**Clase 3.2: El corte: estructura de control explícita. La negación en la Programación Lógica.**

**Profesores:** Dra. Yisel Garí

## El corte: estructura de control explícita

Como vimos la programación lógica pura en su realización en Prolog resuelve el problema del control en la ejecución de algoritmos adoptando BPP-RC como la estrategia de navegación en el espacio de búsqueda. En verdad, el lector se puede percatar de que el árbol de derivación de un objetivo con respecto a un programa puede contener muchas ramas que terminan en fracasos, las que BPP-RC no puede evitar generar. No obstante, nuestro conocimiento sobre cómo el intérprete ejecuta nuestro programa puede ser utilizado para realizar "podas" de tales ramas estériles en la deducción del objetivo, lo cual resultaría en una mayor eficiencia.

Prolog suministra un predicado primitivo que permite al programador afectar la generación del árbol de deducción de un objetivo eliminando ramificaciones que a juicio del programador den lugar a ramas fracasos. Se trata de un predicado 0-ario, denominado corte(cut) y denotado por !/0. El corte puede ser introducido como un sub-objetivo más en el cuerpo de cualquier cláusula o de un objetivo, su valor como constante proposicional es verdadero, por lo tanto, como objetivo a evaluar en cualquier derivación el corte siempre triunfa, de ahí que no afecte para nada la lectura y semántica declarativas de un programa si es correctamente utilizado. Su efecto sobre la ejecución del intérprete es relevante: actúa como un mecanismo de control que reduce el espacio de búsqueda podando dinámicamente ramas del árbol de deducción de un objetivo con respecto a un programa que supuestamente no conducirían a soluciones. Veamos a través de un ejemplo la mecánica del corte.

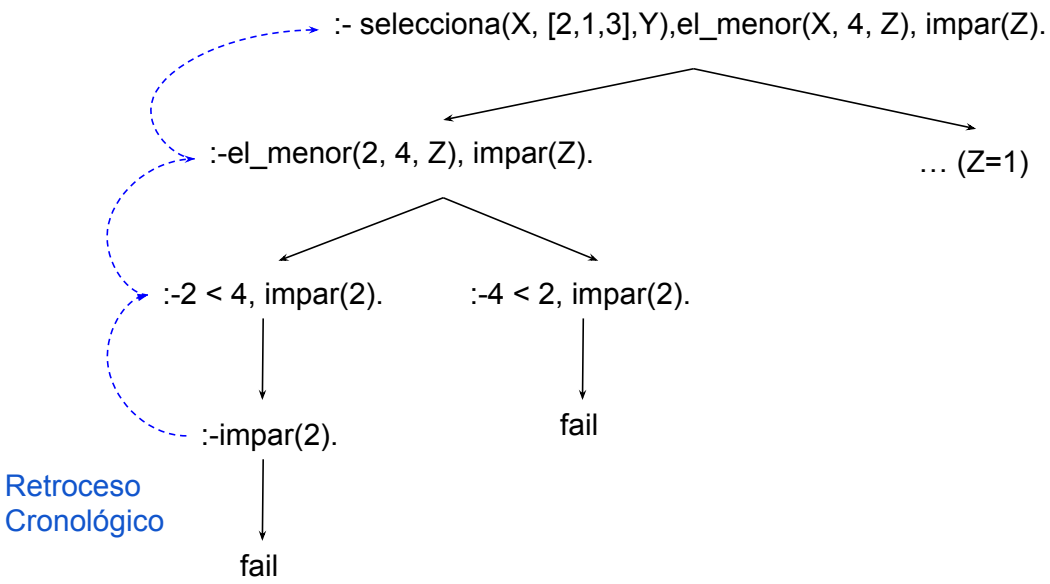
Ejemplo: Considere el siguiente programa donde se define el predicado el_menor/3 para hallar el menor entre dos números (note que el predicado fracasa si los números son iguales):

```
el_menor(X, Y, X) :- X < Y.
el_menor(X,Y,Y) :- Y < X.
```

Considere el siguiente objetivo que triunfa, si puede seleccionar de una lista de números un número que cumpla con la condición de ser menor que 4 y ser impar:

```
:-selecciona(X, [2, 1, 3], Y), el_menor(X, 4, Z), impar(Z).
```

El árbol de deducción simplificado es el siguiente:

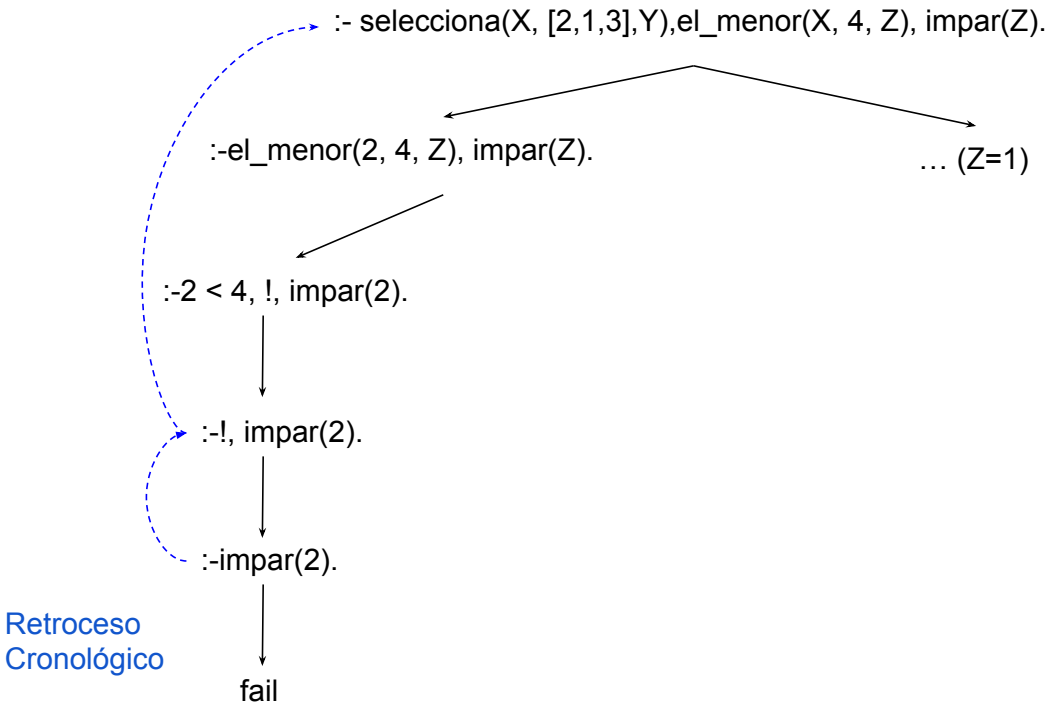



Se puede apreciar que el intérprete genera innecesariamente la rama intermedia que es un fracaso.

Considere ahora el siguiente programa del predicado el_menor/3, en cuya definición ocurre el corte.

```
el_menor(X, Y, X) :- X < Y, !.
el_menor(X,Y,Y) :- Y < X.
```

y evalúese nuevamente con este programa el objetivo
`:-selecciona(X, [2, 1, 3], Y), el_menor(X, 4, Z), impar(Z).`



Analizando el nuevo árbol de deducción se verifica lo siguiente:
Cuando algún objetivo a la derecha del corte fracasa (en el ejemplo: impar(2)), si el corte se alcanza por retroceso cronológico (como sucede en el ejemplo), esto trae como consecuencia que el intérprete:

1. No reevalúe ninguno de los sub-objetivos a la izquierda del corte (en el ejemplo: 2 <4)
2. Tampoco selecciona ninguna de las restantes cláusulas que pudiesen aplicarse del predicado que introdujo el corte (en el ejemplo: la segunda cláusula de el_menor/3).
3. El intérprete reinicia el proceso de deducción en el sub-objetivo inmediatamente a la izquierda del sub-objetivo que introdujo el corte (en el ejemplo selecciona(X, [2, 1, 3], Y)).

**Control del no-determinismo** Como hemos visto una de las características más interesantes de la PL es la posibilidad de definir varias alternativas que puede tener un predicado (procedimiento) y el no determinismo asociado. Sin embargo, como hemos visto en el ejemplo desarrollado, pueden definirse predicados en los que la aplicación de una de las cláusulas por el intérprete para evaluar un objetivo excluye la aplicación de otras cláusulas del mismo predicado.

Considere como caso más convincente el siguiente predicado fusion_ordenada(X, Y, Z) donde Z es la lista ordenada de enteros que se obtiene al fundir las listas ordenadas de enteros X y Y:

```
fusion_ordenada(X, [], X).
fusion_ordenada([], X, X).
fusion_ordenada([X |X1],[Y |Y1],[X |Z]):- X < Y, fusion_ordenada(X1, [Y |Y1], Z).
fusion_ordenada([X |X1],[Y |Y1],[X, Y |Z]):- X = Y, fusion_ordenada(X1, Y1, Z).
fusion_ordenada([X |X1],[Y |Y1],[Y |Z]):- X > Y, fusion_ordenada([X |X1],Y1,Z).
```

Observe que en las cláusulas que definen este predicado no sólo la unificación del objetivo con las cabezas de las cláusulas es determinante en su aplicación, también lo es el primer sub-objetivo en el cuerpo de cada regla que, como puede apreciarse, es una comparación de números. Se verifica entonces que una vez realizada la comparación, si esta resulta verdadera, se continúa con la aplicación del predicado fusion_ordenada/3 y resulta innecesario y por demás deficiente que el intérprete debido a su estrategia BPP-RC intente aplicar otra de las cláusulas si hubiese un fracaso posterior. Esto se debe además a que el predicado fusion_ordenada/2 es esencialmente funcional y la derivación de un objetivo de la forma:
`:- fusion_ordenada([1, 3, 5], [2, 4], X).`
tiene una sola solución. La introducción conveniente del corte en las cláusulas de este predicado permite declarar la aplicación mutuamente excluyente o determinista:

```
fusion_ordenada(X, [], X).
fusion_ordenada([], X, X).
fusion_ordenada([X |X1],[Y |Y1],[X |Z]):- X < Y, !, fusion_ordenada(X1, [Y |Y1], Z).
fusion_ordenada([X |X1],[Y |Y1],[X, Y |Z]):- X = Y, !, fusion_ordenada(X1, Y1, Z).
fusion_ordenada([X |X1],[Y |Y1],[Y |Z]):- X > Y, fusion_ordenada([X |X1],Y1,Z).
```

Es posible utilizando el corte modificar la definición de un predicado existente de tal manera que se pueda obtener un nuevo predicado con distinta funcionalidad.
Ejemplo: Analice la siguiente modificación del predicado member/2 y determine qué nuevo predicado resulta ser member_1/2 y cuál es su diferencia con member/2.

```
member_1(X, [X| _]):- !.
member_1(X, [_|Y]):- member_1(X,Y).
```

Estos usos del corte no afectan la lectura declarativa de un programa.
**Usos incorrectos del corte** Sin embargo puede haber usos incorrectos del corte que pueden atentar contra la corrección de un programa. Por ejemplo, utilizándolo para omitir condiciones en la definición de un predicado.
Analice las siguientes dos definiciones del predicado min/3:

```
% min(N1,N2;M): M el mínimo de los números N1 y N2
min1(X,Y,X):- X =< Y, !.
min1(X,Y,Y):- Y < X.

min2(X,Y,X):- X =< Y, !.
min2(X,Y,Y).
```

El lector verificará que la primera definición (min1/3) hace un uso correcto del corte. La segunda definición (min2/3) tiene toda la apariencia de un "if then else" al omitirse la comparación en la segunda cláusula. Considere ahora el siguiente objetivo:

```
:- min(2,5,5).
```

El lector verificará que con min1/3 se obtienen la respuesta negativa correcta mientras que con min2/3 se obtiene una respuesta afirmativa incorrecta.

El corte, como estructura de control explícita que puede ser de utilidad en lograr una mayor eficiencia en la ejecución de un programa debe ser utilizado sin afectar la interpretación proyectada de un predicado.

Debe enfatizarse que el uso del corte no es necesario: nada agrega a la capacidad computacional de Prolog, su uso es en aras de la eficiencia en la búsqueda y su empleo debe reducirse a este fin.

## La negación en Prolog

Si se analiza un programa lógico, por ejemplo, el programa Familia, se verá que se trata de un conjunto de cláusulas (hechos, reglas) que enuncian solo conocimiento positivo. Las cláusulas definidas de un programa lógico no permite expresar conocimiento negativo: los hechos son literales positivos cuya verdad se afirma en el programa. Las reglas del programa contienen sólo literales positivos en la cabeza y en el cuerpo, en esencia, las reglas son implicaciones que sólo permiten derivar hechos positivos.

Sin embargo, no es posible prescindir de la negación para representar y procesar conocimiento acerca de muchos problemas. No se puede por ejemplo preguntar negativamente al programa familia. Tampoco es posible escribir un programa definitorio de listas disjuntas, es decir, es decir de listas que no tienen ningún elemento en común.

No es necesario argumentar aquí lo importante que resulta extender la PL de conocimiento positivo a una PL que permita de alguna manera la representación y el procesamiento lógico de conocimiento negativo. Los desarrollos que siguen establecen la forma en que Prolog ha sido

habilitado para esta extensión.

Se introduce en Prolog la operación proposicional de la negación como un predicado unario not/1.

Los intérpretes modernos usan el símbolo \+ ( mnemotécnica para "no probable") aunque aceptan ambos por compatibiliad con códigos anteriores.

Ejemplo:
?- \+(2 = 4).
true.

?- not(2 = 4).
true.

¿Cuál puede ser la interpretación que defina semánticamente a not/1 en Prolog?. Evidentemente no puede ser la interpretación de la negación en LPO ya estudiada dado que como se ha visto la sintaxis no permite la afirmación de hechos negativos y los predicados se definen intencionalmente mediante reglas que son implicaciones en un solo sentido, es decir, las reglas declaran la parte "si" de una definición, no la parte "solo si".
No obstante nada impide extender la sintaxis de Prolog introduciendo objetivos negados y también realizar definiciones de predicados utilizando la negación en el cuerpo de las cláusulas, teniendo en cuenta que cuando el intérprete aplica una cláusula de un programa, su cuerpo se convierte en parte del objetivo.

Lo que no se declara como un hecho (conocimiento positivo) es un hecho falso, por lo tanto, la negación de un hecho (conocimiento negativo) es verdadera si el hecho no está declarado. (Hipótesis del Mundo Cerrado (HMC)(Closed World Hipótesis (CWH)).

¿Cuál sería una definición admisible del predicado not/1? Aprovechando los fracasos se podría quizás introducir la HMC como una "regla de inferencia" más en el intérprete de Prolog que se ocuparía de interpretar la negación. Su definición como tal podría ser la siguiente:

**Negación como Fracaso Finito (NFF)**
Sea P un programa y G un literal, entonces
Existe refutación-SLD de P ∪ {:- not G} si y solo si P ∪ {:- G} fracasa finitamente.

Además G deberá ser básico (recordatorio: no hay ocurrencia de variables en G).

Al añadir tal regla al intérprete denominaremos a su nuevo mecanismo de inferencia Resolución-SLDNFF.
Ejemplos:

1. se verifica que Familia ∪ {`:- abuelo(a, e.)`} fracasa finitamente, luego aplicando la regla anterior se tiene que Familia ∪ {`:- not abuelo(a, e.)`} triunfa.

2. el lector analizará la siguiente definición del predicado disjuntas/2 que verifica si dos listas no tienen ningún elemento en común,
```
disjuntas(Lista1, Lista2):-not((member(X, Lista1), member(X, Lista2))).
```

3. analice la propuesta de una definición del predicado union/3 que halla la unión de dos listas

```
union([],X,X).
union([X|Y],Z,[X|W]) :- not(member(X,Z)) , union(Y,Z,W).
union([X|Y],Z,W) :- union(Y,Z,W).
```

Denominaremos *conjunto de fracasos finitos* de un programa lógico P al conjunto de objetivos G tal que G fracasa finitamente con respecto a P. Luego un objetivo {`:- not(G)`} es una consecuencia de un programa P por la regla NFF si G pertenece al conjunto de fracasos finitos de P.