

Solución

```
%----
% fila_completa_valida:
% para cada elemento que se agrega a la fila (con between)
% se verifica que no esté entre los que ya se agregaron
% (para esto se agrega un nuevo argumento Ac)
% se usan los predicados between y member de SWI

fila_completa_valida(N, F) :-
      fila_compl_val(F,N,[]).

fila_compl_val([],_,_) :- !. % ! verde
fila_compl_val([E|Es],N,Ac) :-
   between(1,N,E),                      % E es un valor entre 1 y N
   \+member(E,Ac),
   fila_compl_val(Es,N,[E|Ac]).

%----
% columna_i_esima
% Col es la columna I-ésima de la matriz
% se usa el predicado nth1 de SWI

columna_i_esima(_,[],[]) :- !.  % ! verde
columna_i_esima(I,[F|Fs],[E|Col]) :-
   nth1(I,F,E),                         % E es el elem. I-ésimo de F
   columna_i_esima(I,Fs,Col).

%----
% sudoku_simple

sudoku_simple(N,Mat) :-
   filas_completas_validas(N,Mat),      % completa filas sin repetir
   columnas_validas(N,0,Mat).           % chequea validez de columnas

filas_completas_validas(_,[]) :- !.     % ! verde
filas_completas_validas(N,[F|Fs]) :-
   fila_completa_valida(N,F),
   filas_completas_validas(N,Fs).
```

```
% se usa el predicado is_set de SWI para chequear no repetidos
columnas_validas(N,N,_) :- !.       % ! rojo
columnas_validas(N,Pos,M) :-        % N distinto de Pos por ! anterior
    Pos1 is Pos+1,
      columna_i_esima(Pos1,M,Col),
      is_set(Col),
      columnas_validas(N,Pos1,M).
```