Evaluación

Solución

Ejercicio 2

a)

i. **Falso**. Un programa lógico es un conjunto de cláusulas definidas. En consecuencia, siempre tienen un literal positivo (la cabeza de la cláusula). Una interpretación que hace verdadero a todos los literales positivos es un modelo del programa.

ii. **Falso**. Un contraejemplo es P={p(X)←}, G=←p(X). P U {G} es insatisfactible, pero por la parte anterior P es satisfactible pues tiene un modelo.

iii. **Falso**. Sirve el mismo contraejemplo de la parte anterior. La única refutación SLD tiene como respuesta computada a θ_1 = { X/X_1 }; sin embargo, θ_2 = { X/a } es una respuesta correcta.

b) Una interpretación que no es modelo:

$n \rightarrow 0$
$p(A, B, C) \rightarrow \{(A,B,C) \in N^3 / C = A * B\}$ [p «es» el producto]
$q(A, B) \rightarrow \{(A,B) \in N^2 / A = B\}$ [q «es» la igualdad]

Esta interpretación no satisface $\forall X.p(X, n, X)$, puesto que no es cierto que el cero es el neutro del producto.

Una interpretación que sí es modelo:

$n \rightarrow 0$
$p(A, B, C) \rightarrow \{(A,B,C) \in N^3 / C = A + B \}$ [p «es» la suma]
$q(A, B) \rightarrow \{(A,B) \in N^2 / A = B\}$ [q «es» la igualdad]

- La primera cláusula se satisface, puesto que el cero es el neutro de la suma.
- La segunda se satisface, pues la suma es conmutativa.
- La tercera se satisface, pues la suma es asociativa.
- La cuarta se satiface puesto que si X+Z= Y+Z, X debe ser igual a Y (cancelativa).

Ejercicio 3

a) Las soluciones son:

As = [b]	Bs = [b,a]
As = [a,b]	Bs = [b]
As = [b]	Bs =[b]
As = [b,a,b]	Bs = []
As = [a,b]	Bs = []
As = [b]	Bs = []

Como justificación, este es el 'trace' [otra opción es hacer el árbol SLD]:

```
foo([b,a,b], As, Bs)
foo([a,b], As, Bs2)
        foo([b], As, Bs3)
                foo([], As, BS4) Falla
                oof([b], As)
                        Solución As = [b], Bs = [b,a]
                        oof([], As) Falla
        oof([a,b], As)
                Solución As = [a,b], Bs = [b]
                oof([b], As)  (****)
                        Solución As = [b], Bs = [b]
                        oof([], As) Falla
oof([b,a,b], As)
        Solución As = [b,a,b], Bs = []
        oof([a,b], As) (****)
                Solución As = [a,b], Bs = []
                oof([b], As)
                        Solución As = [b], Bs = []
                        oof([], As) Falla
```

b) No es posible. La negación se prueba si el árbol es finitamente fallado. No es el caso porque tiene una rama infinita:

```
foo(Xs, [], [])
oof(Xs, [])
        oof(Xs2, [])
                oof(Xs3, [])
                        ...
```

c)

i. El cut ahí no tiene ningún efecto, porque está al principio del cuerpo de esa cláusula y no hay más cláusulas de *foo*, así que las soluciones son las mismas que en (a).

ii. El cut corta la segunda rama de *oof*; los subárboles marcados con (****) no se recorren y las soluciones son:

```
As = [b]        Bs = [b,a]
As = [a,b]      Bs = [b]
As = [b,a,b]    Bs = []
```

d) El cut de c) i) no corta nada, así que es verde. El cut de c) ii) elimina soluciones, así que es rojo.

Ejercicio 4

a)

```
% resArbol(G,R)<- ver letra.
resArbol(G, R) :-
        cuadrifica(G,GC),               %paso el goal a una lista.
        res(GC, R).                     %resuelvo la lista de átomos

% res(G,R)<- idem a resArbol, pero G es una lista "recta"
res([], [[]]).
res([H|T], [[H|T]|R]) :-                %"guardo" el nodo-sld actual
        clause(H, Cuerpo),              % busco cláusula que unifique
        cuadrifica(Cuerpo,CuerpoC),     % paso el lado derecho a una lista
        append(CuerpoC, T, NuevoG),     % hago la reducción
        res(NuevoG, R).                 % resuelvo el nuevo objetivo

% cuadrifica(Xs,CXs)<- CXs es la representación en lista de la lista curva Xs.
cuadrifica((A,B),[A|CB]):-
        !,
        cuadrifica(B,CB).
cuadrifica((true),[]):- !.
cuadrifica((A),[A]).

% append(Xs,Ys,Zs)<- Zs es la concatenación de Xs con Ys.
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-
        append(Xs,Ys,Zs).
```

b)

```
sentencias --> [].
sentencias --> sent, sentencias.
sent --> sent_while.
sent --> sent_asignacion.
sent_while --> [while,'('], explog, [')',do], sentencias, [endw,';'].
sent_asignacion --> variable, [':='], explog, [';'].
explog --> operando.
explog --> operando, [and], explog.
explog --> operando, [or], explog.
explog --> [not,'('], explog, [')'].
operando --> variable.
operando --> [true].
operando --> [false].
variable --> [Var], {es_identificador(Var)}.

% es_identificador(Var) <-     Var es un identificador si cumple con las reglas de formación
%                              de identificadores del lenguaje reconocido.
```

Ejercicio 5

```
% novueltan(+Tab) <- ver letra
novuelan(tab(_, _, Oc)) :-
        novuelan(Oc, Oc).

% novuelan(+Sust,+Ocs) <- Sust es una lista con lugares sustentados por Ocs
novuelan([], _).
novuelan([oc(F, C, _)|T], Oc) :-                % recorro las fichas, probando una a una
        novuela(F,C,Oc),!,
        novuelan(T, Oc).

% novuela(+F,+C,+Ocs) <- la posición (F,C) esta "sostenida" por alguna ficha de Ocs, o es la primera
novuela(1, _, _):-!.                            %es la primera de la fila
novuela(F, C, Oc) :-
        F>1,
        F1 is F-1,                              % tiene otra abajo que la sustenta
        member(oc(F1, C, _), Oc).


% otra solución
novuelan(tab(_, _, Oc)) :-
        \+ ((member(Pos,Oc), flota(Pos, Oc))).  % no existe una ficha que flote

flota(oc(Fila,Col,_),Oc):-
        Fila>1,                                 % una ficha flota si hay un "hueco" debajo de ella
        Fila1 is Fila-1,
         \+ member(oc(Fila1,Col,_),Oc).


% siguientes(+T,?Movs) <- ver letra
siguientes(T, Movs) :-
        findall(Mov, siguiente(T, Mov), Movs).

% siguiente(+T,?Mov) <- Mov una posible siguiente movida en el tablero T.
siguiente(tab(NF, NC, Oc), pos(F, C)) :-
        between(1, NC, C),                      %elijo una coordenada del tablero
        between(1, NF, F),
        \+ member(oc(F,C,_), Oc),               %reviso que no esté ocupada
        novuela(F, C, Oc).                      %reviso que tenga sustento

% klinea(+T,K,Color) <- ver letra
klinea(tab(_, _, Oc), K, Color) :-
        member(oc(Fil,Col,Color),Oc),   %(Fil,Col) es la posición a partir de la que está la klinea
        direccion(DX, DY),              % dirección de búsqueda
        klin(K, Fil,Col, Color, Oc, DX, DY).

% direccion(-DX,-DY) <- DX y DY son la diferencia de las filas y columnas, respectivamente,
%                       de la coordenada siguiente y la actual
direccion(1,0).         % hacia arriba
direccion(0,1).         % hacia la derecha
direccion(1,1).         % diagonal hacia arriba a la derecha
direccion(1,-1).        % diagonal hacia arriba a la izquierda

%klin(K,Fil,Col,Color,Ocs,DX,DY)<-hay K fichas de Color en Ocs a partir de (Fil,Col) en dirección (DX,DY)
klin(0, _, _, _, _, _, _):-!.
klin(K, Fil, Col, Color, Ocs, DX, DY) :-
        K>0,
        member(oc(Fil, Col, Color), Ocs),       %el elemento es del color buscado.
        K1 is K-1,
        C1 is Col+DX,                           %avanzo en la dirección buscada
        F1 is Fil+DY,
        !,
        klin(K1,F1, C1, Color,Ocs,DX,DY).       %busco las k-1 fichas restantes
```

```
% para resolver el predicado jugada se tomó la opción de devolver primero todas las jugadas ganadoras
% y luego todas las no tan buenas (en backtracking)
% una solución "rápida" para no dar las no tan buenas en backtracking si existen ganadoras es agregar
% al comienzo de la segunda cláusula \+ ((siguiente(Tab,Pos),gana(Tab,K,Pos,Color)), con esto se
% podría quitar en esa misma cláusula \+ gana(Tab, K, Pos, Color),

%jugada <- ver letra
jugada(Tab, K, Color, Pos) :-
        siguiente(Tab, Pos),            %elijo una jugada posible
        gana(Tab, K, Pos, Color).       %verifico que gane

jugada(Tab, K, Color, Pos) :-
        contrario(Color,Color2),
        siguiente(Tab, Pos),
        \+ gana(Tab, K, Pos, Color),            %elijo una jugada posible que no gano
        siguiente_tab(Tab, Pos, Color, TNew), %simulo la jugada
        %no se cumple que el contrario juegue y gane
        \+((siguiente(TNew, PosCon),gana(TNew, K, Color2, PosCon))).


% siguiente_tab(T, Pos, Color, TNew)<- TNew es el tablero que se obtiene si Color juega Pos en Tab.
siguiente_tab(tab(NF, NC, Oc), pos(Fil,Col), Color, tab(NF,NC,[oc(Fil,Col,Color)|Oc])).

% gana(+T, +K, +Pos, +Color)<- Color gana si juega Pos en Tab.
gana(T, K, Pos, Color):-
        siguiente_tab(T, Pos, Color, TNew),     %simulo la jugada
        klinea(TNew, K, Color).                 %verifico k en linea

% contrario(Color,Color2) <- Color2 es el contrario de Color.
contrario(b, n).
contrario(n, b).

%member(?X,?Xs) <- X es un elemento de Xs
member(X,[X|_]).
member(X,[_|Ys]):- member(X,Ys).

% between(+N,+M,?P) <- N =< P =< M
between(N,M,N):-N=<M.
between(N,M,P):-N<M, K is N+1, between(K,M,P).
```