Tema 3. CONTROL en la PROGRAMACION LOGICA

TEMA 3. CONTROL en la PROGRAMACION LOGICA

En este tema veremos aspectos del control en PROLOG. En primer lugar, cómo afecta el orden de los literales y de las sentencias en el funcionamiento del programa. Ya hemos visto cómo crear el árbol de llamadas y cómo explorarlo siguiendo una estrategia SLD. Sin embargo, muchas veces al programar y conociendo las características del problema, se puede asegurar que cierta parte del árbol no dará lugar a soluciones y por tanto no necesitaría ser explorada. Para indicar a PROLOG cuestiones de este estilo y ganar en eficiencia, disponemos del operador **corte** del que hablaremos en las dos siguientes secciones. Finalmente, dedicamos la última sección a estudiar la negación como fallo finito y su relación con la negación lógica.

3.1. Control mediante el Orden en literales y sentencias

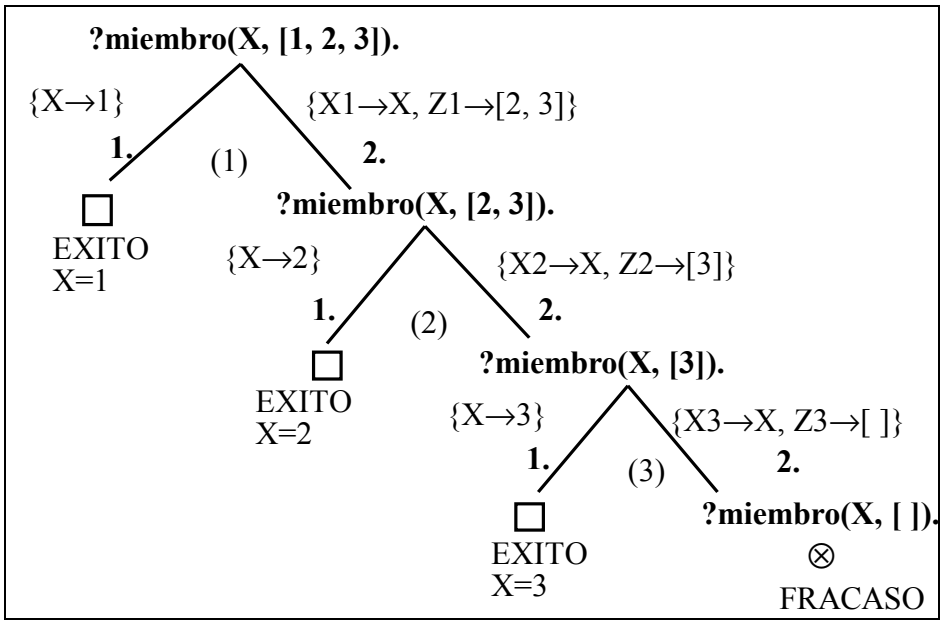
El orden en que aparecen los literales dentro de una sentencia (dentro del cuerpo) o el orden en que se introducen las sentencias en el programa son importantes en PROLOG. El orden afecta tanto al correcto funcionamiento del programa, como al recorrido del árbol de llamadas, determinando, entre otras cosas, el orden en que PROLOG devuelve las soluciones a una pregunta dada.

El ***orden de las sentencias*** determina el orden en que se obtienen las soluciones ya que varía el orden en que se recorren las ramas del árbol de búsqueda de soluciones.

Ejemplo: A continuación se presentan dos versiones del programa "miembro de una lista". Ambas versiones tienen las mismas sentencias pero escritas en distinto orden. A ambas versiones les hacemos la misma pregunta **?miembro (X, [1,2,3]).**

Primera versión:

1. **miembro (X, [X|_]).**
2. **miembro (X, [_|Z]) :- miembro (X, Z).**



? miembro (X, [1, 2, 3]).

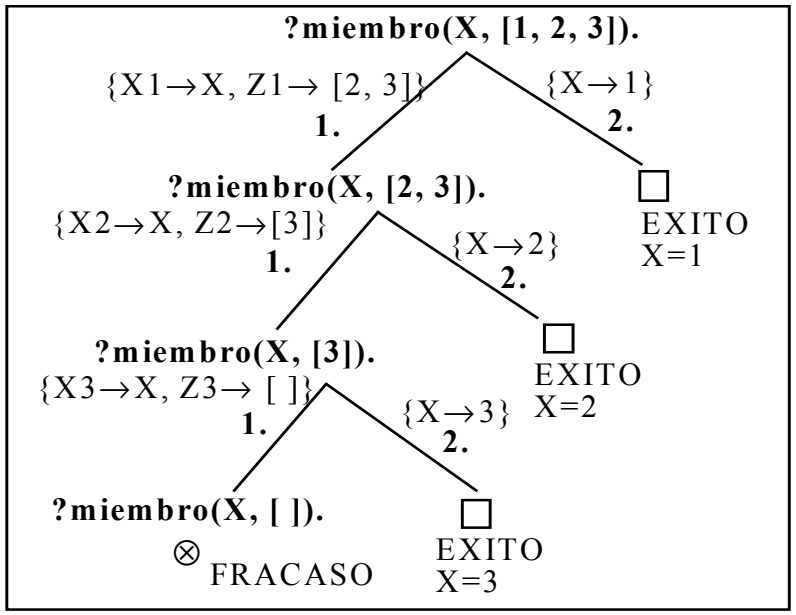
X = 1;

X = 2;

X = 3

Segunda versión:

1. **miembro (X, [_|Z]) :- miembro (X, Z).**
2. **miembro (X, [X|_]).**



? miembro (X, [1, 2, 3]).

X = 3;

X = 2;

X = 1

Si el árbol de búsqueda tiene alguna rama infinita, el orden de las sentencias puede alterar la obtención de las soluciones, e incluso llegar a la no obtención de ninguna solución. (Ver como ejemplo el generador de números naturales, de la sección **Generación infinita** en el tema anterior).

Como regla heurística a seguir cuando se programa en PROLOG, es recomendable que los hechos aparezcan antes que las reglas del mismo predicado.

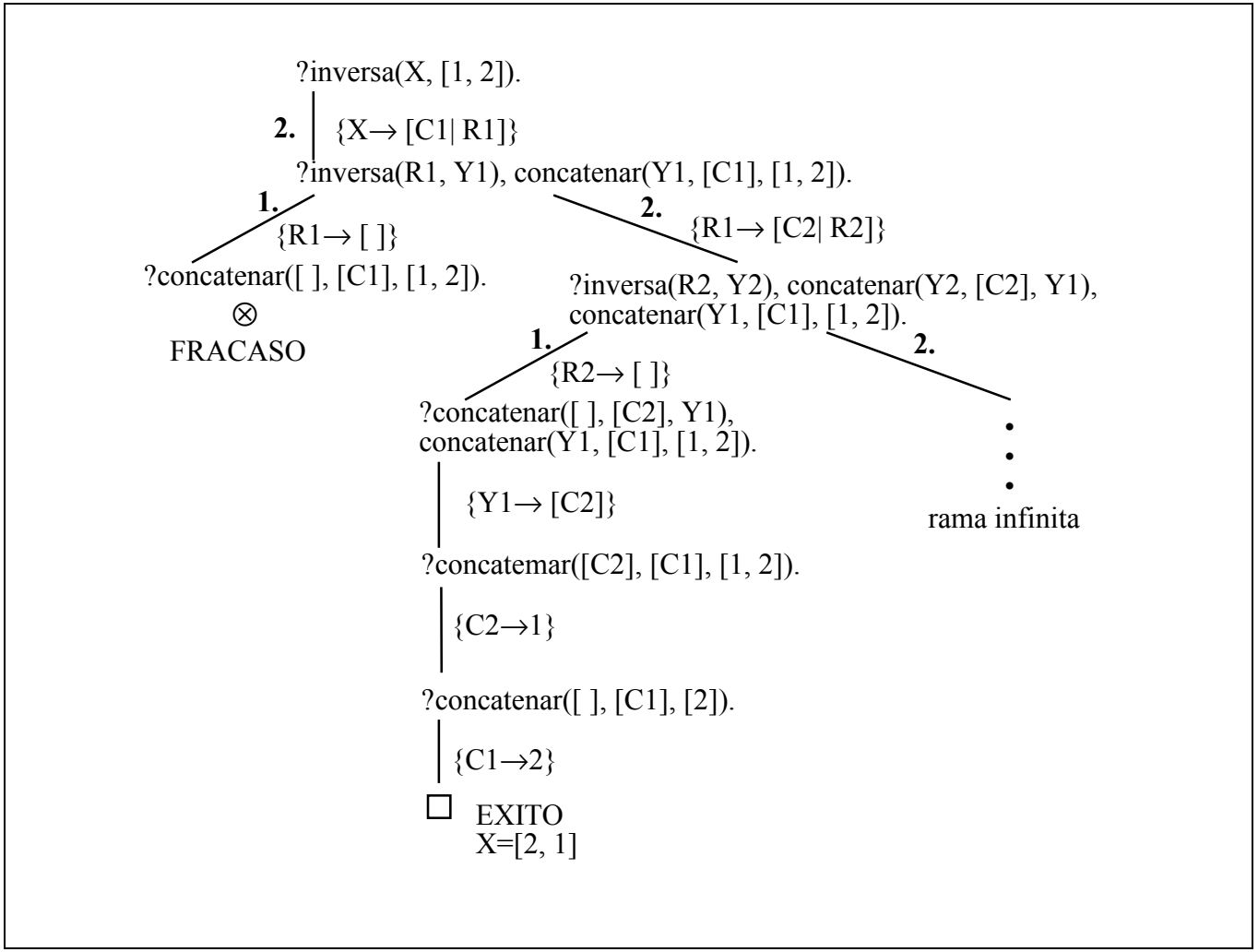
El ***orden de los literales*** dentro de una sentencia (en el cuerpo de una regla PROLOG) afecta más profundamente al espacio de búsqueda y a la complejidad de los cómputos lógicos:

- Distintas opciones en el orden de los literales pueden ser preferibles para distintos modos de uso.
  Ejemplo: La definición del predicado **hijo** puede hacerse:
  hijo(X, Y) :- hombre(X), padre(Y, X).
  para modo **(in, out)**: Se comprueba primero que el dato X es hombre y se busca a su padre Y.
  hijo(X, Y) :- padre(Y, X), hombre(X).
  para modo **(out, in)**: Se buscan los hijos de Y y se seleccionan si son hombres.

- El orden de los literales en el cuerpo de una regla influye también en la terminación. (Un ejemplo se vio en la sección **Recursión a izquierdas** del tema anterior).
  Ejemplo: El siguiente programa de "inversa de una lista" terminará para preguntas en modo **(in, out)**. Sin embargo, en modo **(out, in)** el árbol de búsqueda tendrá una rama infinita, por lo que tras dar la respuesta correcta se quedaría ciclando.

  1. **inversa([ ], [ ]).**
  2. **inversa([C|R], Z) :- inversa(R, Y), concatenar(Y, [C], Z).**

Hacemos la pregunta **? inversa(X, [1, 2]).**



3.2. Control con cortes

PROLOG tiene un predicado del sistema, llamado "corte", que afecta al comportamiento procedural de los programas. Su principal función es reducir el espacio de búsqueda de las computaciones PROLOG, "podando" el árbol de forma dinámica.
El corte puede usarse:

- para que el sistema no recorra caminos en el árbol que el programador sabe que no producen soluciones.
- para cortar caminos que dan soluciones y poder implementar una forma débil de negación.

Muchos de los usos de corte sólo pueden ser interpretados proceduralmente, en contraste con el estilo declarativo de la programación lógica "pura". Sin embargo, haciendo un buen uso de él, se mejora la **eficiencia** de los programas sin comprometer su **claridad**.

3.2.1. Operador "corte"

El corte es, desde el punto de vista sintáctico, un átomo formado por un símbolo de predicado denotado "**!**" y sin argumentos. Como **fin** se satisface siempre y no puede re-satisfacerse. Su uso permite podar ramas del árbol de búsqueda de soluciones. Como consecuencia, un programa que use el corte será generalmente **más rápido** y ocupará **menos espacio** en memoria (no tiene que recordar los puntos de "backtracking" para una posible reevaluación).
Ejemplo: El programa "**mezcla (L1, L2, L)**", en modo (in,in,out), mezcla dos listas ordenadas dadas **L1** e **L2** devolviendo la mezcla en la lista ordenada **L**

```
1. mezcla ([X| Xs], [Y| Ys], [X| Zs]) :- X<Y, mezcla (Xs, [Y| Ys], Zs).
2. mezcla ([X| Xs], [Y| Ys], [X,Y | Zs]) :- X=Y, mezcla (Xs, Ys, Zs).
3. mezcla ([X| Xs], [Y| Ys], [Y| Zs]) :- X>Y, mezcla ([X| Xs], Ys, Zs).
4. mezcla (Xs, [ ], Xs).
5. mezcla ([ ], Ys, Ys).
```

La mezcla de dos listas ordenadas es una operación determinista. Sólo una de las cinco sentencias "mezcla" se aplica para cada fin (no trivial) en una computación dada. En concreto, cuando comparamos dos números X e Y, sólo uno de los tres tests X<Y, X=Y ó X>Y será cierto.

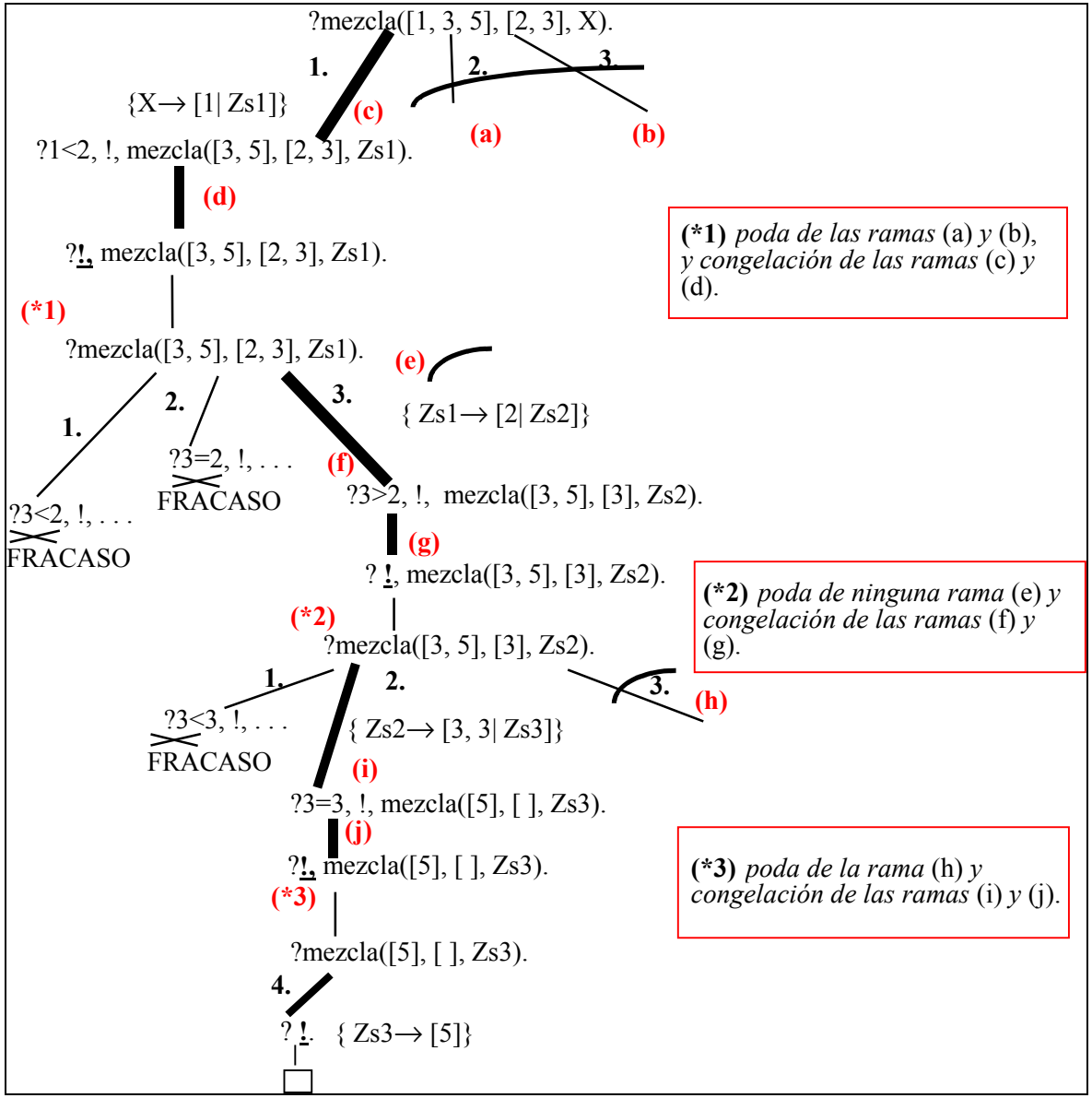
El corte puede usarse para expresar la naturaleza **mutuamente exclusiva de los tests** de las tres primeras sentencias. *Se colocará en las sentencias después de los tests*:

```
1. mezcla ([X | Xs], [Y | Ys], [X | Zs]) :- X<Y, !, mezcla (Xs, [Y | Ys], Zs).
2. mezcla ([X| Xs], [Y| Ys], [X, Y| Zs]) :- X=Y, !, mezcla (Xs, Ys, Zs).
3. mezcla ([X| Xs], [Y| Ys], [Y| Zs]) :- X>Y, !, mezcla ([X| Xs], Ys, Zs).
4. mezcla (Xs, [ ], Xs):- !.
5. mezcla ([ ], Ys, Ys).
```

Por otra parte, los dos casos básicos del programa (sentencias 4 y 5) son también deterministas: La sentencia correcta a usar se elige por unificación con la cabeza, por eso el corte aparece como el primer fin (en este caso el único) en el cuerpo de la regla 4. Dicho corte elimina la solución redundante (se volvería a obtener con la regla 5) al objetivo **?mezcla([ ], [ ], X).**

Siguiendo con el ejemplo de **mezcla**, en el recuadro siguiente se muestra el árbol de búsqueda de soluciones relativo a la pregunta **?mezcla([1,3,5], [2,3], X)** sobre el programa de **mezcla con cortes** añadidos. La explicación de las "podas" realizadas es la siguiente:

- El objetivo principal **?mezcla[1,3,5], [2,3], X)** se reduce primero (por la regla 1) al objetivo **? 1<2, !, mezcla([3,5], [2,3], X1).** El fin **1<2** se satisface, llegándose a un nodo del árbol cuyo primer fin es el corte.
- El efecto de ejecutar el corte (paso marcado con **(*1)**) es podar las ramas marcadas con **(a)** y **(b)**, y congelar las decisiones tomadas desde la sentencia que produjo el corte hasta la satisfacción del corte (ramas **c** y **d**).
- El corte de la regla 1 ha aparecido en el árbol porque el fin **mezcla[1, 3, 5], [2, 3], X)** se unificó con la cabeza de la regla 1. A dicho fin se le denomina ***fin padre*** del corte introducido.
- El resto del desarrollo del árbol es semejante, con dos cortes más podando ramas y congelando decisiones.


?mezcla([1, 3, 5], [2, 3], X).
1.
2.
3.
{X→ [1| Zs1]}
(c)
(a)
(b)
?1<2, !, mezcla([3, 5], [2, 3], Zs1).
(d)
?!, mezcla([3, 5], [2, 3], Zs1).
(*1)
?mezcla([3, 5], [2, 3], Zs1).
(e)
1.
2.
3.
{ Zs1→ [2| Zs2]}
?3=2, !, . . .
(f)
?3<2, !, . . .
FRACASO
FRACASO
?3>2, !, mezcla([3, 5], [3], Zs2).
(g)
? !, mezcla([3, 5], [3], Zs2).
(*2)
?mezcla([3, 5], [3], Zs2).
1.
2.
3.
(h)
?3<3, !, . . .
{ Zs2→ [3, 3| Zs3]}
FRACASO
(i)
?3=3, !, mezcla([5], [ ], Zs3).
(j)
?!, mezcla([5], [ ], Zs3).
(*3)
?mezcla([5], [ ], Zs3).
4.
? !. { Zs3→ [5]}
(*1) poda de las ramas (a) y (b), y congelación de las ramas (c) y (d).
(*2) poda de ninguna rama (e) y congelación de las ramas (f) y (g).
(*3) poda de la rama (h) y congelación de las ramas (i) y (j).

Operacionalmente, el efecto del corte es el siguiente: ! se satisface y *"congela"* todas las elecciones hechas desde que *"su fin padre"* se unificó con la cabeza de la sentencia que le contiene.

Esta definición dada sobre el efecto del corte es completa y precisa, no obstante, se pueden aclarar sus implicaciones:

- Primero, un corte **poda todas las sentencias debajo de él** (viéndolo en el programa). Un fin P unificado con una sentencia que contiene un corte que se ha satisfecho no podrá producir soluciones usando sentencias debajo de aquélla.
- Segundo, un corte **poda todas las soluciones alternativas de la conjunción de fines que aparezcan a su izquierda** en la sentencia. Esto es, una conjunción de fines seguida por un corte producirá como máximo una solución (se congelan las decisiones tomadas desde el fin que provocó el corte hasta que éste se satisface).
- Por otra parte, **el corte no afecta a los fines que estén a su derecha en la sentencia**. Estos pueden producir más de una solución, en caso de vuelta atrás (backtracking). Sin embargo, una vez que esta conjunción fracasa, la búsqueda continuará a partir de la última alternativa que había por encima de la elección de la sentencia que contiene el corte.

Resumiendo, de forma general, el ***efecto de un corte*** en una regla **C** de la forma **$A :- B_1, ..., B_k, !, B_{k+2}, ..., B_n$,** es el siguiente:

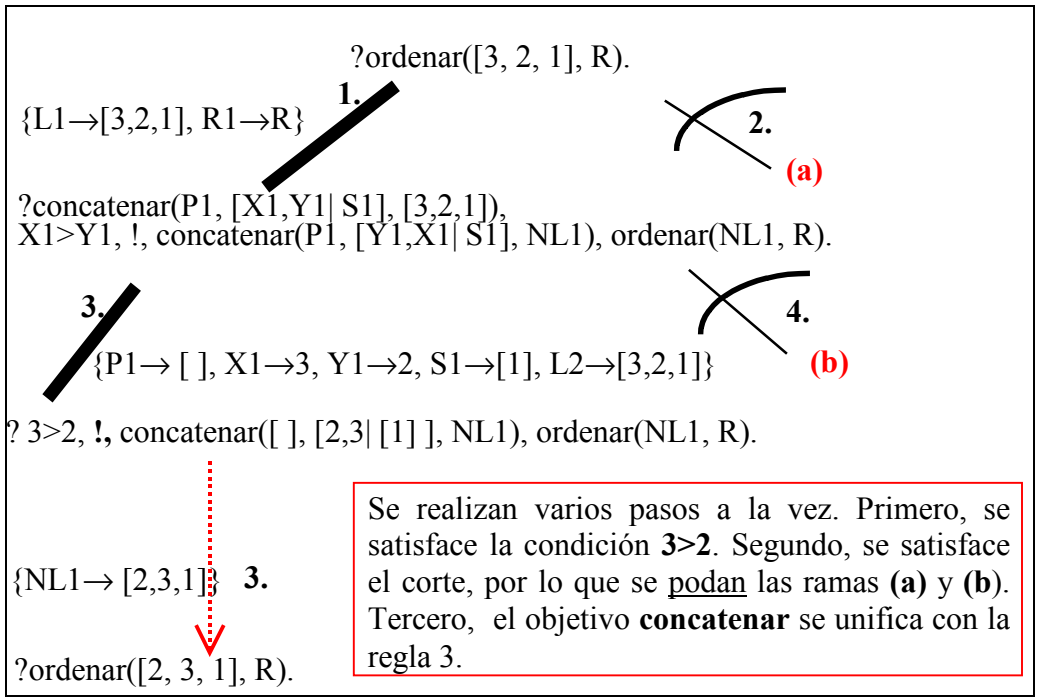
- Si el fin actual **G** se unifica con **A** y los fines **$B_1, ..., B_k$** se satisfacen, entonces el programa fija ya la elección de esta regla para reducir **G**; cualquier otra regla alternativa para A que pueda unificarse con G se ignora.
- Además, si los **Bi** con **i>k** fracasan, la vuelta atrás sólo puede hacerse hasta el corte. Las demás elecciones que quedaran para computar los **Bi** con **i≤ k** se han cortado del árbol de búsqueda.
- Si el backtracking llega de hecho al corte entonces éste fracasa y la búsqueda continúa desde la última elección hecha antes de que **G** eligiera la regla **C**.

3.2.2. Cortes verdes

En el ejemplo anterior, los cortes introducidos no han alterado el significado declarativo del programa. Se llaman "**cortes verdes**". Generalmente se usan para expresar **determinismo**: la parte del cuerpo que precede al corte (o a veces el patrón de la cabeza) comprueba un caso que excluye a todos los demás.

Ejemplo: El predicado **ordenar(L,R)** indica que **R** es el resultado de ordenar la lista **L** por medio de intercambios sucesivos. Este predicado usará **ordenada(L)** que nos dice si la lista L está ordenada.

1. **ordenar (L, R) :- concatenar(P, [X,Y | S], L), X>Y, !, concatenar(P, [Y,X| S], NL), ordenar(NL, R).**
2. **ordenar (L, L) :- ordenada(L).**
3. **concatenar ([ ], L, L).**
4. **concatenar ([X|Y], L, [X|Z]) :- concatenar(Y, L, Z).**



Un objetivo como **? ordenar([3, 2, 1], R)**, se resolverá intercambiando primero 3 con 2 pues es la primera solución que se calcula para el objetivo. Por tanto los primeros pasos de la ordenación son cómo se ven en el fragmento del árbol de búsqueda anterior. Al satisfacerse el corte, se evita la solución redundante que comenzaría intercambiando 2 con 1 y se ha obtenido la ordenación parcial de los dos primeros elementos de la lista. Se deja como ejercicio: continuar con el árbol desarrollando además las llamadas correspondientes al predicado concatenar, ¿qué sucede si el predicado concatenar tiene un corte en su caso trivial?

3.2.3. Cortes rojos

Un corte es **"rojo"** si ***afecta a la semántica declarativa*** del programa. En el ejemplo anterior de ordenación de una lista por intercambios sucesivos, el efecto del corte hace que un objetivo se resuelva con la sentencia 2 sólo cuando la lista ya está ordenada, pudiéndose por tanto suprimir el cuerpo de la sentencia 2 (**ordenada(L)**) sin afectar al comportamiento operacional del programa:

1. **ordenar (L, R) :- concatenar(P, [X,Y | S], L), X>Y, !, concatenar(P, [Y,X| S], NL), ordenar(NL, R).**
2. **ordenar (L, L).**

Así, este nuevo programa "funciona" igual que el anterior y además evita la comprobación última de ver que la lista está ordenada. Sin embargo, en esta propuesta queda afectada la semántica declarativa del programa: ¡la segunda sentencia (por sí sola) nos dice que toda lista está ordenada! Por tanto, el corte del programa es ahora **rojo**. Se debe destacar que el ***orden de las sentencias es ahora esencial***. Por otro lado, el cuerpo de la sentencia 2 en la versión anterior sí mantiene la corrección declarativa del programa.

Generalmente los cortes rojos aparecen al omitir del programa la comprobación explícita de una condición que se sabe se va a satisfacer en un cierto modo de uso (en nuestro caso el modo de uso sería **in** para el primer parámetro y **out** para el segundo).
Para otros usos del programa el corte rojo puede provocar incorrecciones no esperadas, por lo que deben utilizarse con cuidado. Veamos a continuación un ejemplo para ilustrar este problema.

Ejemplo:
Queremos definir "**minimo(X, Y, Z)**" para indicar que Z es el mínimo de X e Y. Pensamos en un uso **(in, in, out)**: dados X e Y , calcular el mínimo y devolverlo en Z. Para ello definimos:

Versión 1

1. **minimo (X, Y, X) :- X<= Y, !.**
2. **minimo (X, Y, Y).**

El corte es **rojo** pues afecta a la semántica declarativa del programa (en la segunda sentencia), pero para usos **(in, in, out)** la semántica operacional (o funcionamiento del programa) se corresponde con la del programa siguiente (donde el corte es **verde** al hacerse explícita la condición de la regla 2):

Versión 2

1. **minimo (X, Y, X) :- X<= Y, !.**
2. **minimo (X, Y, Y) :- Y<X.**

Por ejemplo, con ambos programas ante la pregunta **?minimo(7, 3, Z)** la contestación es **Z=3**

Sin embargo, *para el modo de uso **(in, in, in)** el programa de la versión 1 es incorrecto*:

? minimo(2, 5, 5)
si

Esto se debe a que **minimo(2, 5, 5)** **NO** se unifica con **minimo(X, Y, X)** por lo que no puede usarse la sentencia 1. Entonces el sistema intenta con la segunda y, al unificarse **minimo(2, 5, 5)** con **minimo(X, Y, Y)** se obtiene la refutación: respuesta **si.**

Con la versión 2 (declarativamente correcta) la misma pregunta obtiene la respuesta esperada:

? minimo(2, 5, 5)
no

3.3 Distintos usos del corte

Veremos a continuación algunos casos estándar en los que se utiliza el corte con un propósito determinado.

3.3.1. Confirmación de la elección de una regla

En el ejemplo visto en el apartado anterior (mezcla de dos listas ordenadas) el uso del corte tras una comprobación confirma la elección adecuada de dicha regla para el objetivo dado (permitiendo descartar las demás). Es decir, este uso se da cuando existe determinismo. Aquí vamos a ver otros dos ejemplos de este uso del corte.

Ejemplo 1.- Borrar todas las apariciones de un cierto elemento en una lista dada. Usaremos **borrar(X, L, LN)** para indicar "LN es la lista obtenida al borrar todas las apariciones de X en la lista L". Modo de uso: **(in, in, out)**, es decir, primer y segundo parámetros de entrada y tercer parámetro de salida.

```
borrar(X, [ ], [ ]) :- !.
borrar(X, [X|L], LN) :- !, borrar(X, L, LN).
borrar(X, [Z|L], [Z|LN]) :- borrar(X, L, LN).
```

Ejemplo 2.- Comprobar si una lista es sublista de otra. Usaremos **sublista(X, Y)** para indicar "X es sublista de Y". Modo de uso **(in, in)**.

```
sublista([ ], L) :- !.
sublista([X|Y], [X|M]) :- prefijo(Y, M), !.
sublista(L, [X|M]) :- sublista(L, M).

prefijo([ ], L).
prefijo([X|Y], [X|Z]) :- prefijo(Y, Z).
```

3.3.2. Combinación corte-fallo

En PROLOG existe un predicado predefinido **fail** que siempre fracasa. La combinación del predicado **corte** con **fail**, permite descartar opciones tras haberse comprobado una o varias condiciones. Puede ser útil al programar procedimientos de los que se conoce precisamente las condiciones por las que, si se cumplen, dicho procedimiento falla.

Ejemplo: El siguiente procedimiento pretende definir cuándo una persona es cura católico. Está claro que existen condiciones que a priori determinan que una persona no será cura católico: si es mujer, si esta casado, etc. Modo de uso: (**in**)

```
1. cura-catolico(X):- mujer(X), !, fail.
2. cura-catolico(X):- casado(X), !, fail.
3. cura-catolico(X):- edad(X, N), N<25, !, fail.
4. cura-catolico(X):- votos(X).
```

El orden de las sentencias es esencial para el buen funcionamiento del programa, ya que se pueden dar simultáneamente condiciones de varias reglas. Por ejemplo, una persona puede haber hecho los votos pero estar casada, entonces ¿será cura católico? Para nuestro programa la solución está clara y su respuesta será negativa. Sin embargo, si invertimos el orden de las reglas, 2 y 4, la respuesta se hace afirmativa.
Notar además que el programa no funciona para modo de uso (out).

3.3.3. Generación y prueba

Se usa también el corte para acabar una ***secuencia de generación y prueba*** en programas con la siguiente estructura:

1.- Una serie de fines generan posibles soluciones vía "backtracking".
2.- Otros fines comprueban si dichas soluciones son las apropiadas.
3.- Se obtiene una solución generada que verifique las pruebas.

Ejemplo: Ordenación de una lista buscando una permutación ordenada de la lista dada. Modo de uso: **(in, out).**

Suponemos que **permutar(X,Y)** obtiene en Y una permutación de X. Usar este fin como generador significa que al hacer una llamada como **?permutar(dato, Z)** se obtienen distintas soluciones vía "backtracking" en **Z.**

```
ordenar(X,Y) :- permutar(X,Y), ordenada(Y), !.
                (1) generador  (2) prueba  (3) obtención
```

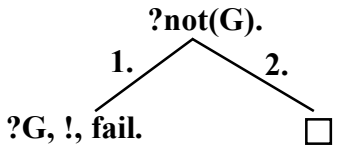
- El fin generador **(1)**, **permutar**, obtiene una permutación de X en Y que pasa al fin prueba para comprobar si Y está ordenada.
- El fin prueba **(2)**, **ordenada**, mira si la lista está ordenada. Si no lo está la vuelta atrás se encargará de re-satisfacer el primer fin buscando una nueva permutación.
- Si lo está, es la solución buscada y se da como respuesta. El corte impide la vuelta atrás **(3)**, por lo que no se generarán más permutaciones.

3.4. La negación como fallo finito

La combinación **corte-fallo** vista en la sección anterior, puede usarse para implementar una forma débil de negación en PROLOG, mediante el *meta-predicado* **not** definido:

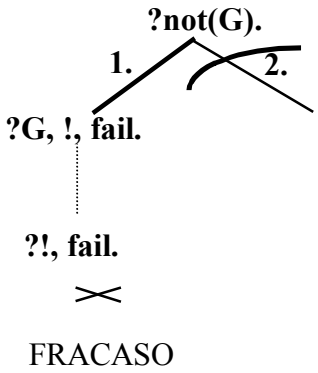
```
1.  not(X) :- X, !, fail.
2.  not(X).
```

En **not(X)** se usa una *meta-variable* **X** que *se unificará con un átomo*, *no con un término*. El comportamiento de este programa, al contestar una pregunta del tipo **?not(G)** donde **G es un fin**, es el siguiente:

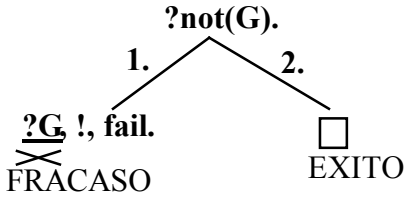


El fin **not(G)** se unifica con la primera regla del programa (ligando X con el **fin** G) y produce el objetivo **?G, ! ,fail.**

Si G se satisface entonces se encuentra el corte **!**
que poda la segunda rama del árbol y "fail" hace que
el fin not(G) fracase.



Si G fracasa entonces se usa la segunda regla y
el fin not(G) se satisface



La terminación del cómputo para una pregunta **?not(G)** depende de la terminación del cómputo para **?G**. Si éste no termina, entonces el primero puede terminar o no, según que en el árbol de búsqueda para **?G** se encuentre una refutación antes que una rama infinita.

Cabe señalar que el uso de la ***negación en PROLOG no se corresponde con la negación lógica***. Por ello, es importante indicar que una pregunta negada que contenga variables, no tendrá como respuesta valores para dichas variables. Es decir, **?not(G)** siempre tendrá como respuesta un **si** (éxito) o un **no** (fracaso), según corresponda, aun cuando G sea un átomo con variables. Además la respuesta, en dicho caso, puede ser incorrecta, como se muestra en el siguiente ejemplo.

Ejemplo:
Dado el programa **P = {p(a). q(b).}** y dado el objetivo con variables **G = p(X)** se tiene:

```
? not(p(X))
no
```

debido a que Prolog pregunta por **?p(X)** y éste se satisface para **X = a.**
Es decir, el objetivo **G = p(X),** que se corresponde con la fórmula **∃X p(X),** se ha hecho cierto**.** Por tanto, la negación de esta fórmula **¬∃X p(X)** equivalente a **∀X¬p(X)** ha de ser **falsa.** La respuesta **"no"** es correcta si se corresponde con preguntar por **∀X¬p(X) .**
Sin embargo, la interpretación correcta de la pregunta **?not(p(X))** , haciendo explícito el cuantificador, es la fórmula **∃X ¬p(X)** que, respecto al programa **P,** debería dar como respuesta **X = b.**
El problema es que **∀X¬p(X)** y **∃X ¬p(X)** son fórmulas no equivalentes.

Para evitar esta incorrección, ***se debe asegurar el uso de la negación en PROLOG sólo sobre átomos de base (sin variables)*** ya que en dicho caso **∀X¬A = ∃X¬A = ¬A**