PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL3: El lenguaje PROLOG, aspectos avanzados

1. El predicado de negación

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

DEFINICIÓN

- La sintaxis de PROLOG estudiada en el tema anterior sólo permite trabajar con información *positiva*.
- Para poder expresar o deducir información negativa se utiliza el operador prefijo `\+`, aplicable a cualquier predicado `P`, escribiendo `\+ P` y leyendo `no P`, siempre que `P` forme parte del cuerpo de una regla o de una una consulta (ni los hechos ni las cabezas de las reglas se pueden negar).
- Para ello se extiende la Programación Lógica Definida pasando a la denominada *Programación Lógica General*, cuyo sistema de demostración, la Resolución SLDNF, implementa una forma de negación conocida como negación por fallo finito. Su funcionamiento, *distinto* al de la negación de la Lógica Matemática, se explica más adelante.

Ejemplos (Negación en programas, 1/2)

- Todos los pájaros vuelan, a menos (= salvo) que sean especiales.
  - Formalización en LPO:
    $\forall X[(\mathit{pájaro}(X) \wedge \neg \mathit{especial}(X)) \rightarrow \mathit{vuela}(X)]$
  - Formalización en PROLOG:

```
vuela(X) :-
    pájaro(X),
    \+ especial(X).
```

- Abel es amigo de todos los que son amigos de Caín y no son enemigos de Abilio.
  - Formalización en LPO:
    $\forall X[(\mathit{amigo}(X, \mathit{cain}) \wedge \neg \mathit{enemigo}(X, \mathit{abilio})) \rightarrow \mathit{amigo}(\mathit{abel}, X)]$
  - Formalización en PROLOG:

```
amigo(abel,X) :-
    amigo(X, cain),
    \+ enemigo(X, abilio).
```

Ejemplos (Negación en programas, 2/2)

- Nadie es enemigo de sí mismo.
  - Formalización en LPO:
    $\forall X(\neg enemigo(X, X))$
  - Formalización en PROLOG:
    No admisible en PROLOG
    (los hechos no se pueden negar)
- Los que son amigos de Abel no son enemigos de Abilio.
  - Formalización en LPO:
    $\forall X[(amigo(X, abel) \rightarrow \neg enemigo(X, abilio)]$
  - Formalización en PROLOG:
    No admisible en PROLOG
    (las cabezas de las reglas no se pueden negar)

Ejercicios (Negación en programas)

Suponga disponibles los predicados `amigo(X,Y)`*, cierto si X es amigo de Y, y* `enemigo(X,Y)`*, cierto si X es enemigo de Y.*

Para cada una de las siguientes frases:

- *Formalícela en Lógica de Primer Orden (LPO).*
- *Razone si podría formar parte de un programa en* PROLOG*, y, en caso afirmativo, facilite su escritura en* PROLOG*.*

1. *Los que no son amigos ni de Abel ni de Caín son amigos de Abilio.*
2. *Nadie es amigo de Caín y de Abel a la vez.*
3. *Los que no son amigos de Abel tampoco son amigos de Abilio.*
4. *Caín no es amigo de nadie.*
5. *Abel es amigo de todos salvo de los que son amigos de Caín.*

Soluciones propuestas:

1. Los que no son amigos ni de Abel ni de Caín son amigos de Abilio.
   LPO: $\forall X[(\neg amigo(X, abel) \wedge \neg amigo(X, cain)) \rightarrow amigo(X, abilio)]$
   PROLOG: *amigo(X, abilio) :- \+ amigo(X, abel), \+ amigo(X, caín).*

2. Nadie es amigo de Caín y de Abel a la vez.
   LPO: $\forall X[amigo(X, abel) \rightarrow \neg amigo(X, cain))]$, o, equivalentemente,
   gracias a las equivalencias lógicas $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_2 \rightarrow \neg\varphi_1$ (ley contrapositiva)
   y $\neg\neg\varphi \equiv \varphi$ (ley de la doble negación), a
   $\forall X[amigo(X, cain) \rightarrow \neg amigo(X, abel))]$
   PROLOG: no admisible en PROLOG (regla con cabeza negada)

3. Los que no son amigos de Abel tampoco son amigos de Abilio.
   LPO: $\forall X[\neg amigo(X, abel) \rightarrow \neg amigo(X, abilio)]$, o, equivalentemente,
   $\forall X[amigo(X, abilio) \rightarrow amigo(X, abel)]$, gracias a la ley contrapositiva.
   PROLOG: *amigo(X, abel) :- amigo(X, abilio).*

4. Caín no es amigo de nadie.
   LPO: $\forall X[\neg amigo(cain, X)]$
   PROLOG: no admisible en PROLOG (hecho negado)

5. Abel es amigo de todos salvo de los que son amigos de Caín.
   LPO: $\forall X[\neg amigo(X, cain) \rightarrow amigo(abel, X)]$
   PROLOG: *amigo(abel, X) :- \+ amigo(X, cain).*

Ejemplos (Negación en consultas)

- ¿Existe algún pájaro que no vuele? ¿Cuál o cuáles?
  - Formalización en LPO: $\exists X(\mathit{pájaro}(X) \wedge \neg \mathit{vuela}(X))$
  - Formalización en PROLOG:

```
?- pájaro(X), \+ vuela(X).
```

- ¿Es cierto que Abel no es amigo de Caín?
  - Formalización en LPO: $\neg \mathit{amigo}(\mathit{abel}, \mathit{cain})$
  - Formalización en PROLOG:

```
?- \+ amigo(abel, cain).
```

- ¿Es cierto que Abel es amigo de todos los amigos de Caín?
  - Formalización en LPO de la negación de la consulta: $\exists X(\mathit{amigo}(X, \mathit{cain}) \wedge \neg \mathit{amigo}(\mathit{abel}, X))$.
  - Formalización en PROLOG (de la negación de la consulta):

```
?- amigo(X,cain), \+ amigo(abel, X).
```

- ¿Qué animales existen que no le gusten a Adán?
  - Formalización en LPO: $\exists X(\mathit{animal}(X) \wedge \neg \mathit{gusta}(X, \mathit{adan}))$
  - Formalización en PROLOG:

```
?- animal(X), \+ gusta(X, adan).
```

Ejercicios (Negación en consultas)

Suponga disponibles los predicados `amigo(X,Y)`*, cierto si X es amigo de Y, y* `enemigo(X,Y)`*, cierto si X es enemigo de Y.*

Para cada una de las siguientes preguntas:

- *Formalícela en Lógica de Primer Orden (LPO).*
- *Escriba la correspondiente consulta en* PROLOG*.*

1. *¿Qué personas hay que no sean amigas ni de Caín ni de Abel?*
2. *¿Hay alguien -no importa quién sea- que no sea enemigo de Caín?*
3. *¿Es cierto que Abel no es enemigo de Caín?*
4. *¿Hay alguien que no sea amigo de Abel y sea enemigo de Caín? ¿Quién o quiénes son?*
5. *¿Es cierto que todos los amigos de Abel son enemigos de Caín?*

Soluciones propuestas:

1. ¿Qué personas hay que no sean amigas ni de Caín ni de Abel?

   LPO: $\exists X[\neg amigo(X, cain) \wedge \neg amigo(X, abel)]$
   PROLOG*: ?- \+ amigo(X, cain), \+ amigo(X, abel).*

2. ¿Hay alguien -no importa quién sea- que no sea enemigo de Caín?

   LPO: $\exists X[\neg enemigo(X, cain)]$
   PROLOG*: ?- \+ enemigo(_, cain).*

   % uso de la variable anónima _

3. ¿Es cierto que Abel no es enemigo de Caín?

   LPO: $\neg enemigo(abel, cain)$
   PROLOG*: ?- \+ enemigo(abel, cain).*

4. ¿Hay alguien que sea enemigo de Caín y no sea amigo de Abel? ¿Quién o quiénes son?

   LPO: $\exists X[enemigo(X, cain) \wedge \neg amigo(X, abel)]$
   PROLOG*: ?- enemigo(X, cain), \+ amigo(X, abel).*

5. ¿Es cierto que todos los amigos de Abel son enemigos de Caín?

   LPO de la negación: $\exists X[amigo(X, abel) \wedge \neg enemigo(X, cain)]$
   PROLOG *(de la negación): ?- amigo(X,abel), \+ enemigo(X, cain).*

IMPLEMENTACIÓN Y FUNCIONAMIENTO

Implementación

El predicado `\+` está implementado internamente mediante las dos siguientes cláusulas (una regla y un hecho):

```
\+ P :-
    P,
    !,
    fail.


\+ P.
```

donde `!` es el predicado de corte y `fail` es un predicado predefinido que siempre falla (siempre devuelve *false*).

Funcionamiento

Ante un objetivo de la forma `\+ P`, PROLOG aplica la implementación anterior, y el resultado será el siguiente:

- PROLOG devolverá cierto si `P` no se puede probar, es decir, si al realizarse la consulta "`?- P.`" se obtiene un Árbol de Resolución finito y con todas sus ramas fallo.
- Devolverá falso si `P` es cierto, es decir, si al realizarse la consulta "`?- P.`" se obtiene al menos un éxito (el Árbol de Resolución contiene al menos una rama éxito más a la izquierda que cualquier rama infinita).
- No terminará si la consulta "`?- P.`" no termina (es decir, si al realizarse esta consulta se obtiene un Árbol de Resolución conteniendo una rama infinita sin ninguna rama éxito a su izquierda).

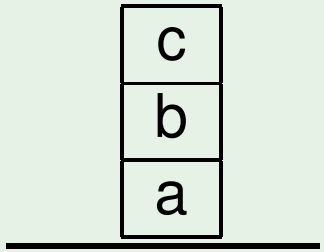
En efecto, dada la ▸ implementación del predicado `\+ P`, los nodos de los Árboles de Resolución de la forma `?- \+ P, O1, ..., On.` siempre tendrán los dos hijos siguientes (con u.m.g.'s vacíos):

1. El hijo izquierdo, obtenido con la regla `\+ P :- P, !, fail.`, será `?- P, !, fail, O1, ..., On.` y:
   - Si `P` resulta ser cierto, la respuesta será `false`, puesto que la combinación del corte y el `fail` posteriores hará que todos los hijos del nodo `?- \+ P, O1, ..., On.` sean ramas fallo o podadas (ver ejemplo (3/4) a continuación).
   - Si por el contrario `P` resulta ser falso (todos los posibles intentos de probar `P` fracasan), no se llega a ejecutar ni el corte ni `fail` y PROLOG retrocede para entrar por el hijo derecho (ver ejemplo (2/4) a continuación).
2. El hijo derecho, obtenido con el hecho `\+ P.`, será `?- O1, ..., On.` Sólo se llegará a él si `P` es falso, y el resultado dependerá de la evaluación de `O1, ..., On`.

Ejemplo (El mundo de los bloques -Blocks World-, 1/4)

Ejemplo clásico en Inteligencia Artificial para ilustrar problemas de planificación y robótica.

```
% encima(X,Y)
% cierto si X está justo encima de Y
encima(b,a).
encima(c,b).

% apilado(X,Y)
% cierto si X está apilado sobre Y
apilado(X,Y) :-  % justo encima
    encima(X,Y).

apilado(X,Y) :-  % más arriba
    encima(X,Z),
    apilado(Z,Y).
```

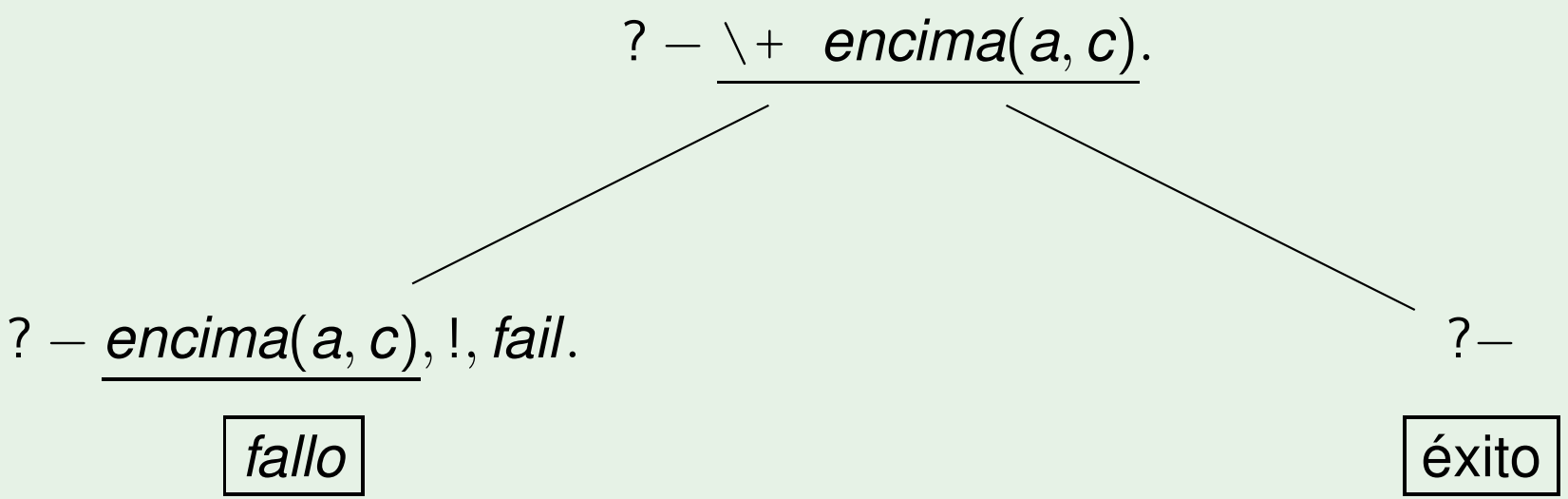


c está encima de b y apilado sobre él

c está apilado sobre a pero no encima suyo

Ejemplo (El mundo de los bloques, 2/4)

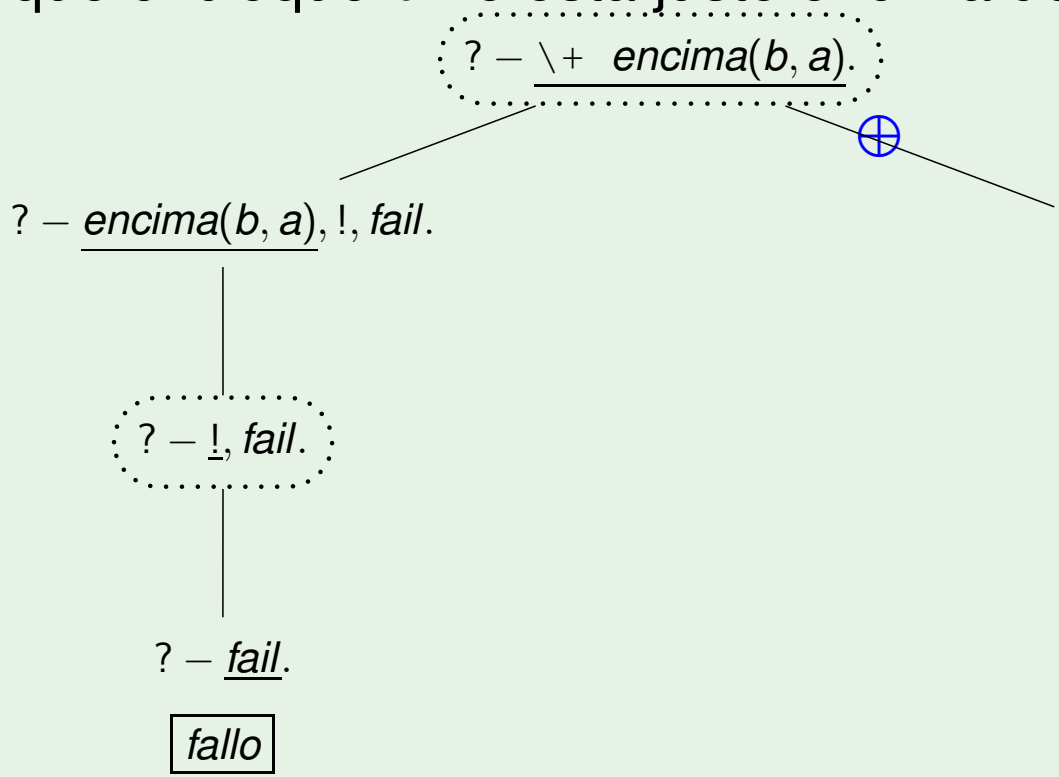
¿Es cierto que el bloque `a` no está justo encima del bloque `c`?



A la vista del árbol anterior, la respuesta de PROLOG es `true`: efectivamente, el bloque `a` no está justo encima del bloque `c`.

Ejemplo (El mundo de los bloques, 3/4)

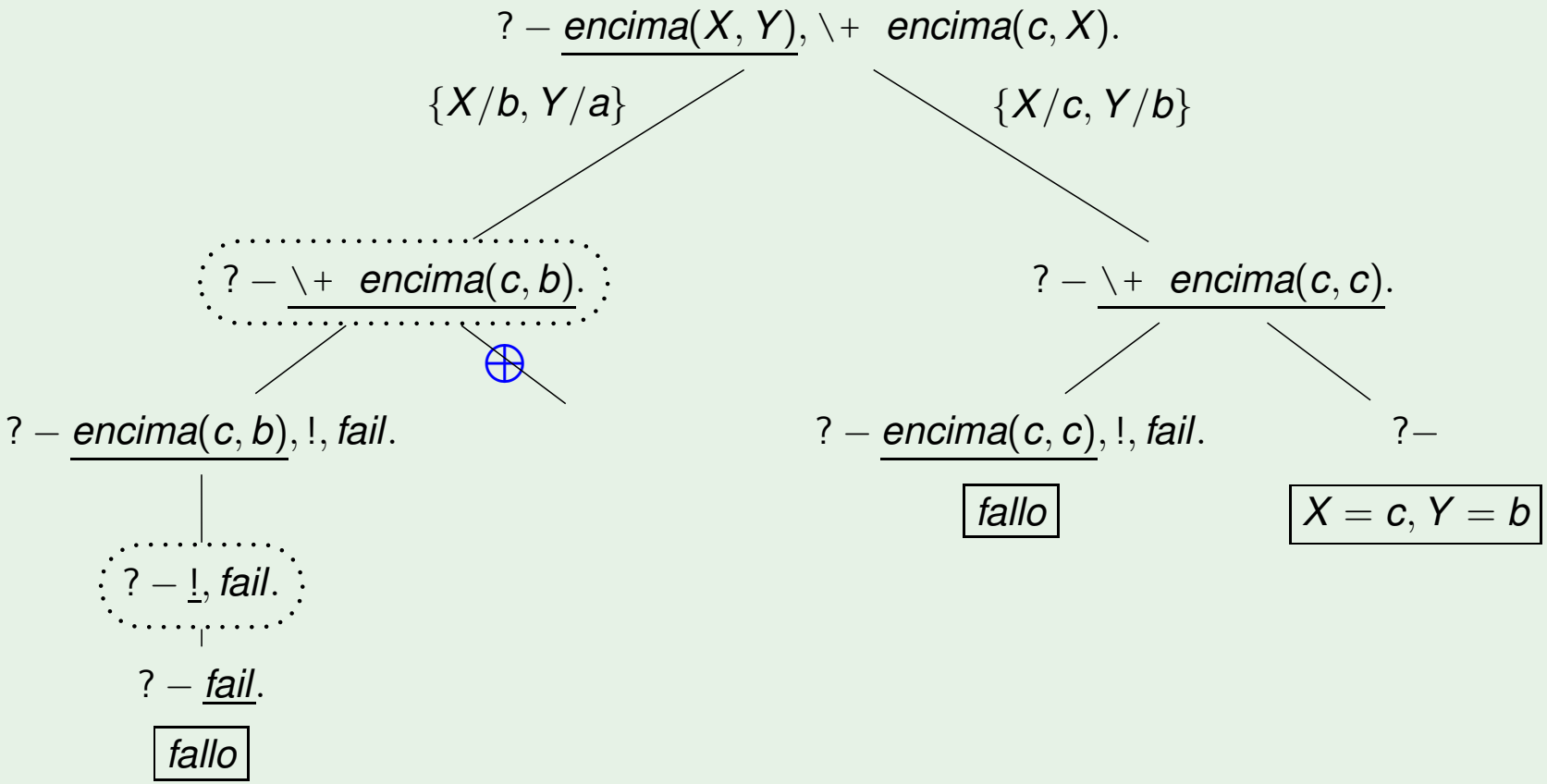
¿Es cierto que el bloque `b` no está justo encima del bloque `a`?



A la vista del árbol anterior, la respuesta de PROLOG es `false`: efectivamente, no es cierto que el bloque `b` no está justo encima del bloque `a`, porque sí que lo está.

Ejemplo (El mundo de los bloques, 4/4)

¿Existen dos bloques tales que el primero está justo encima del segundo y el bloque c no está justo encima del primero?



Respuesta de PROLOG: sí, y la única solución es X=c, Y=b.

Ejercicios (Árboles de Resolución con negaciones)

Considere el programa en PROLOG *relativo al mundo de los bloques del último ejemplo.*

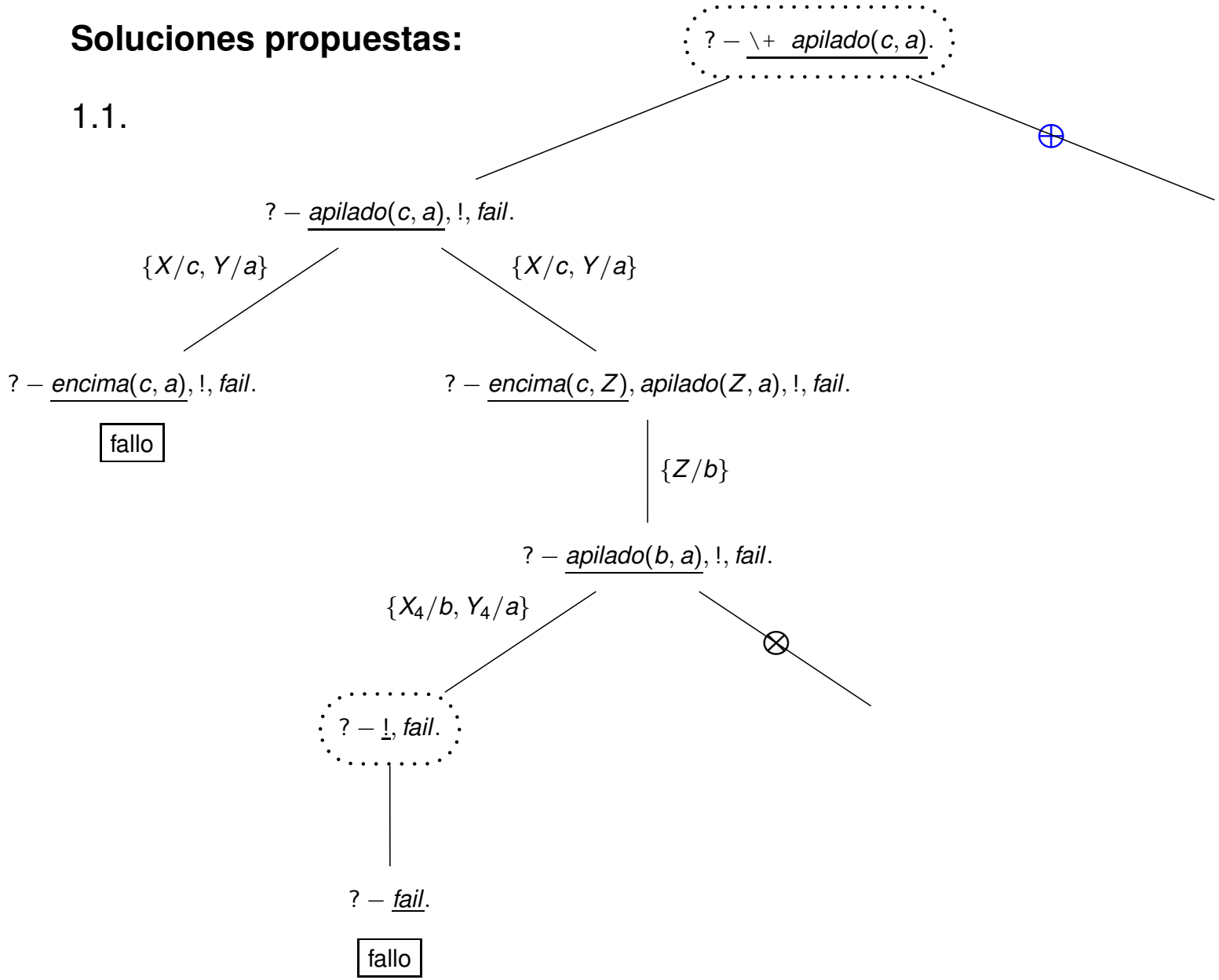
1. *Escriba en* PROLOG *cada una de las consultas que se describen a continuación y construya los Árboles de Resolución adecuados para averiguar qué respuestas ofrecería* PROLOG*, y en qué orden.*
   1. *¿Es cierto que el bloque c no está apilado sobre el bloque a?*
   2. *¿Es cierto que el bloque b no está apilado sobre el bloque c?*
   3. *¿Qué bloques hay tales que están apilados sobre el bloque a pero no están justo encima suyo?*
2. *Reflexione sobre qué respuestas ofrecería* PROLOG *ante la consulta*

   ```
   ?- apilado(X,a), !, \+ encima(X,a).
   ```

   Compruebe su respuesta construyendo el Árbol de Resolución correspondiente.

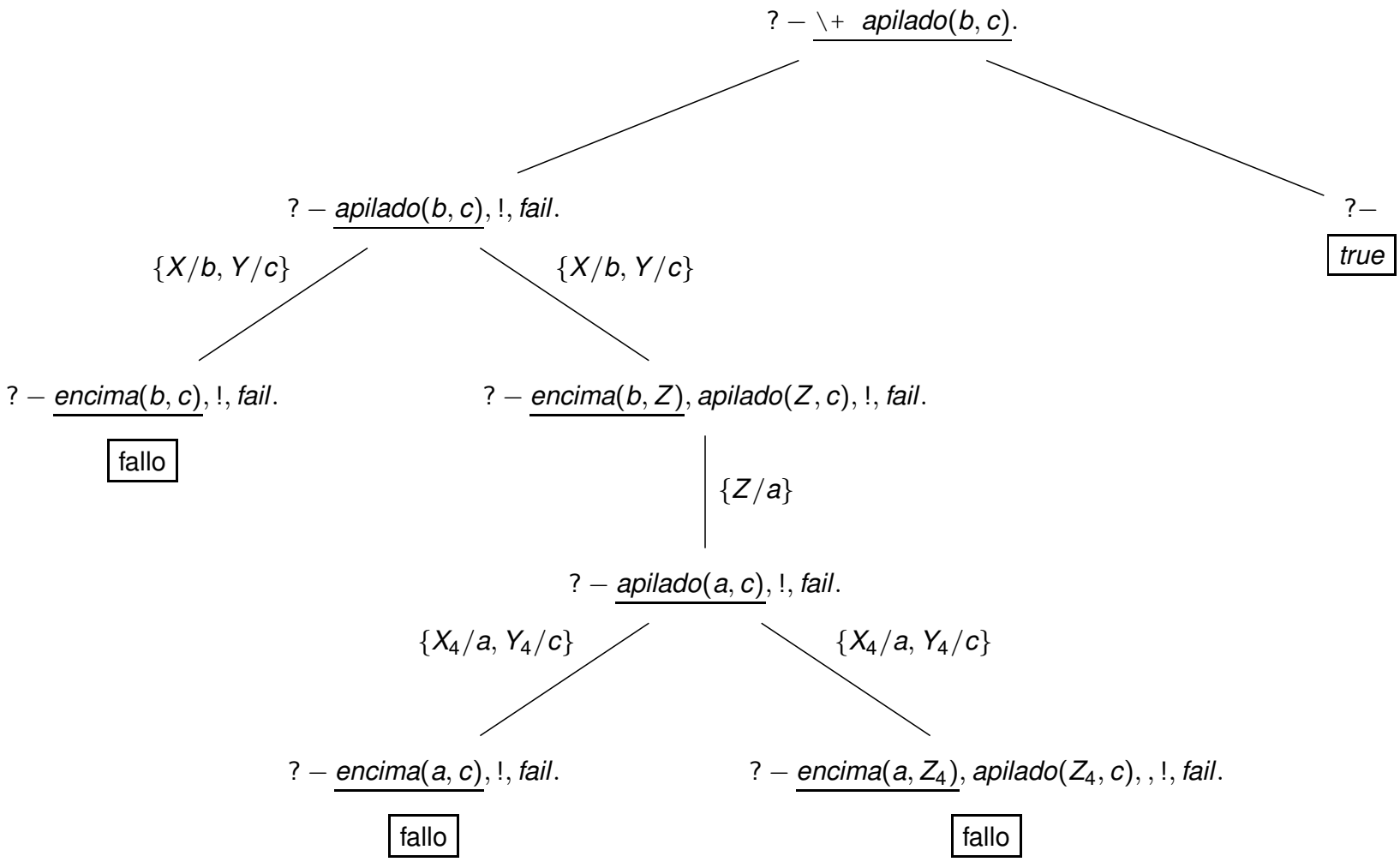
Soluciones propuestas:

1.1.



A la vista del Árbol anterior, la respuesta de PROLOG es `false`: efectivamente, es falso que c no está apilado sobre a, ¡sí que lo está!

1.2.



A la vista del Árbol anterior, la respuesta de PROLOG es `true`: efectivamente, es cierto que b no está apilado sobre c.

1.3. La consulta correspondiente en PROLOG sería `?- apilado(X,a), \+ encima(X,a)`, dando como única solución `X=c`. El Árbol de Resolución asociado, de izquierda a derecha en profundidad, consta de las siguientes ramas: rama fallo, rama podada, tres ramas fallo, rama éxito con solución `X=c` y dos ramas fallo.

2. Esta consulta es la misma que la última del punto anterior, ¿qué bloques están apilados sobre a pero no están justo encima suyo?, pero de forma que el corte impide la reevaluación del primer subobjetivo, `apilado(X,a)`. Esto hace que la consulta falle, puesto que la primera -y única, debido al corte- respuesta que encuentra `apilado(X,a)` es `X=b`, el bloque que está justo encima suyo, por lo que el siguiente subobjetivo, `\+ encima(b,a)`, falla.

NO MONOTONÍA

El predicado `\+` se basa en la Hipótesis del Mundo Cerrado (Closed Word Assumption, CWA) y permite implementar Razonamiento No Monótono (Non Monotonic Reasoning, NMR).

- Hipótesis del Mundo Cerrado.

  Si de una base de conocimientos no se puede deducir un cierto hecho `P`, entonces ese hecho se puede considerar falso, y por lo tanto su negación, `\+ P`, se puede considerar cierta.

- Razonamiento No Monótono.

  Razonamiento en el que un aumento de las premisas puede provocar una disminución de las conclusiones, cosa que nunca ocurre en el razonamiento de la Lógica Matemática, que es monótono.

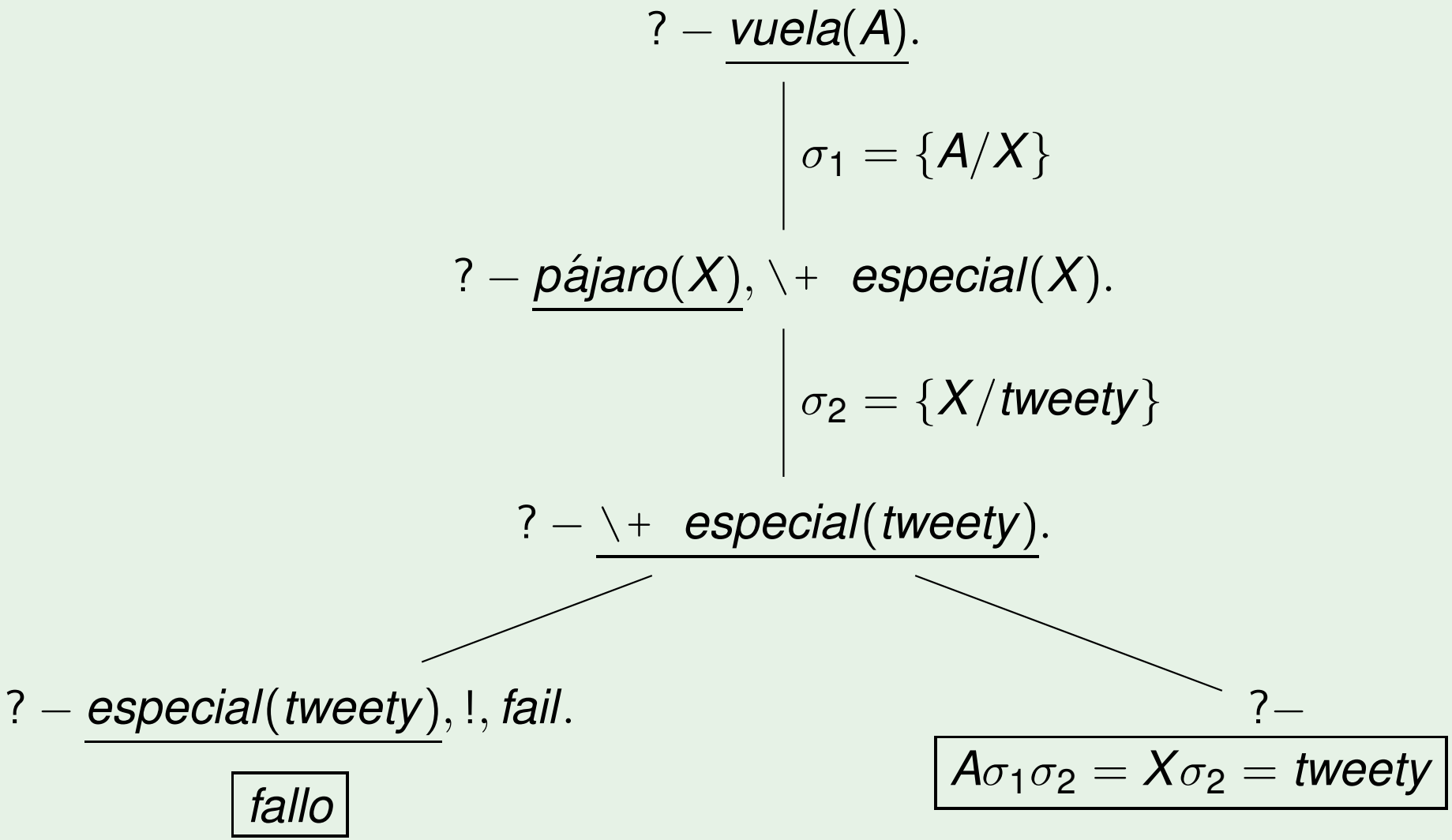
Ejemplo (El pájaro Tweety, 1/3)

Ejemplo clásico en IA para ilustrar razonamiento no monótono.

```
%todos los pájaros vuelan, salvo que sean especiales
        vuela(X) :-
            pájaro(X),
            \+ especial(X).
% tweety es un pájaro
        pájaro(tweety).
% pepito es especial
        especial(pepito).
```

Con la información anterior, PROLOG puede concluir que `tweety` vuela (ver Árbol de Resolución a continuación), dado que sabe que es un pájaro y, *al no poder demostrar que sea especial*, deduce que no lo es (en Lógica Matemática no sería posible demostrar que `tweety` no es especial y por lo tanto no se podría deducir que vuela).

Ejemplo (El pájaro Tweety, 2/3)



Ejemplo (El pájaro Tweety, 3/3)

Si el programa relativo al pájaro Tweety se extiende con las dos siguientes cláusulas adicionales:

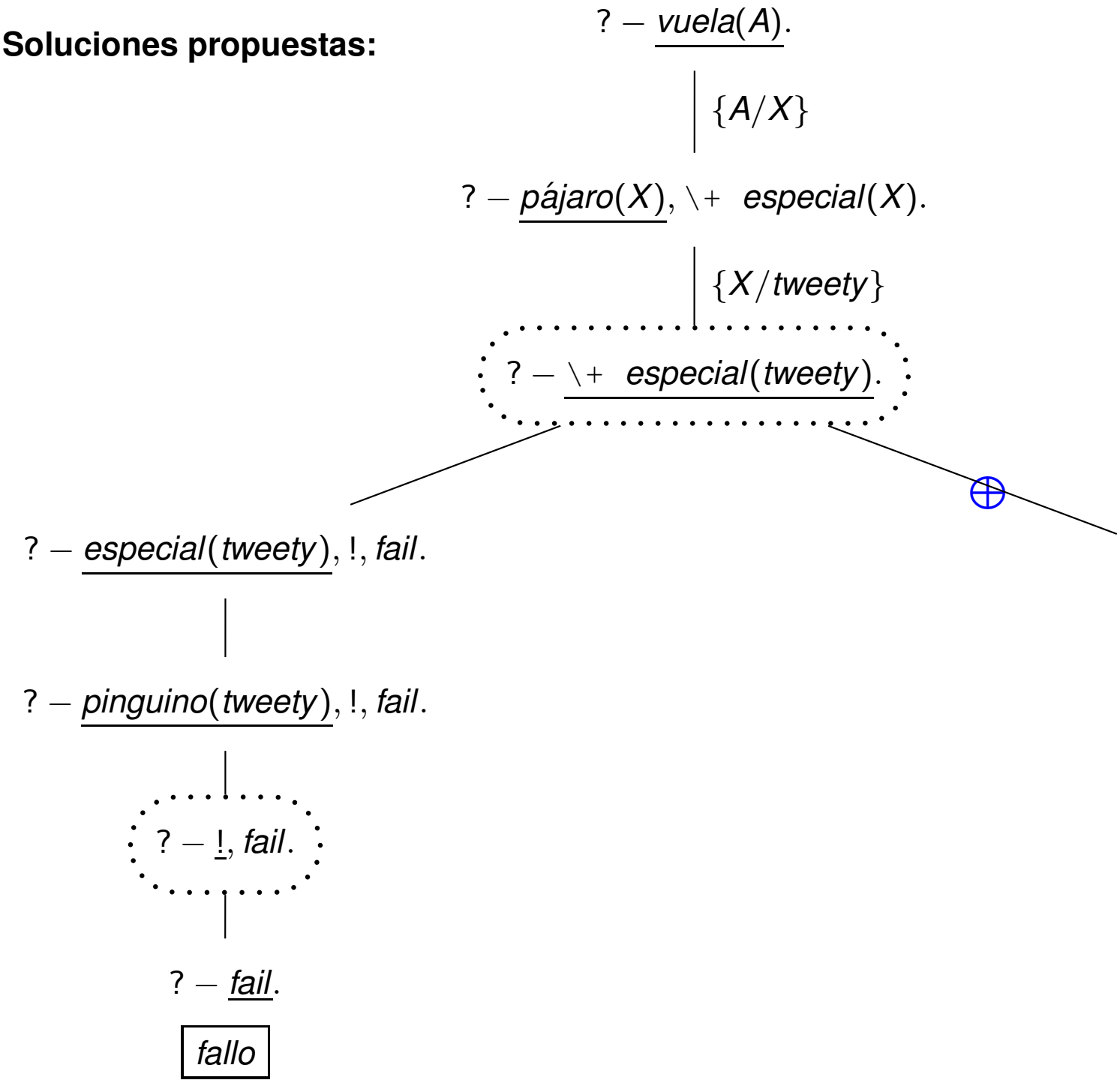
```
especial(X) :-
    pingüino(X).

pingüino(tweety).
```

ya no es posible deducir que Tweety puede volar, puesto que ahora sí se puede demostrar que Tweety es especial. Se trata de un ejemplo de razonamiento no monótono: un aumento de las premisas obliga a retractar una de las conclusiones obtenidas previamente.

Ejercicios

Compruebe lo anterior haciendo el Árbol de Resolución para `?- vuela(A)` *una vez añadidas las nuevas cláusulas.*

Soluciones propuestas:



LIMITACIONES DEL PREDICADO DE NEGACIÓN

El predicado de negación puede funcionar de forma distinta a la esperada en ciertas ocasiones $\Rightarrow$ ¡debe usarse con cuidado!

1. La invocación de un objetivo del tipo "`\+ P`" no asigna valores a variables, es decir, no computa.

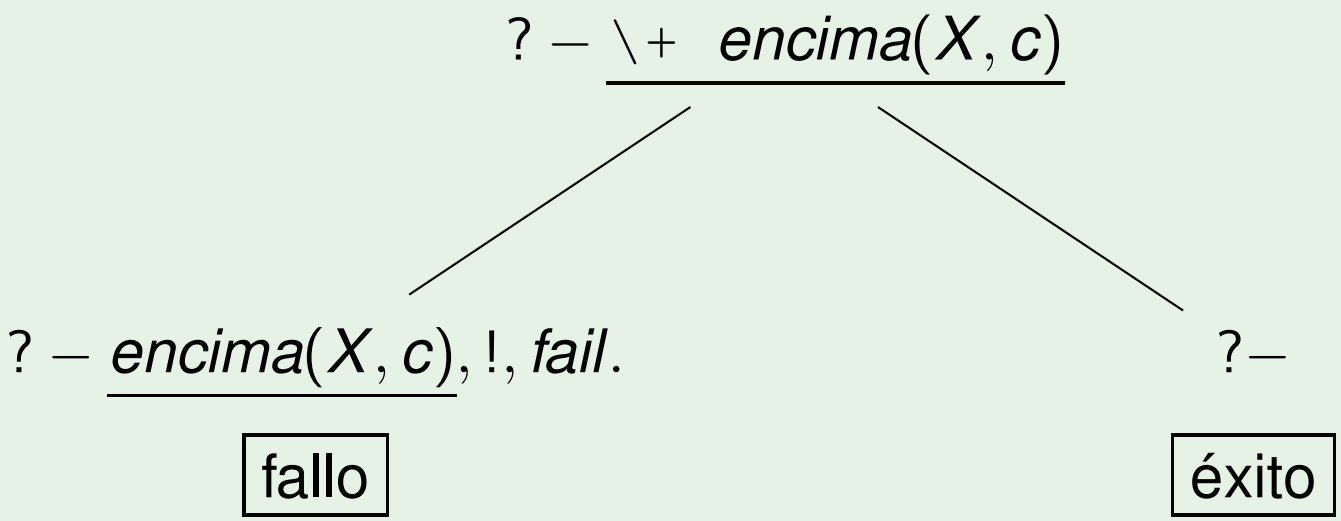
   $\Rightarrow$ Este tipo de objetivos se puede usar para realizar comprobaciones, pero *no para computar*.

2. La invocación de un objetivo del tipo "`\+ P`" con variables sin instanciar (sin valor) puede dar resultados erróneos: ¡PROLOG puede contestar negativamente a una consulta cuya respuesta debiera ser afirmativa!

   $\Rightarrow$ Es importante tener en cuenta este hecho a la hora de programar.

Ejemplo (El mundo de los bloques)

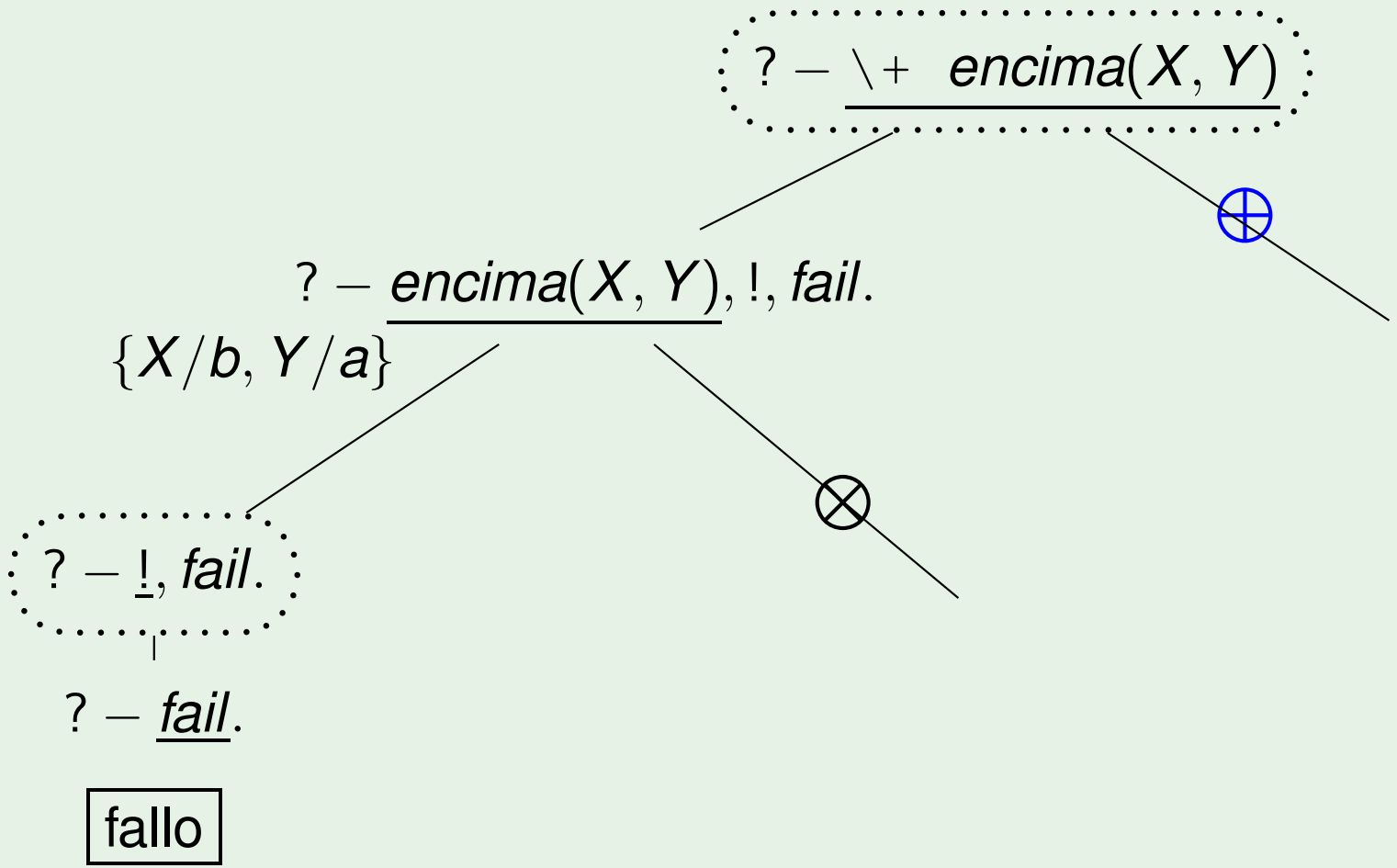
¿Existe algún bloque que no esté justo encima del bloque `c`?



PROLOG contesta `true` y su respuesta es correcta, pero no devuelve los valores de X que hacen cierta la consulta, es decir, ¡no computa!

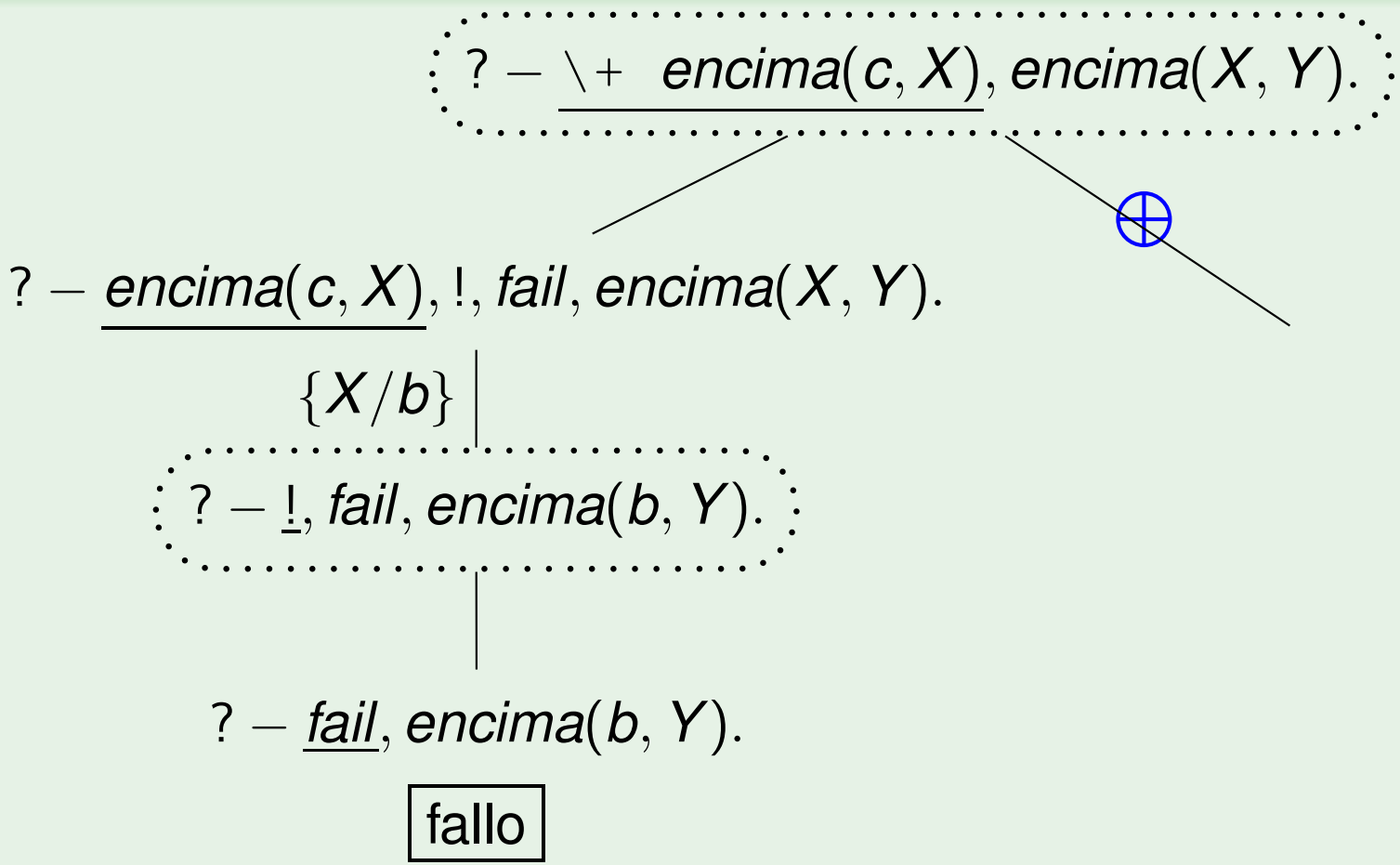
Ejemplo (El mundo de los bloques)

¿Existen dos bloques tales que el 1º no esté justo encima del 2º?



La respuesta obtenida, `false`, es ¡incorrecta!

Ejemplo (El mundo de los bloques)



La respuesta obtenida, `false`, es ¡incorrecta! (compare este árbol con el ▸ Árbol de Resolución hecho previamente para la misma consulta pero con los dos subobjetivos intercambiados).

Las respuestas incorrectas pueden producirse en cualquier nivel de un Árbol de Resolución en el que el predicado a evaluar (el de más a la izquierda) sea un predicado negado que en ese momento contenga alguna variable sin instanciar (sin valor concreto).

$\Longrightarrow$ Siempre que sea posible (no lo es por ejemplo si solo hay un predicado negado, como en la consulta `?- \+ encima(X,Y)` mencionada más arriba), habrá que colocar los predicados negados lo más a la derecha posible: conviene escribir

```
encima(X,Y), \+ encima(c,X).
```

en lugar de

```
\+ encima(c,X), encima(X,Y).
```

ya que en el ▸ primer caso, a diferencia del ▸ segundo, cuando PROLOG evalúa el predicado negado, la variable `X` ya está instanciada.

Ejercicios

Dados el programa

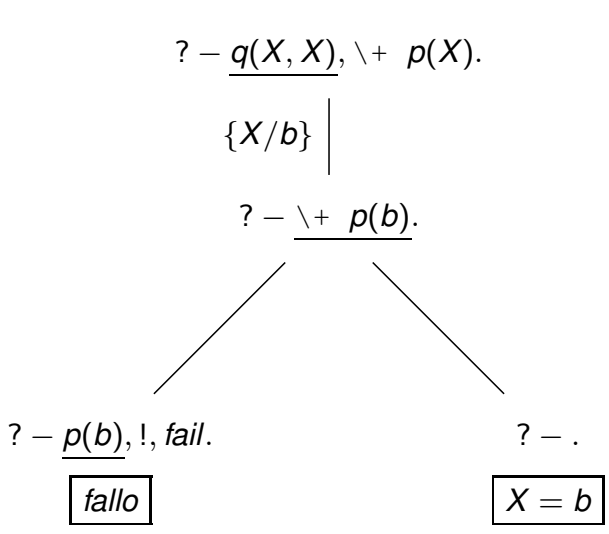
```
p(a).
q(b,b).
```

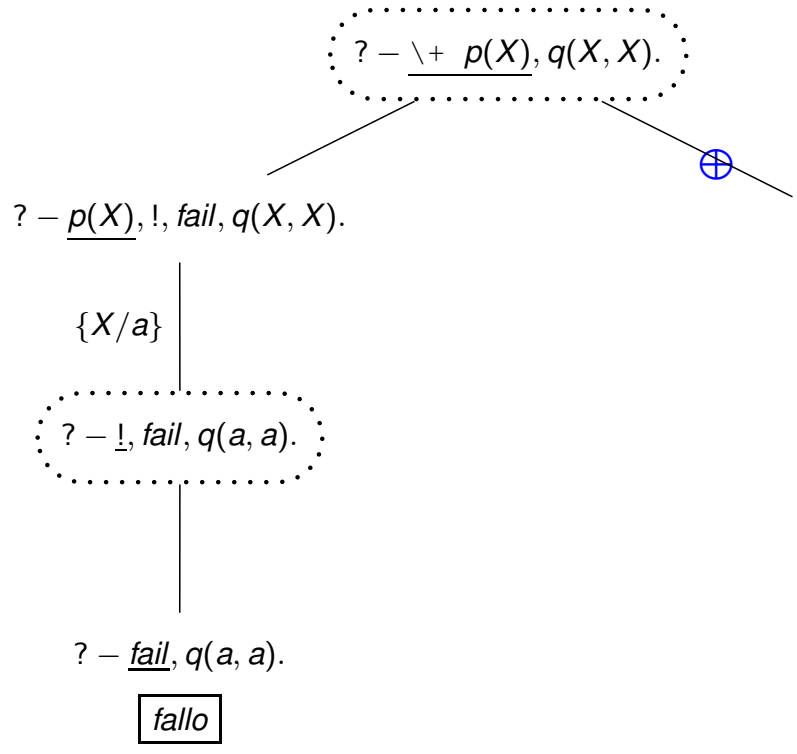
y las consultas (iguales, salvo en el orden de sus objetivos)

1. `?- q(X,X),\+ p(X).`
2. `?- \+ p(X), q(X,X).`

construya los Árboles de Resolución pertinentes para averiguar qué contestaría PROLOG *y explique las diferencias en las respuestas obtenidas.*

Soluciones propuestas:





PROLOG construye los árboles anteriores y contesta `X=b` y `false` (indicando que no hay más soluciones) en la primera consulta y `false` en la segunda. La diferencia en las dos respuestas anteriores se debe a que en el segundo caso se está resolviendo un predicado negado, `\+ p(X)`, con una variable que no está instanciada, y en estos casos PROLOG no siempre funciona correctamente.

Ejercicios

Dado el programa

```
q(a,b).
q(b,c).
q(a,c).
p(X,Z) :- q(X,Z).
p(X,Z) :- q(X,Y), p(Y,Z).
```

construya los Árboles de Resolución pertinentes para averiguar el comportamiento de PROLOG *ante las siguientes consultas:*

1. `?- \+ p(V,c), q(a,V).`
2. `?- q(a,V), \+ p(V,c).`

¿Qué ocurriría si en las consultas anteriores se añadiese un corte entre los dos subobjetivos?

Soluciones propuestas:

La respuesta de PROLOG ante la primera consulta es `false` (el Árbol de Resolución correspondiente tiene, de izquierda a derecha, una rama fallo y tres ramas podadas por un corte). La respuesta es errónea debido a que se evalúa el predicado negado `\+ p(V,c)` con la variable `V` sin instanciar.

La respuesta de PROLOG ante la segunda consulta es `V=c` y a continuación `false` indicando que no hay más soluciones (el Árbol de Resolución correspondiente tiene, de izquierda a derecha, una rama fallo, dos ramas podadas por un corte, dos ramas fallo y una rama éxito con solución `V=c`). La respuesta ahora es correcta porque ya no se evalúa `\+ p(V,c)` sino `\+ p(b,c)` (que resulta ser falso) y después `\+ p(c,c)` (que da cierto).

Si se añadiese un corte entre los dos subobjetivos:

- En la primera consulta no afectaría en nada, nunca se llegaría a evaluar ese corte puesto que el primer subobjetivo falla.
- En la segunda consulta el corte introducido podaría el hijo derecho de la raíz, que es el que encuentra la solución, por lo que la respuesta de PROLOG pasaría a ser `false`.

Ejercicios

Considere el programa PROLOG

```
p(A,B) :- q(A), !, r(B,A).   |   q(1).
r(A,B) :- q(A).              |   q(2).
t(2).                        |   q(3).
```

y la consulta

```
?- p(Y,X), \+ t(X).
```

1. *¿Qué se pretende averiguar con esa consulta?*
2. *Construya el Árbol de Resolución correspondiente.*
3. *Deduzca de lo anterior qué respuesta(s) ofrecería* PROLOG *ante la consulta dada, y en qué orden las facilitaría.*

Soluciones propuestas:

1. El objetivo de la consulta es averiguar si, dado el conocimiento expresado en el programa, existen objetos Y y X tales que el primero está relacionado con el segundo mediante la relación “p” de forma que el segundo, además, no cumple la propiedad “t”.
2. La construcción paso a paso del Árbol de Resolución solicitado está disponible aquí.
3. Ante una consulta, PROLOG construye el árbol de Resolución, en profundidad, y facilita las soluciones según las va encontrando. Por lo tanto, ante la consulta dada, y a la vista del árbol anterior, la respuesta de PROLOG será la siguiente:

Y=1, X=1 ;% primera solución encontrada
Y=1, X=3 ;% segunda solución encontrada
false % indicando que no quedan más soluciones

BIBLIOGRAFÍA

- L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.
- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fifth edition, 2003.
- I. Bratko.*Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 2001.
- J. Lloyd. *Foundations of Logic Programming*, (Second Edition). Springer-Verlag, 1987.
- R. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, MA, 1990.
- U. Nilsson and J. Maluszynski. Logic, Programming and Prolog. John Wiley & Sons Ltd, 1996.
- SWI-Prolog, entorno de programación en Prolog de dominio público.
- comp.lang.prolog. Faq