Examen
de
Programación Declarativa

(19 de enero de 2009)

Andrés Cordón Franco

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, enero de 2009

Ejercicio 1 ¿Qué responde Haskell al evaluar las siguientes expresiones?

1. `:type zip`
2. `map (mod 3) [1..5]`
3. `takeWhile (< 500) (map (2^) [1..])`
4. `[x > 5 | x <- [2..7]]`

Solución:

1. `zip ::  [a] -> [b] -> [(a,b)]`
2. `[0,1,0,3,3]`
3. `[2,4,8,16,32,64,128,256]`
4. `[False,False,False,False,True,True]`

Ejercicio 2 Se define en Haskell por recursión la función:

```
existePar p (x:y:xs) = p x y || existePar p (y:xs)
existePar p _ = False
```

1. Determina el tipo inferido por Haskell para la función `existePar`.
2. Escribe una definición *no recursiva* para la función `existePar`.

Solución:

1. `existePar ::  (a -> a -> Bool) -> [a] -> Bool`
2. `existePar p xs = or [p x y | (x,y)<-zip xs (tail xs)]`

Ejercicio 3 Se define el tipo de datos `Grafo a`

```
data Grafo a = G [a] [(a,a)]
```

donde (G xs ys) representa el grafo *no dirigido* cuyos vértices son los elementos de xs y cuyas aristas son los elementos de ys.

1. Define en Haskell la función:

   `grado :: Eq a => Grafo a -> a -> Int`

   tal que `(grado g x)` devuelve el grado del vértice x en el grafo g (es decir, el número de aristas de g que inciden en x). Si x no es un vértice de g, se devolverá (-1). Por ejemplo:

   `(grado (G [1,2,3,4] [(1,2),(3,1),(2,3)]) 1)` ⇝ `2`

   `(grado (G [1,2,3,4] [(1,2),(3,1),(2,3)]) 5)` ⇝ `-1`

2. Define en Haskell la función:

   `prop :: Eq a => Grafo a -> Bool`

   tal que `(prop g)` se verifica si la suma de los grados de los vértices de g coincide con el doble del número de aristas de g. Por ejemplo:

   `(prop (G [1,2,3,4] [(1,2),(3,1),(2,3)])` ⇝ `True`

Solución:

1. ```
   grado (G xs ys) x |notElem x xs = -1
                     |otherwise    = length [x | (a,b)<-ys, a==x||b==x]
   ```

2. ```
   prop g@(G xs ys) = sum (map (grado g) xs) == 2*length ys
   ```

Ejercicio 4 Determina si al responder Prolog a las siguientes preguntas tiene éxito o falla. Si tiene éxito, escribe los valores que asigna a las variables. Si falla, explica la razón del fallo.

1. `?- [a,b,c] = [A|B].`
2. `?- [a,b,c] == [A|B].`
3. `?- [a,b,c] =.. [A|B].`
4. `?- findall(X,append(_,X,[a,b,c]),L).`
5. `?- member(X,[a,b,c]), not(member(X,[a,b])).`
6. `?- not(member(X,[a,b])), member(X,[a,b,c]).`

Solución:

1. `A = a B = [b, c]`
2. Falla, puesto que `T1 == T2` tiene éxito si T1 y T2 son términos *idénticos*.
3. `A = '.' B = [a, [b, c]]`
4. `L = [[a, b, c], [b, c], [c], []]`
5. `X = c`
6. Falla. La pregunta `?-member(X,[a,b])` tiene éxtito y, por tanto, la pregunta `?-not(member(X,[a,b]))` falla. En consecuencia, para cualquier `P`, la pregunta `?-not(member(X,[a,b])),P` también falla.

Ejercicio 5 Se considera el siguiente programa lógico:

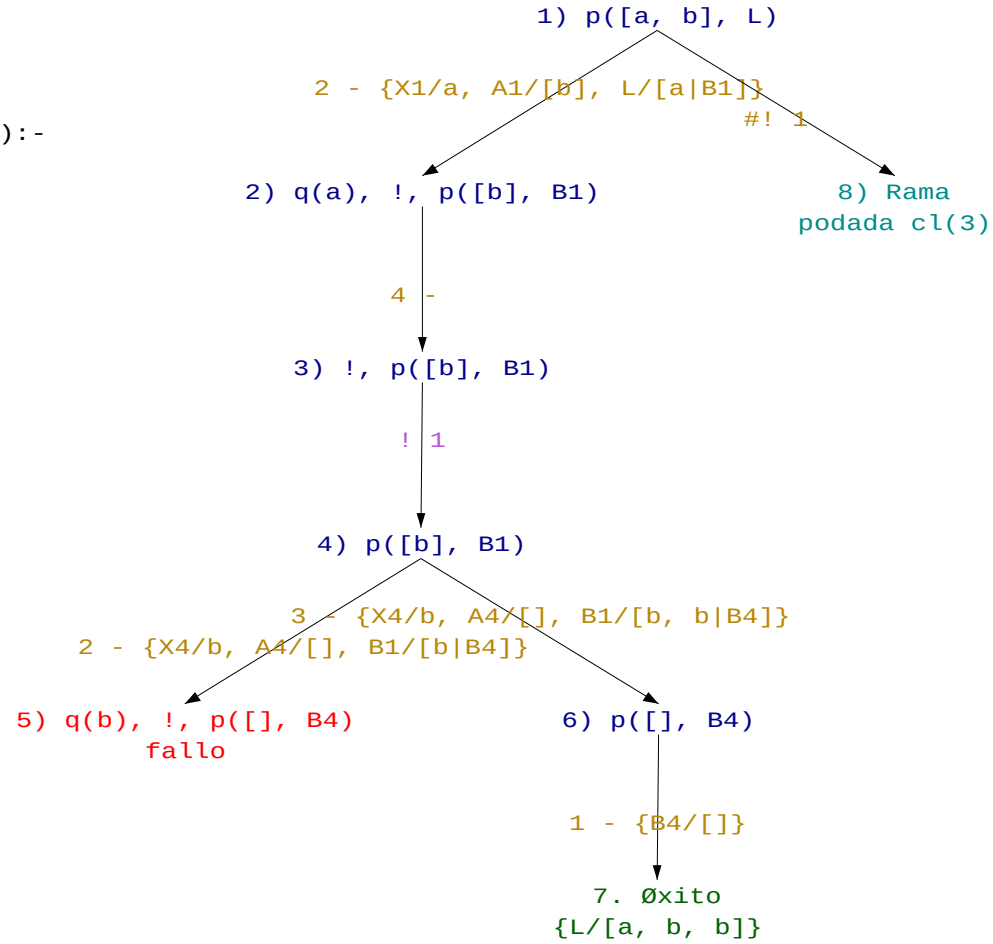
```
p([],[]).                              %R1
p([X|A],[X|B]) :- q(X),!,p(A,B).       %R2
p([X|A],[X,X|B]) :- p(A,B).            %R3
q(a).                                  %R4
```

Construye el árbol de resolución para el programa anterior y la pregunta:

```
?- p([a,b],L).
```

Solución:

```
1 p([], []).
2 p([X|A], [X|B]):-
        q(X),
        !,
        p(A, B).
3 p([X|A], [X, X|B]):-
        p(A, B).
4 q(a).
```



Ejercicio 6

1. Escribe una definición *recursiva* en Prolog del predicado `copia(+X,+N,-L)` que se verifica si la lista L contiene N copias de X. Por ejemplo:

```
?- copia(a,3,L).
L = [a,a,a]
```

2. Define en Prolog el predicado `replica(+L1,+N,-L2)` que se verifica si L2 es la lista obtenida al copiar N veces cada elemento de L1. Por ejemplo:

```
?- replica([a,b,c,d],2,L).
L = [a,a,b,b,c,c,d,d]
```

Solución:

1.
```
copia(_,0,[]).
copia(X,N,[X|L]) :- N > 0,
                    N1 is N-1,
                    copia(X,N1,L).
```

2.
```
replica([],_,[]).
replica([X|L1],N,L2) :- copia(X,N,L3),
                        replica(L1,N,L4),
                        append(L3,L4,L2).
```