

PROGRAMACIÓN DECLARATIVA

PROGRAMACIÓN LÓGICA

Tema PL2: El lenguaje PROLOG, aspectos básicos

8. El predicado de corte

Grado en Ingeniería Informática

URJC

Ana Pradera

Contenido

1 DEFINICIÓN, EFECTOS Y PROPIEDADES

- Definición
- Efectos
- Propiedades

2 USO PARA SIMULACIÓN DE ESTRUCTURAS CONDICIONALES

- Cálculo de una función definida a trozos
- Cálculo del máximo de dos números
- Pertenencia de un elemento a una lista
- Eliminación de la primera ocurrencia de un elemento en una lista
- Ejercicios

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

- $$B :- B1, B2, !, B3. \quad B :- !.$$

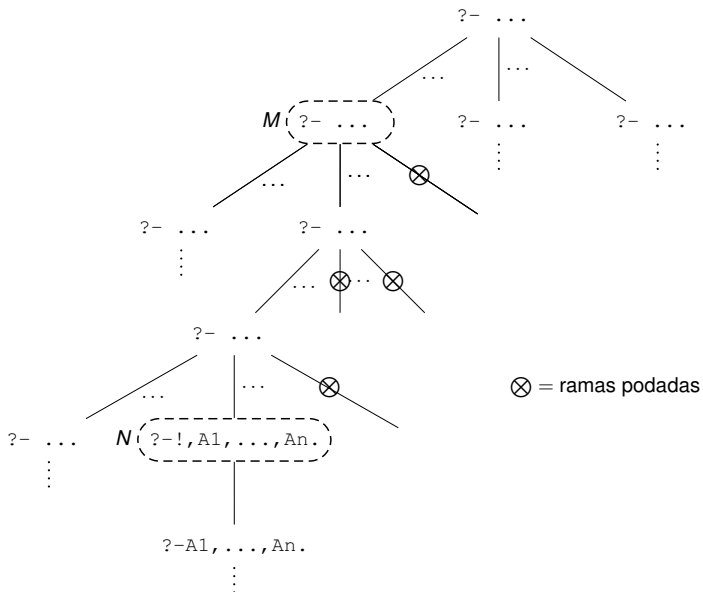
$$B :- B1, B2, !. \quad \text{son reglas válidas.}$$
- $$?- A1, !, A2. \quad \text{o} \quad ?- A1, !. \quad \text{son consultas válidas.}$$

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 ($N = ? - !, A_1, \dots, A_n.$)
 - 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 \otimes en el dibujo).

- PROLOG retrocede automáticamente al encontrar una rama podada (no reporta las podas).



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 :-

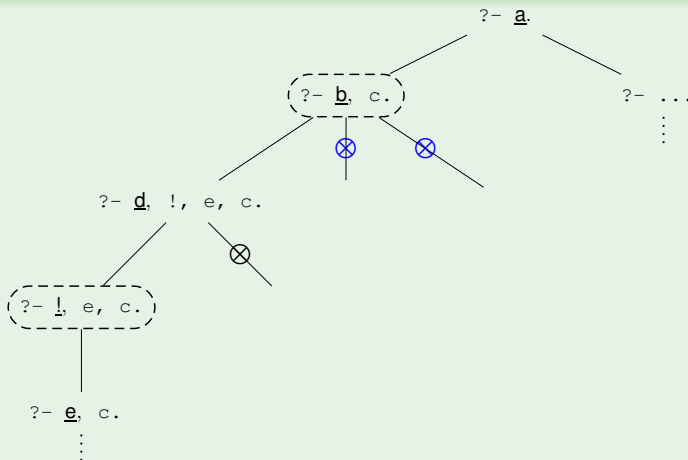
d.

d :-

e :-

La página siguiente muestra el Árbol de Resolución para la consulta
?- a. Las ramas marcadas con \otimes son las 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 , \otimes).

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

Considere el programa PROLOG dado por las siguientes cláusulas:

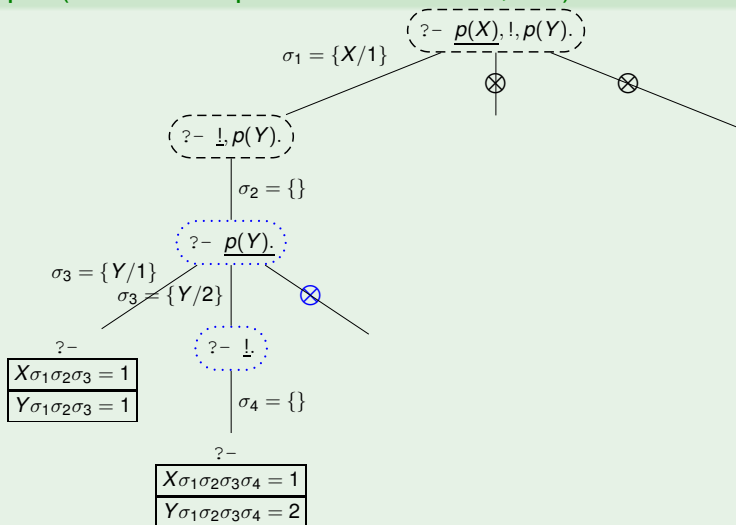
```
p(1) .  
p(2) :- ! .  
p(3) .
```

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

$B :- B_1, \dots, B_i, !, B_{i+1}, \dots, B_m.$

- Impide las posibles reevaluaciones de los objetivos situados *a la izquierda* del corte: si se consigue probar B_1, \dots, B_i , no se vuelve a intentar su prueba al retroceder en la construcción del árbol (ver podas \otimes en los dos últimos ejemplos). 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 \otimes en los dos últimos ejemplos).

2 La introducción de un corte en una consulta

$?- A_1, \dots, A_i, !, A_{i+1}, \dots, A_n.$

- Impide las posibles reevaluaciones de los objetivos situados *a la izquierda* del corte: si se consigue probar A_1, \dots, A_i , 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 .

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.	\Rightarrow	$a \text{ :- } b_1, \text{ !, } c_1.$ $a \text{ :- } b_2, \text{ !, } c_2.$ $a \text{ :- } b_n, \text{ !, } c_n.$ $a \text{ :- } c.$
--	---------------	---

- 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
 $a :- b_i, !, c_i$, es porque las condiciones $b_1, \dots, b_{(i-1)}$ han fallado pero sin embargo se ha probado que b_i es cierto, y la ejecución del corte tiene los dos siguientes efectos:
 - 1 **Poda las posibles formas alternativas que haya para probar la condición b_i** (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 b_i 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

$$f(X, 0) \text{ :- } X \leq 10, \text{ !}.$$

$$f(X, 1) \text{ :- } X \leq 20, \text{ !}.$$

$$f(_, 2).$$

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:

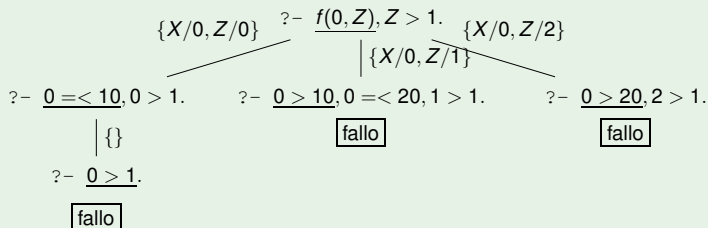
$$f(X, Y) \text{ :- } X \leq 10, \text{ !}, Y=0.$$

$$f(X, Y) \text{ :- } X \leq 20, \text{ !}, Y=1.$$

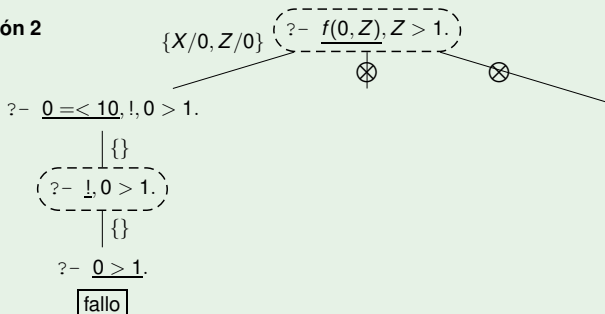
$$f(_, 2).$$

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) .
```

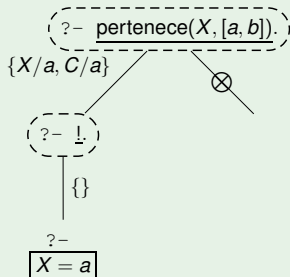
Ejemplo (Pertenencia de un elemento a una lista, 2/2)

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
    borrar(elemento(R,E, NR)).
```

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

```
?- borrar(elemento([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.*

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.

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**

© 2022 Ana Pradera Gómez

Algunos derechos reservados

Este documento se distribuye bajo la licencia

“Atribución-CompartirIgual 4.0 Internacional” de Creative Commons,
disponible en

<https://creativecommons.org/licenses/by-sa/4.0/deed.es>