PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

8. El predicado de corte

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

DEFINICIÓN, EFECTOS Y PROPIEDADES

Definición

- El corte es un predicado predefinido que permite al programador intervenir en el mecanismo de búsqueda de soluciones de PROLOG.
- Se denota mediante un punto de exclamación, `!`, no tiene argumentos, y es siempre cierto.
- Se puede incluir, como un predicado más, en el *cuerpo* de las reglas o en las consultas, en cualquier posición.

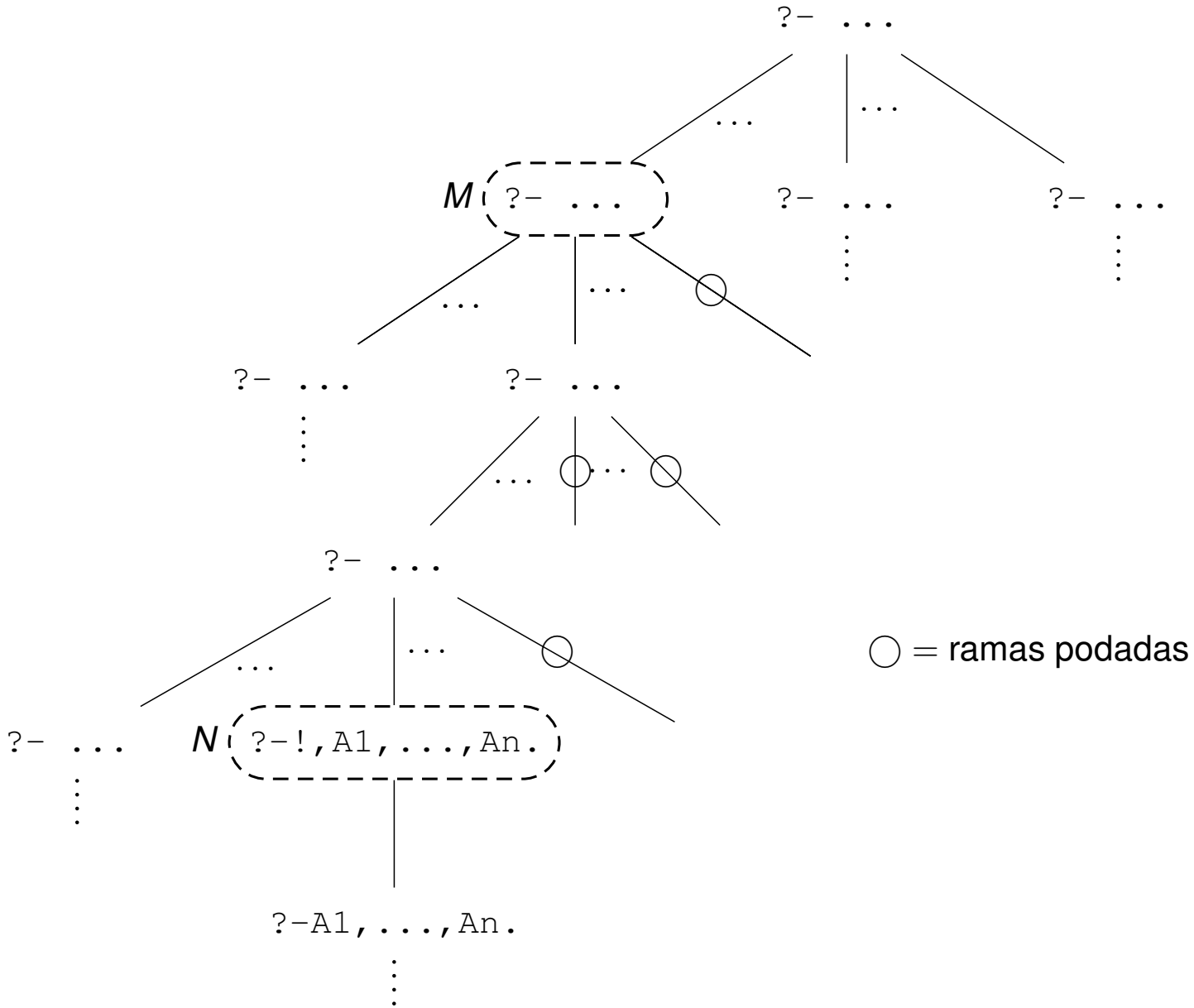
Ejemplos

- `B :- B1, B2, !, B3.` `B :- !.`
  `B :- B1, B2, !.` son reglas válidas.
- `?- A1, !, A2.` o `?- A1, !.` son consultas válidas.

Efectos de un corte en un Árbol de Resolución

- Los cortes podan (impiden el desarrollo) ciertas ramas de los Árboles de Resolución, las elegidas de acuerdo con lo que se explica a continuación.
- Si en un Árbol de Resolución (ver dibujo siguiente):
  - `N` es un nodo que *empieza* con un corte (`N = ?- !,A1,...,An.`)
  - `M` es su ascendiente más cercano que no contiene ese corte (o la raíz del árbol si todos sus ascendientes lo contienen)

  entonces el nodo `N` se expande normalmente (en particular, como el corte es cierto, `N` tiene siempre un único hijo, él mismo sin el corte), pero al retroceder en la construcción del Árbol el sistema poda todas las ramas del árbol que pasan por `M` y no por `N` y que están situadas más a la derecha que la rama que lleva de `M` a `N` (todas las marcadas con ◯ en el dibujo).
- PROLOG retrocede automáticamente al encontrar una rama podada (no reporta las podas).


?- ...
M
?- ...
?- ...
?- ...
?- ...
?- ...
?- ...
N
?-!,A1,...,An.
?-A1,...,An.
= ramas podadas

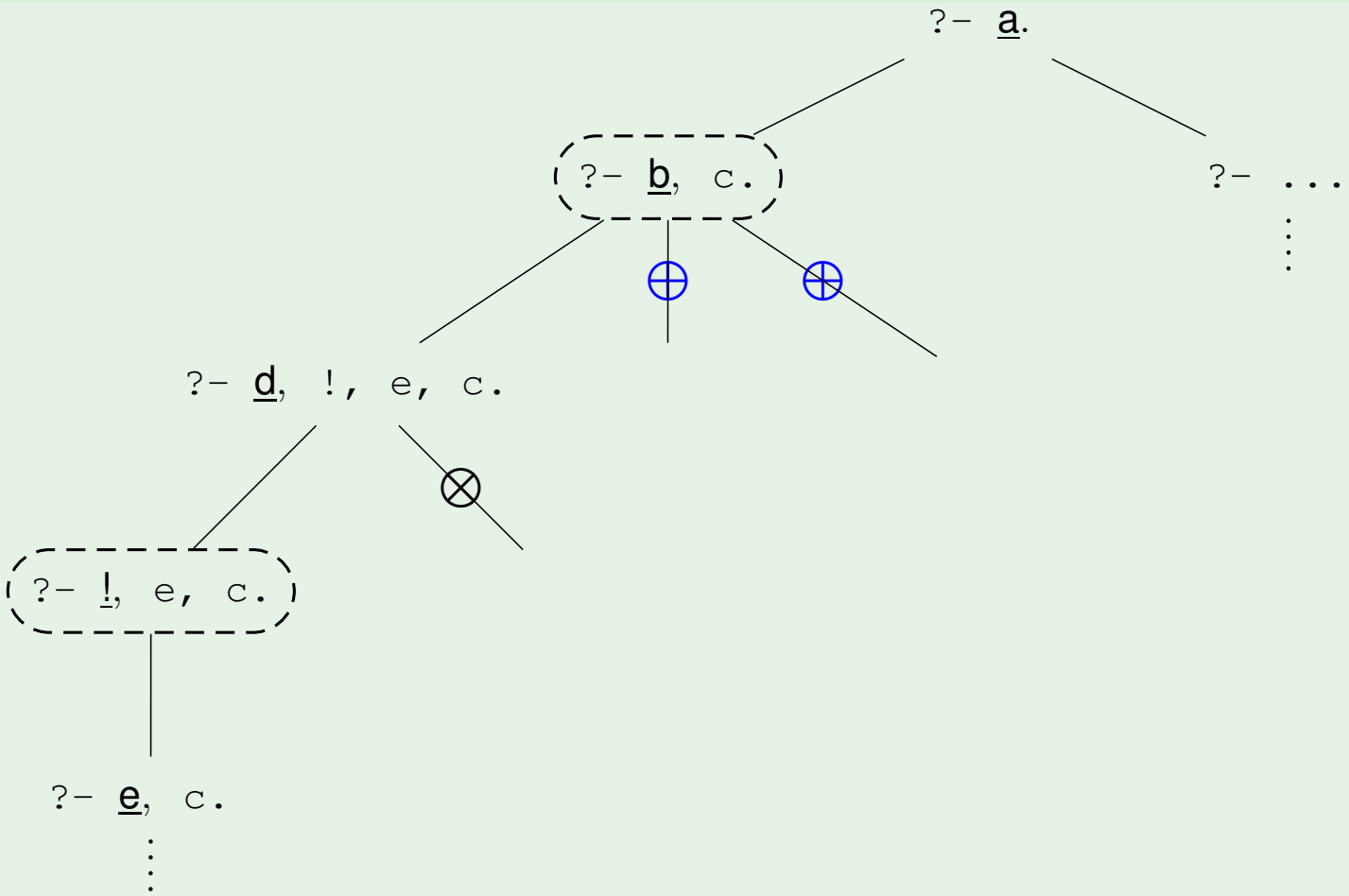
Ejemplo (Efectos del predicado de corte, 1/4)

Considere el programa PROLOG dado por las siguientes cláusulas:

```
a :- b, c.
a :- ....
b :- d, !, e.
b.
b :- ....
d.
d :- ....
e :- ....
```

La página siguiente muestra el Árbol de Resolución para la consulta `?- a`. Las ramas marcadas con $\otimes$ y $\oplus$ son las ramas podadas que se deben ignorar debido al corte de la regla `b :- d, !, e.`

Ejemplo (Efectos del predicado de corte, 2/4)



El corte de `b :- d, !, e.` poda tanto las alternativas para probar lo que hay a su izquierda (el predicado `d` en este caso, $\otimes$), como las alternativas para probar la cabeza de la regla (el predicado `b`, $\oplus$).

Ejemplo (Efectos del predicado de corte, 3/4)

Considere el programa PROLOG dado por las siguientes cláusulas:

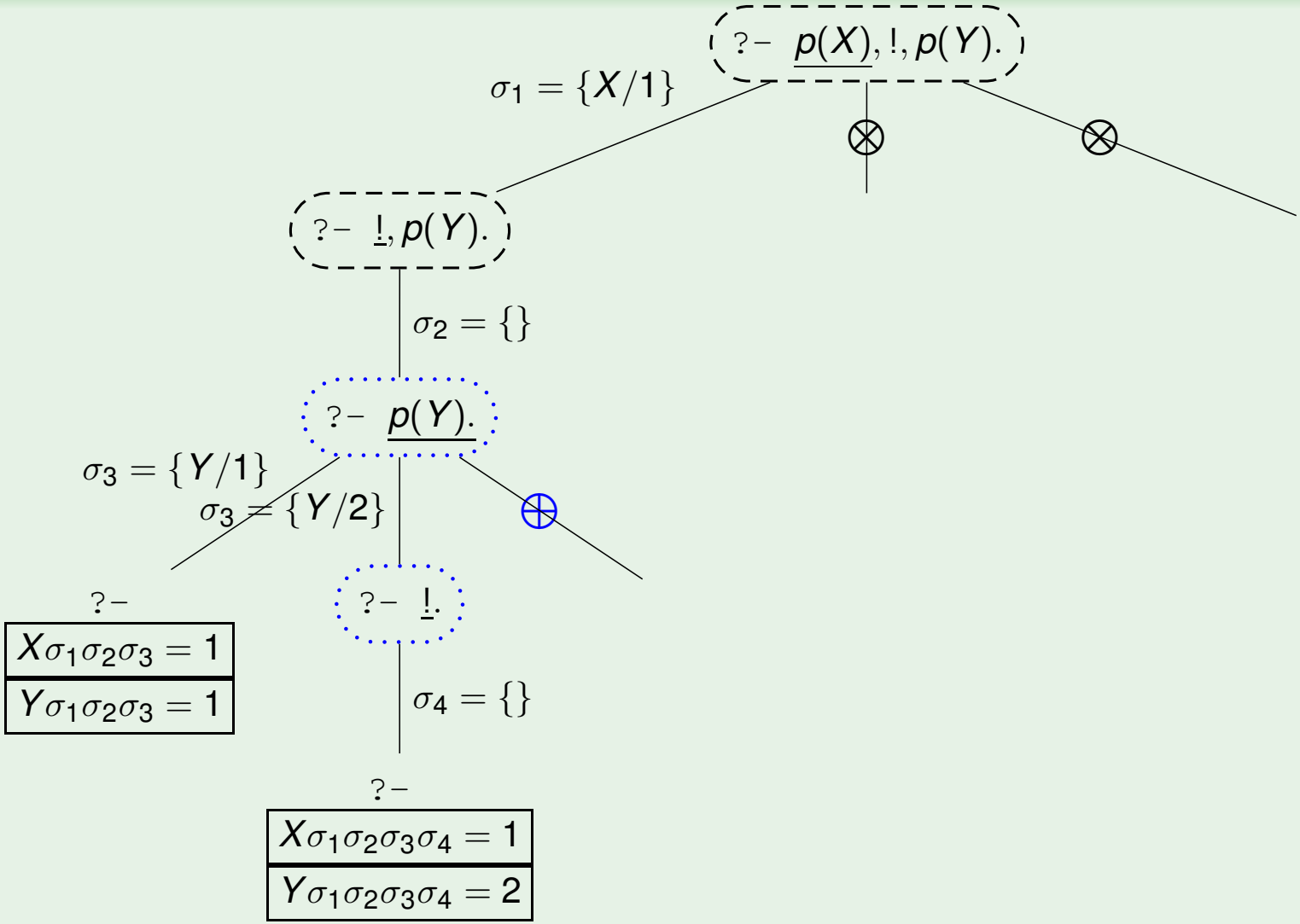
```
p(1).
p(2) :- !.
p(3).
```

La página siguiente muestra el Árbol de Resolución resultante al realizarse la consulta `?- p(X), !, p(Y).`

Este ejemplo ilustra lo siguiente:

1. Un Árbol de Resolución puede verse afectado por *varios* predicados de corte (marcados con distinto color en el dibujo).
2. El nodo `M` mencionado previamente en el caso general puede ser la propia raíz del árbol (lo será cuando la raíz contiene el corte).

Ejemplo (Efectos del predicado de corte, 4/4)



Soluciones: 1) `X=1, Y=1` 2) `X=1, Y=2` 3) `false` indicando que no hay más soluciones (las podas no se reportan).

1. La introducción de un corte en el cuerpo de una regla

```
B :- B1, ..., Bi, !, Bi+1, ..., Bm.
```

   - Impide las posibles reevaluaciones de los objetivos situados *a la izquierda* del corte: si se consigue probar B1, ..., Bi, no se vuelve a intentar su prueba al retroceder en la construcción del árbol (ver podas $\otimes$ en el penúltimo ejemplo). El corte, sin embargo, *no* impide la reevaluación de los objetivos situados a su derecha.
   - Impide las posibles reevaluaciones del predicado `B` mediante cualquier cláusula posterior con cabeza unificable con `B` (ver podas $\oplus$ en los dos últimos ejemplos).

2. La introducción de un corte en una consulta

```
?- A1, ..., Ai, !, Ai+1, ..., An.
```

   - Impide las posibles reevaluaciones de los objetivos situados *a la izquierda* del corte: si se consigue probar A1, ..., Ai, no se vuelve a intentar su prueba al retroceder en la construcción del árbol (ver podas $\otimes$ en el último ejemplo). El corte *no* impide la reevaluación de los objetivos situados a su derecha.

Ejercicios

1. *Dado el programa del último ejemplo, piense qué respuestas ofrecería* PROLOG*, y en qué orden, para cada una de las siguientes consultas, y compruebe sus respuestas construyendo los Árboles de Resolución correspondientes:*

```
?- p(X).                    |    ?- !, p(X), p(Y).
?- p(X), p(Y).              |    ?- p(X), p(Y), !.
```

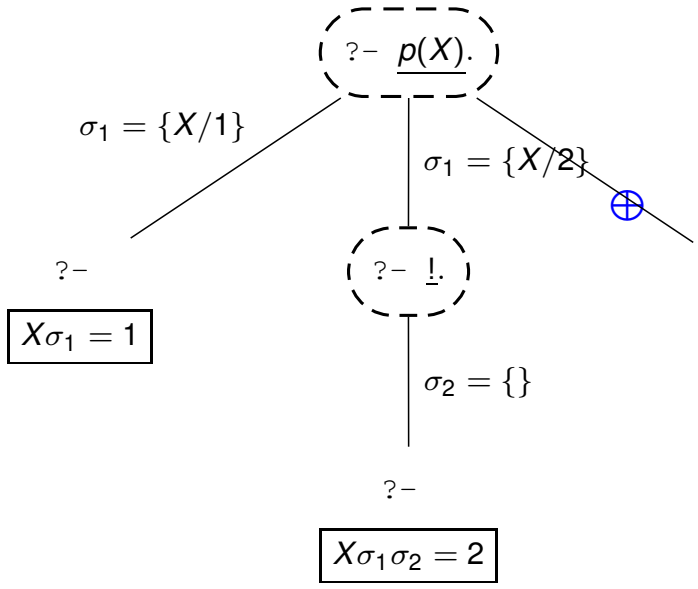
2. *Considere la consulta* `?- s(Z).` *y el programa*

```
s(Y) :- q(X,Y), !, r(Y).    |    q(a,a).
s(X) :- q(X,X).             |    q(a,b).
r(b).
```

   - *¿Qué respuesta(s) ofrecería* PROLOG *a la consulta planteada, y en qué orden? Construya el Árbol de Resolución correspondiente.*
   - *Misma pregunta que en el apartado anterior pero intercambiando las dos cláusulas del predicado* `q`*.*

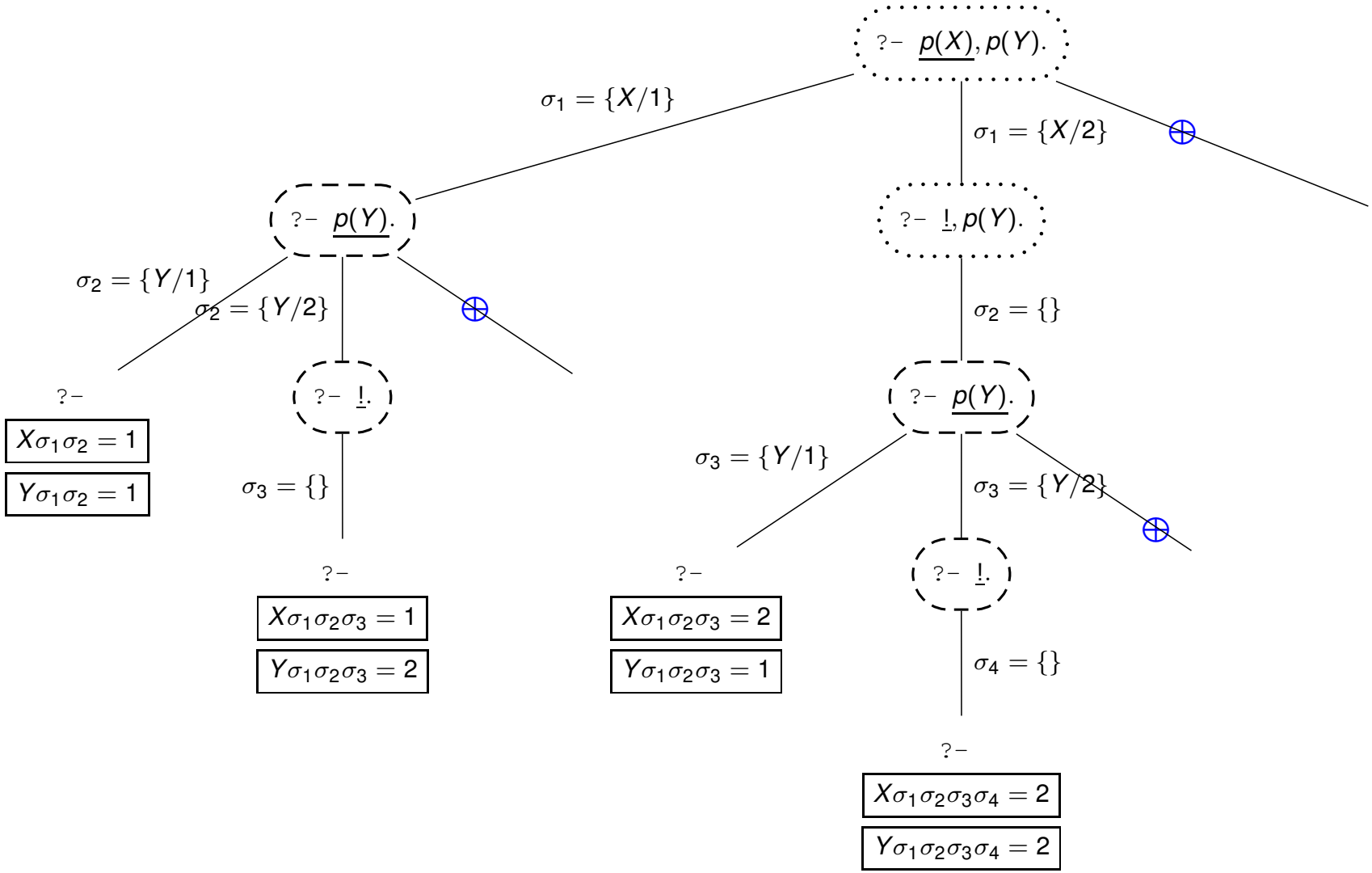
Soluciones propuestas:

1. Consulta `p(X)`. El corte de la regla `p(2) :- !.` impide la reevaluación de `p(X)` con cualquier cláusula posterior a ella, en este caso con el hecho `p(3)`. Por ello, PROLOG solo ofrecerá dos soluciones: `X=1` y `X=2`. En efecto, el Árbol de Resolución correspondiente es el siguiente:



Soluciones: 1) `X=1` 2) `X=2` 3) `false` indicando que no hay más soluciones (las podas no se reportan).

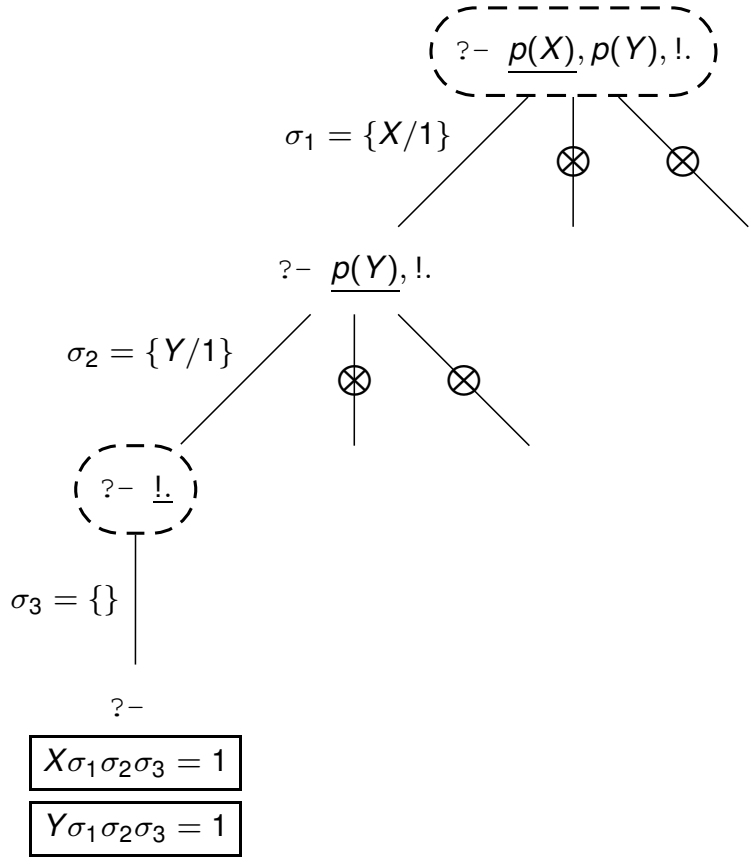
Consulta `?- p(X),p(Y).` El corte de la regla `p(2) :- !.` impide la reevaluación tanto de `p(X)` como de `p(Y)` con el hecho posterior `p(3)`. En efecto:



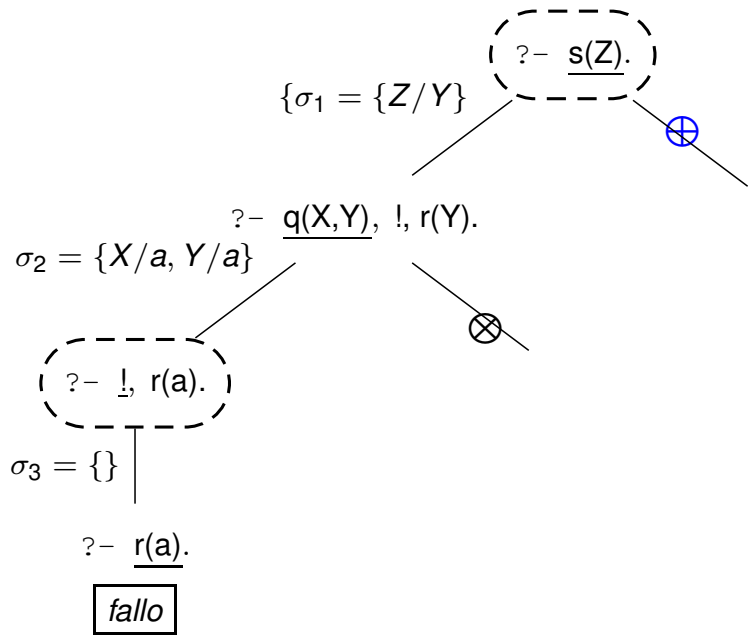
Soluciones: 1) `X=1, Y=1` 2) `X=1, Y=2` 3) `X=2, Y=1` 4) `X=2, Y=2` 5) `false` indicando que no hay más soluciones (las podas no se reportan).

Consulta `?- !, p(X), p(Y).` El Árbol de Resolución correspondiente a esta consulta -y por lo tanto las soluciones ofrecidas- es el mismo que el de la consulta anterior, `?- p(X), p(Y)`, con la única diferencia de que habría que añadirle la consulta `?- !, p(X), p(Y).` como raíz con un único hijo. Colocar un corte en la cabeza de una consulta no aporta nada (recuerde que los cortes en las consultas sirven para impedir la reevaluación de lo que haya a su *izquierda*).

Consulta `?- p(X), p(Y), !.` El corte de la consulta impide las posibles reevaluaciones de todos los objetivos situados a su izquierda, por lo que tanto `p(X)` como `p(Y)` se evaluarán solo con la primera cláusula, dando como única solución `X=1,Y=1`. En efecto, el Árbol de Resolución asociado es el siguiente:



2. El corte de la regla `s(Y) :- q(X,Y), !, r(Y)` impide todas las posibles reevaluaciones tanto de `q` (por estar a la izquierda del corte) como de `s`, provocando que la única respuesta de PROLOG ante la consulta `?- s(Z)` sea `false`: en efecto, la primera (y única debido al corte) forma de probar `q(X,Y)` es utilizando el hecho `q(a,a)` mediante u.m.g. $\{X/a, Y/a\}$, por lo que el siguiente objetivo es `r(a)`, que falla, y `s` no se reevalúa con `s(X) :- q(X,X)` otra vez debido al corte. Todo lo anterior queda reflejado en el Árbol de Resolución correspondiente:



Si se intercambian las dos cláusulas del predicado `q`, la rama fallo se convierte en rama éxito con solución `X=a,Y=b`, puesto que la `Y` se sustituiría por `b` en lugar de por `a`.

Ventajas del corte

- Permite mejorar la eficiencia de los programas, evitando la exploración de partes del Árbol de Resolución de las que se sabe de antemano que no conducirán a ninguna nueva solución.
- Permite aumentar la expresividad del lenguaje (por ejemplo, la negación por fallo finito que se verá más adelante).

Inconvenientes del corte

- Interviene en *cómo* se debe resolver el problema, entrando en contradicción con los principios de la programación lógica pura.
- Puede conducir a programas difíciles de leer y de validar, y puede provocar muchos errores de programación.

Conclusión

El corte es una herramienta de carácter extra-lógico, muy potente pero que debe usarse con mucho cuidado y en casos muy concretos.

Ejercicios

Vuelva al ejercicio nº 2 de la Práctica de PROLOG nº 3 y reflexione sobre las consultas que contienen cortes, comparándolas con las que no los contienen.

USO PARA SIMULACIÓN DE ESTRUCTURAS CONDICIONALES

Uno de los usos más habituales del corte es la simulación de estructuras condicionales.

si b_1 entonces c_1;
si no: si b_2, entonces c_2;
....
si no: si b_n, entonces c_n;
si no: c.

$\Longrightarrow$

```
a :- b1, !, c1.
a :- b2, !, c2.
....
a :- bn, !, cn.
a :- c.
```

- En la estructura condicional anterior, los cortes permiten indicar que las distintas reglas que definen el predicado `a` son incompatibles entre sí (puesto que se rigen por *"si no's"*).

- Si se llega a ejecutar el predicado de corte de una regla `a :- bi, !, ci`, es porque las condiciones `b1,..,b(i-1)` han fallado pero sin embargo se ha probado que `bi` es cierto, y la ejecución del corte tiene los dos siguientes efectos:

  1. Poda las posibles formas alternativas que haya para probar la condición `bi` (en general, para probar todo lo que hubiese delante del corte en el cuerpo de la regla), puesto que ya se ha demostrado que esa condición es cierta.

  2. Poda las posibles formas alternativas posteriores que haya para probar `a`, puesto que esas alternativas solo son válidas cuando `bi` resulta no ser cierto.

Ejemplo (Cálculo de una función definida a trozos, 1/3)

Suponga que necesita definir un predicado en PROLOG que permita calcular la siguiente función *fun*:

$$fun(x) = \begin{cases} 0, & \text{si } x \leq 10 \\ 1, & \text{si } 10 < x \leq 20 \\ 2, & \text{si } x > 20 \end{cases}$$

VERSIÓN 1: sin usar el predicado de corte

```
f(X,0) :- X =< 10.
f(X,1) :- X > 10, X =< 20.
f(X,2) :- X > 20.
```

La incompatibilidad entre las tres reglas se consigue con comprobaciones explícitas (`X>10` en la segunda regla, `X>20` en la tercera).

Ejemplo (Cálculo de una función definida a trozos, 2/3)

VERSIÓN 2: usando el predicado de corte

```
f(X,0) :- X =< 10, !.
f(X,1) :- X =< 20, !.
f(_,2).
```

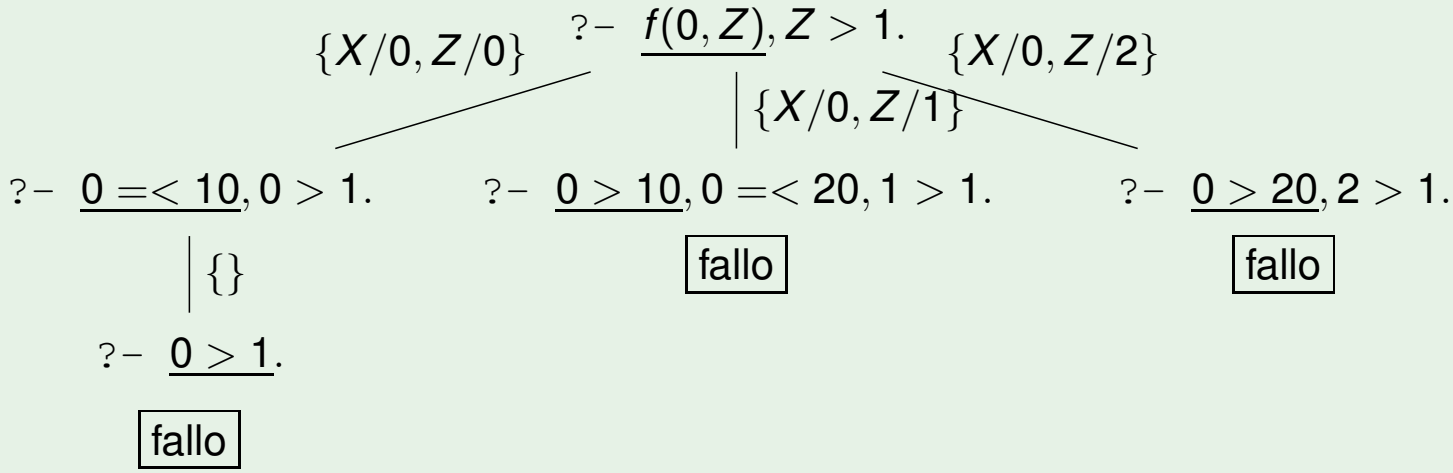
Esta versión es más eficiente: los cortes garantizan la incompatibilidad entre las reglas, evitando así las comprobaciones de la versión anterior (compare los Árboles de Resolución a continuación).

Note sin embargo que la versión 2 no siempre funciona correctamente, resulta por ejemplo `?- f(0,2).` ¡¡true!!. Solución correcta:

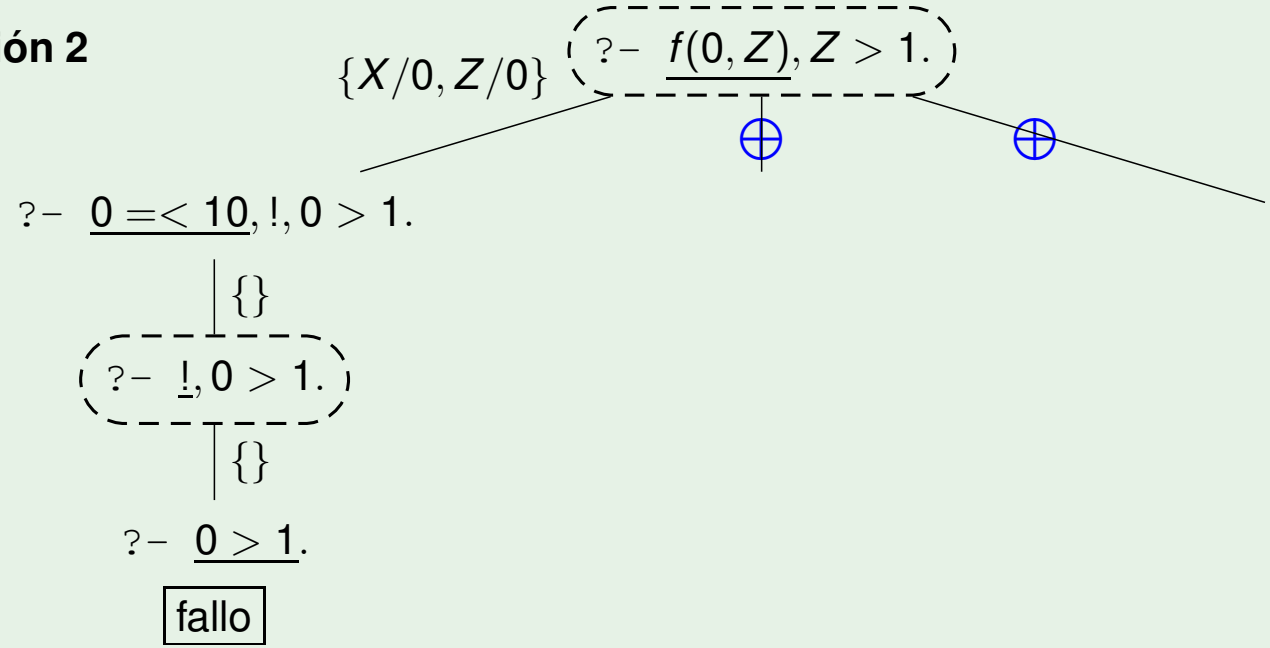
```
f(X,Y) :- X =< 10, !, Y=0.
f(X,Y) :- X =< 20, !, Y=1.
f(_,2).
```

Ejemplo (Cálculo de una función definida a trozos, 3/3)

Versión 1



Versión 2



Ejemplo (Cálculo del máximo de dos números)

VERSIÓN 1: sin usar el predicado de corte

```
maximo(X,Y,X) :- X >= Y.
maximo(X,Y,Y) :- X < Y. % comprobación necesaria
```

VERSIÓN 2: usando el predicado de corte

```
maximo(X,Y,X) :- X >= Y, !.
maximo(_,Y,Y).  % comprobación innecesaria
```

La segunda versión es más eficiente pero incorrecta en ciertos casos (por ejemplo `?-maximo(3,0,0).` ¡¡true!!). Solución correcta:

```
maximo(X,Y,Z) :- X >= Y, !, Z=X.
maximo(_,Y,Y).
```

Ejemplo (Pertenencia de un elemento a una lista, 1/2)

VERSIÓN 1: con las dos reglas compatibles

```
pertenece(C, [C|_]).
pertenece(C, [_|R]) :- pertenece(C,R).
```

Como se ha visto previamente, esta versión (`member` en SWI-Prolog) no solo sirve para averiguar si un elemento pertenece a una lista sino también para recorrer una lista (mediante backtracking):

```
?- pertenece(X,[a,b]).   X=a  ; X=b
```

VERSIÓN 2: con las dos reglas incompatibles, sin corte

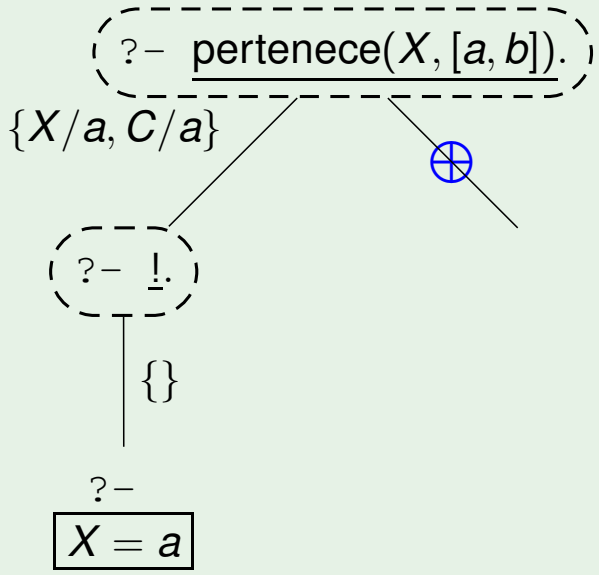
```
pertenece(C, [C|_]).
pertenece(E, [C|R]) :-
    C \= E,
    pertenece(E,R).
```

Ejemplo (Pertenencia de un elemento a una lista, 2/2)

VERSIÓN 3: con las dos reglas incompatibles, con corte

```
pertenece(C, [C|_]) :- !.
pertenece(C, [_|R]) :- pertenece(C,R).
```

Esta última versión (`memberchk` en SWI-Prolog) es más eficiente que la anterior pero ambas son menos potentes que la primera (ya no sirven para recorrer una lista: `?- pertenece(X,[a,b])` solo da `X=a`):



Ejemplo (Eliminación de la primera ocurrencia de un elemento en una lista, 1/2)

VERSIÓN 1: con las dos últimas reglas compatibles

```
borrarelemento([],_,[]).                % C1
borrarelemento([C|R],C,R).              % C2
borrarelemento([C|R],E,[C|NR]) :-       % C3
    borrarelemento(R,E,NR).
```

Esta implementación es incorrecta (construya el Árbol de Resolución):

```
?- borrarelemento([a,b,a], a, L).
L = [b, a] ;      solución correcta
L = [a, b] ;      ¡Solución incorrecta!
L = [a, b, a].    ¡Solución incorrecta!
```

Para solucionar el problema, es necesario indicar que las cláusulas C2 y C3 son incompatibles: C3 *solo* se debe usar cuando la cabeza de la lista, `C`, no coincida con el elemento a borrar, `E`.

Ejemplo (Eliminación de la primera ocurrencia, 2/2)

VERSIÓN 2: con las dos últimas reglas incompatibles, pero sin usar el corte

```
borrarelemento([],_,[]).
borrarelemento([C|R],C,R).
borrarelemento([C|R],E,[C|NR]) :-
    C \= E, % incompatible con la anterior
    borrarelemento(R,E,NR).
```

VERSIÓN 3: con las dos últimas reglas incompatibles, pero usando el corte

```
borrarelemento([],_,[]).
borrarelemento([C|R],C,NL) :-
    !, % incompatible con la regla siguiente
    NL = R.
borrarelemento([C|R],E,[C|NR]) :-
    borrarelemento(R,E,NR).
```

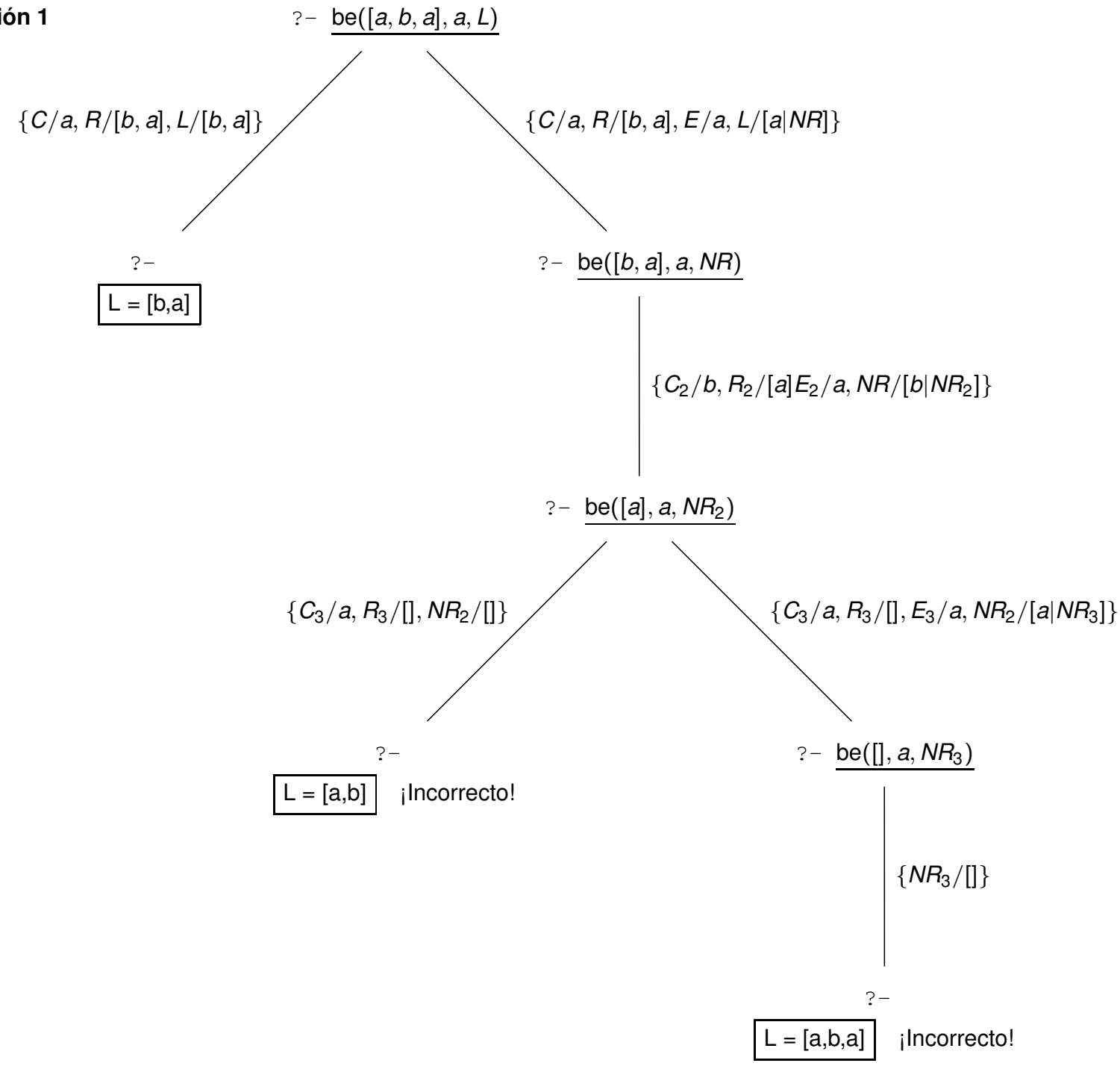
Observación

La sustitución de la segunda regla de la Versión 3 por la más sencilla `borrarelemento([C|R],C,R) :-!.` haría que el predicado no funcionase correctamente en ciertos casos, como por ejemplo con la consulta `?- borrarelemento([1],1,[1])`, que daría `true` en lugar de `false`.

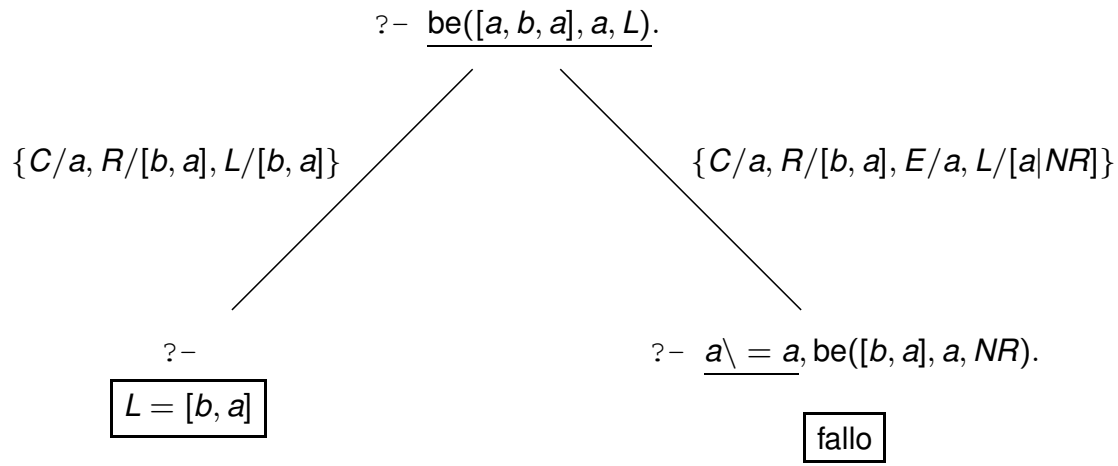
Ejercicios

1. *Construya y compare los Árboles de Resolución para la consulta* `?- borrarelemento([a,b,a], a, L)` *y las tres versiones del predicado* `borrarelemento` *discutidas más arriba.*
2. *Construya y compare los Árboles de Resolución para la consulta* `?- borrarelemento([1], 1, [1])` *con la versión 3 de* `borrarelemento` *y la versión dada en la observación posterior.*

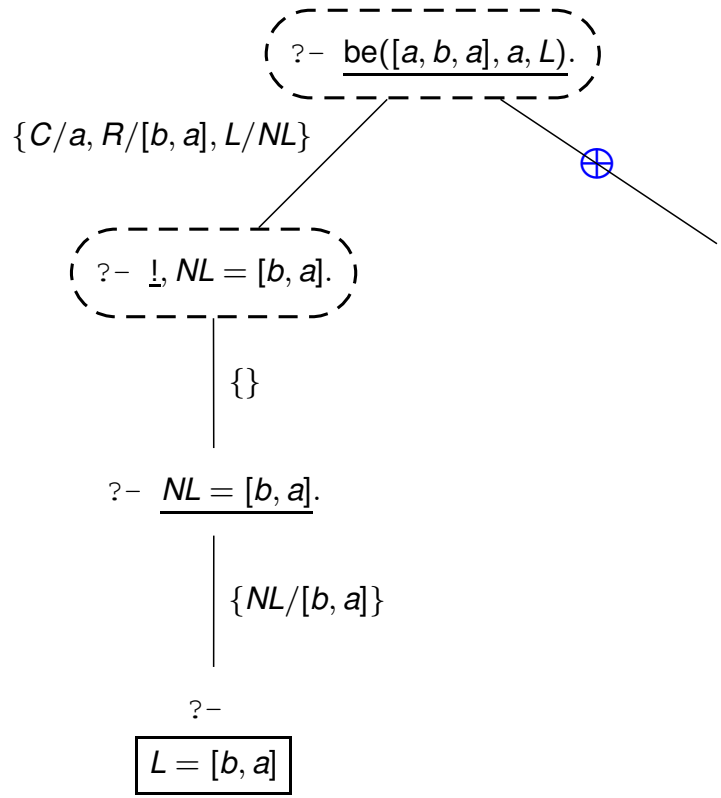
1. Versión 1


?- be([a, b, a], a, L)
{C/a, R/[b, a], L/[b, a]}
{C/a, R/[b, a], E/a, L/[a|NR]}
?-
L = [b,a]
?- be([b, a], a, NR)
{C2/b, R2/[a]E2/a, NR/[b|NR2]}
?- be([a], a, NR2)
{C3/a, R3/[], NR2/[]}
{C3/a, R3/[], E3/a, NR2/[a|NR3]}
?-
L = [a,b]
¡Incorrecto!
?- be([], a, NR3)
{NR3/[]}
?-
L = [a,b,a]
¡Incorrecto!

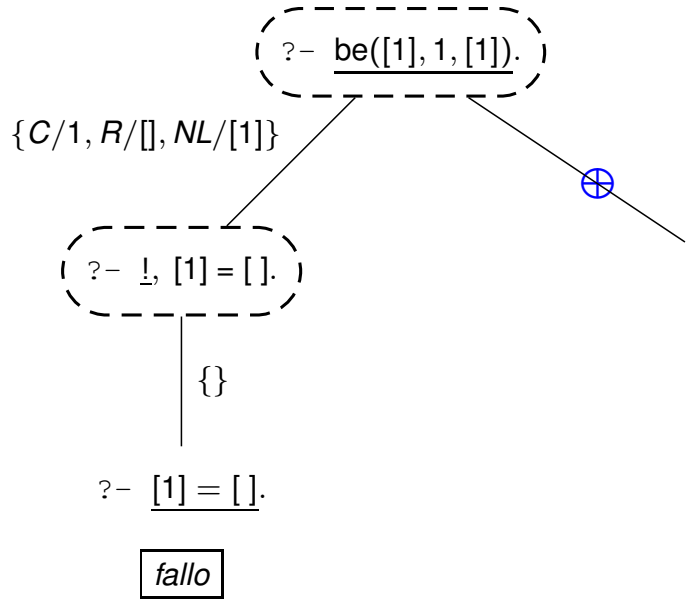
Versión 2



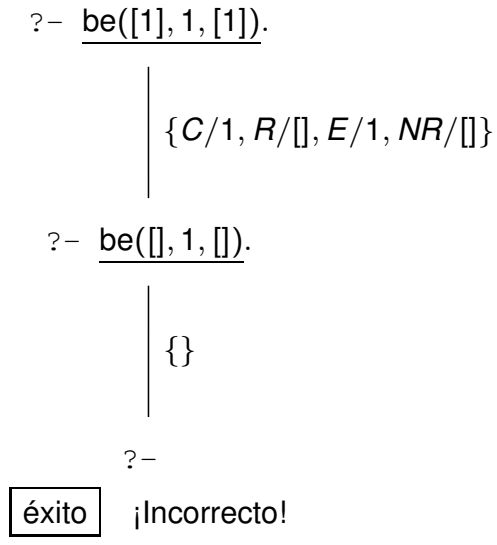
Versión 3



2. Versión 3



Versión de la "Observación"



Ejercicios

1. *Vuelva al ejercicio nº 3 de la Práctica de* PROLOG *nº 3 y reflexione sobre los posibles usos del predicado de corte en sus implementaciones.*
2. *Implemente el predicado* `reemplazar(+L,+E,+F,?NL)`*, cierto si NL es la lista resultante de reemplazar en la lista L todas las ocurrencias de E por F. Haga las consultas adecuadas para comprobar el funcionamiento de su código. Pruebe en particular la consulta* `reemplazar([a,c,a],a,b,L)` *y pida a* PROLOG *todas las posibles soluciones. En caso de que su implementación resulte ser incorrecta, explique por qué y haga las modificaciones necesarias para solucionar el problema. Construya el Árbol de Resolución correspondiente a alguna consulta.*

Soluciones propuestas:

1. Ver soluciones comentadas en Práctica de PROLOG nº 3 con soluciones.

2.

```
% reemplazar(+L,+E,+F,?NL)
% cierto si NL es la lista resultante de reemplazar
% en L todas las ocurrencias de E por F
```

reemplazar([],_,_,[]).

reemplazar([E|R],E,F,[F|NR]) :- !, reemplazar(R,E,F,NR).

reemplazar([C|R],E,F,[C|NR]) :- reemplazar(R,E,F,NR).

Si de la implementación anterior se suprimiese el predicado de corte, la implementación resultante ofrecería una primera solución correcta pero el resto de soluciones serían incorrectas. El motivo es que el corte es necesario para indicar a PROLOG que la tercera cláusula sólo debe utilizarse en caso de que no se pueda utilizar la segunda. Una versión alternativa sin usar el corte, correcta pero algo menos eficiente, consistiría en añadir la comprobación de que *C* no es unificable con *E* al comienzo del cuerpo de la tercera regla. El último ejemplo, en el que se discute la implementación del predicado `borrarelemento` presenta un problema similar a este.

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