Programación en PROLOG(2)

Inteligencia Artificial
2º cuatrimestre de 2009

Departamento de Ciencias e Ingeniería de la Computación

Universidad Nacional del Sur

Operador de corte (cut): Motivaciones

- Prolog es un lenguaje de programación altamente declarativo.
- Para alcanzar esta declaratividad, en ocasiones se paga un alto costo computacional.
- Los diseñadores del lenguaje incorporaron un operador de corte que permite podar los espacios de búsqueda.

Cut (!): Definición formal

- El operador de corte (cut) es un predicado predefinido notado !.

- Operacionalmete se define como sigue:

La meta '!' siempre tiene éxito y provoca el descarte de todas las (ramas) alternativas que quedaron pendientes de ser exploradas desde el instante en que se utilizó para resolver la regla conteniendo dicho '!'.

Cut (!): Implicancias de la Definición

1. El cut poda alternativas correspondientes a claúsulas por debajo de él.

```
p:- q, !, r.
p:- s, t.
```

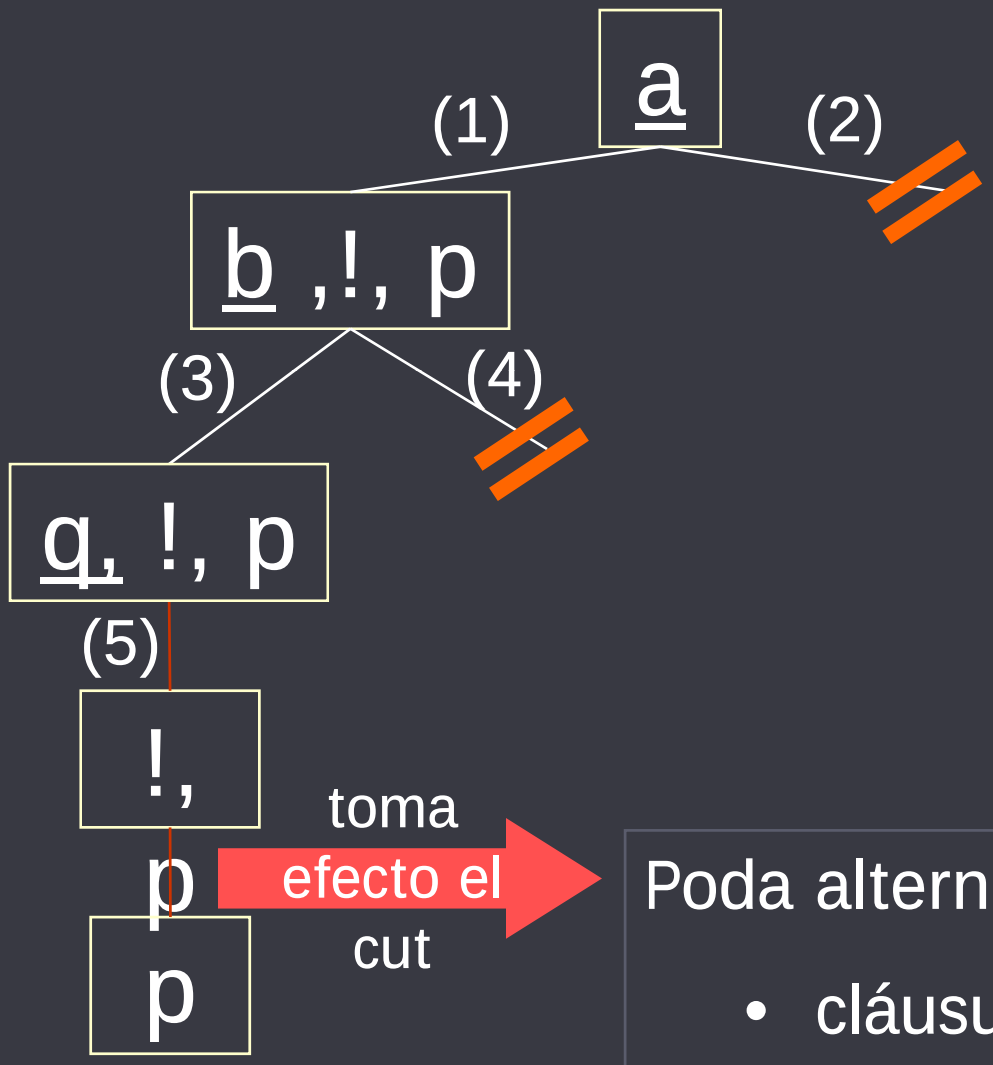
2. El cut poda las soluciones alternativas de la meta conjuntiva a la izquierda del mismo.

```
p:- q, r, !, s, t.
```

3. El cut permite soluciones alternativas en la meta conjuntiva a la derecha del mismo.

```
p:- q, r, !, s, t.
```

Cut (!): Ejemplo




```
1. a:–b,!, p.
2. a:–r.
3. b:–q.
4. b:–e.
5. q.
6. r.
```

```
?–a.
no
```

Poda alternativas correspondientes a:

- cláusulas por debajo
- soluciones alternativas a izquierda

Aplicaciones

- Entre sus principales aplicaciones se encuentran:
  - Expresar determinismo.
  - Omitir condiciones.
  - Implementar una forma de negación diferente a la clásica, denominada negación por falla.

Aplicaciones: expresar determinismo

- Consideremos un predicado definido de la siguiente forma:

```
p(…):- c1(…), ….
p(…):- c2(…), ….
…
p(…):- cn(…), ….
```

condiciones

- Si las condiciones son mutuamente excluyentes, podemos mejorar la eficiencia colocando un cut luego de cada condición:

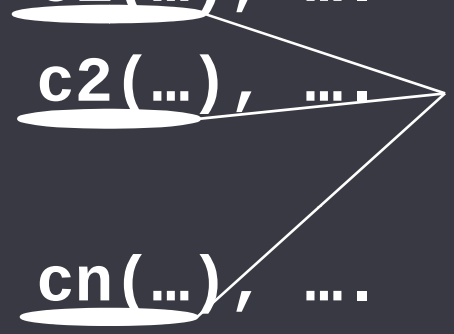
```
p(…):- c1(…), !, ….
p(…):- c2(…), !, ….
…
p(…):- cn(…), !, ….
```

Este último cut no es necesario

Aplicaciones: omitir condiciones

- Consideremos un predicado definido de la siguiente forma:

```
p(…):- c1(…), ….
p(…):- c2(…), ….
…
p(…):- cn(…), ….
```



- Si además las condiciones son exhaustivas, podemos omitir la última de ellas al agregar los cuts.

```
p(…):- c1(…), !, ….
p(…):- c2(…), !, ….
…
p(…):- cn-1(…), !, ….
p(…):- cn(…), !, ….
```

Ejemplo

- Dados dos números naturales X e Y, hallar la diferencia absoluta entre X e Y:

```
dif_abs(X, Y, Z):- X >=Y, Z is X-Y.
dif_abs(X, Y, Z):- X < Y, Z is Y-X.
```

- Como las condiciones son mutuamente excluyentes, podemos mejorar la eficiencia utilizando cuts:

```
dif_abs(X, Y, Z):- X >=Y, !, Z is X-Y.
dif_abs(X, Y, Z):- X < Y, Z is Y-X.
```

- Como además las condiciones son exhaustivas, directamente podemos omitir la última de ellas.

```
dif_abs(X, Y, Z):- X >=Y, !, Z is X-Y.
dif_abs(X, Y, Z):-           Z is Y-X.
```

Cuidado! Ahora este cut resulta fundamental para la correctitud del programa.

Advertencia

- El operador de corte debe ser empleado con cuidado y analizando las consecuencias en cada situación.

- Si se emplea mal puede causar comportamiento poco predecible.

Negación por falla: not/1

- not/1 es un predicado de segundo orden, i.e. recibe como argumento a otro predicado (más precisamente una consulta).
- **`not(Goal)` tiene éxito** cuando **Goal** no puede ser probado.

```
a:- b, p.
b:- e.
e.
```

```
?- a.
no
```

```
?- not(a).
yes
```

```
?- b.
yes
```

```
?- not(b).
no
```

```
?- member(X, [1,2]).
X=1
X=2
```

```
?- not(member(X, [1,2])).
no
```

Implementación del not mediante ! y fail

- La negación por falla (pred. not/1) se define en términos de los predicados ! y fail/0.
- El único efecto del predicado predefinido fail/0 es fallar.

```
not(Goal):- Goal, !, fail.

not(_).
```

- Comportamiento de **not/1**: Si la meta **Goal** se satisface, se dispara la poda y **not(Goal)** falla. Caso contrario, si la meta **Goal** falla, el cut no se ejecuta y **naf(Goal)** se satisface.

Predicados all-solutions: findall/3

```
findall(+Term, +Goal, -ListOfTerms)
```

pred. de 2do orden

- Retorna en **`ListOfTerms`** la lista de todas las instanciaciones de **`Term`** correspondientes a soluciónes alternativas (en virtud del backtracking) de la meta **`Goal`**.
- Ejemplos:

```
?- findall(X, member(X,[1,2,3]), Rta).
Rta = [1,2,3]
```

```
?-findall(s(X,Y),(member(X,[1,2,3]),Y is X+1),Rta).
Rta = [s(1, 2), s(2, 3), s(3, 4)]
```

findall/3: Ejemplo

- El producto cartesiano entre dos conjuntos **A** y **B** (**AxB**) es el conjunto de todos los pares **(Ea,Eb)** tal que **Ea** pert. **A** y **Eb** pert. **B**.

- Definir un predicado **prod_cart/3** que dados dos conjuntos **A** y **B** retorne **AxB**, el producto cartesiano entre **A** y **B**.

```
?- prod_cart([1,2], [a,b,c], Rdo).
Rdo = [[1,a],[1,b],[1,c],[2,a],[2,b],[2,c]]
```

```
prod_cart(A, B, AxB):-
 findall([Ea,Eb], (member(Ea,A),member(Eb,B)), AxB)
```

Predicados all-solutions: forall/2

```
forall(+Ant, +Cons)
```

- Tiene éxito si para todas las soluciones alternativas de **Ant** (antecedente) se verifica **Cons** (consecuente).

Ejemplo de uso del forall/2

lista_num_pares(+L)

- Tiene éxito si **L** es una lísta conteniendo números pares.

```
?- lista_num_pares([2,4,5]).
no

?- lista_num_pares([2,4,6]).
yes
```

```
lista_num_pares(L):-
    forall(member(X,L), 0 is X mod 2).
```

Ejemplo de uso del forall/2

seguidos(X,Y,L)

- Tiene éxito si **Y** está inmediatamente después que **X** en la lista **L**.

```
seguidos(X,Y, [X,Y|_RestoL]).
seguidos(X,Y, [_Z|RestoL]):-
                        seguidos(X,Y,RestoL).
```

---

ordenada(+L)

- Dada una lista de números **L**, determina si se encuentra ordenada (de menor a mayor).

```
ordenada(L):-
  forall(seguidos(X,Y,L), X<Y).
```

Manipulación de estructuras

• Hay predicados especiales para analizar la estructura de términos compuestos.

=../2 ejemplo

```
?- h(a,g(X),8) =.. L.
L = [h, a, g(x), 8]

?- T =.. [f, a, b].
T = f(a,b).
```

---

compound/1 ejemplo

```
?- compound(a).
no

?- compound(f(a)).
yes
```

Manipulación de estructuras

functor/3

ejemplo

```
?- functor (h(a,g(X),8), Functor, Aridad).
Functor = h
Aridad = 3
```

---

arg/3 ejemplo

```
?- arg (2, h(a,g(2),8), SegArg).
SegArg = g(2)
```

Modificando el Programa dinámicamente

• SWI-PROLOG ofrece toda una familia de predicados que permiten **agregar y quitar dinámicamente hechos y reglas** del programa, es decir, permiten modificar dinámicamente definiciones de predicados.

`assert(+Term)`

• recibe como argumento un término representando un hecho o regla y lo agrega al programa. **`Term`** se agrega como último hecho o regla del predicado correspondiente.

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
juegaBienTruco(pablo).
```

```
?- assert(mentiroso(juancito)).
```

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
mentiroso(juancito).
juegaBienTruco(pablo).
```

```
?- assert(mentiroso(juancito)).
yes
?- assert(mentiroso(leo)), fail.
```

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(leo).
juegaBienTruco(pablo).
```

```
?- assert(mentiroso(juancito)).
yes
?- assert(mentiroso(leo)), fail.
no
```

El backtracking no deshace el efecto colateral provocado por el assert.

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(leo).
juegaBienTruco(pablo).
```

```
?- assert(mentiroso(X):- juegaBienTruco(X)).
```

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(leo).
mentiroso(X):- juegaBienTruco(X).
juegaBienTruco(pablo).
```

```
?- assert(mentiroso(X):- juegaBienTruco(X)).
yes
```

Modificando el Programa dinámicamente

`retract(+Term)`

- Quita del programa el primer hecho o regla que unifique con **`Term`**. Responde **no** si tal hecho o regla no existe .

- Aclaración: **`Term`** no puede ser simplemente una variable.

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(leo).
mentiroso(X):- juegaBienTruco(X).
juegaBienTruco(pablo).
```

```
?- retract(mentiroso(leo)).
```

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(leo).
mentiroso(X):- juegaBienTruco(X).
juegaBienTruco(pablo).
```

```
?- retract(mentiroso(leo)).
yes
```

Modificando el Programa dinámicamente

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(X):- juegaBienTruco(X).
juegaBienTruco(pablo).
```

```
?- retract(mentiroso(Y)).
Y=jaimito ;
Y=juancito ;
no
```

pido solución ALTERNATIVA

pido solución ALTERNATIVA

La exploración de soluciones alternativas tiene efectos colaterales!!!

Modificando el Programa dinámicamente

Otros predicados de la misma familia:

- **`asserta(+Term)`**: equivalente a **`assert/1`** sólo que **`Term`** se agrega como primer hecho o regla del predicado correspondiente.

- **`retractall(+Head)`**: quita del prog. todas las cláusulas (hechos y reglas) cuyas cabezas unifiquen con **`Head`**.

- **`abolish(+PredName, +Arity)`**: quita del programa todas las cláusulas (hechos y reglas) del predicado **`PredName`** de aridad **`Arity`**.

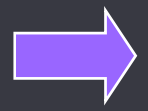
retractall vs. retract

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(X):-
        jBienTruco(X).
jBienTruco(pablo).
```

```
?- retractall(mentiroso(X)).
yes
```

```
mentiroso(jaimito).
mentiroso(juancito).
mentiroso(X):-
        jBienTruco(X).
jBienTruco(pablo).
```

```
?- retract(mentiroso(X)).
X=jaimito ;
X=juancito ;
no
```



pido solución ALTERNATIVA

pido solución ALTERNATIVA

Modificando el Programa dinámicamente

`dynamic +Name/+Arity`

- Informa al intérprete que la definición del predicado puede cambiar durante la ejecución (usando **`assert/1`** y/o **`retract/1`**).

- Su utilidad tiene que ver, principalmente, con el control de acceso concurrente y la compilación de predicados.

Modificando el Programa dinámicamente

• Además, como ayuda al programador, el intérprete del SWI diagnostica el error "predicado indefinido" frente a una consulta a un predicado que no posee reglas ni hechos en el programa. Ejemplo:

```
p(a).
p(X):-q(X).

?- r(b).

ERROR: Undefined procedure: r/1
```

```
?- dynamic r/1.
yes
?- r(b).
no
```

• Este mensaje, en lugar de un **no**, resulta de gran ayuda para detectar errores en nombres de predicados, cometidos accidentalmente.

• Si hubiésemos declarado a r/1 como dinámico, entonces la respuesta del intérprete sería simplemente **no**, ya que sabe que se trata de un predicado dinámico que no posee reglas o hechos en ese momento.

Invocando metas: call/1

call(+Goal)

- Invoca la meta **Goal**/ realiza la consulta por **Goal**.

```
?- call(member(2,[1,2])).
yes
```

```
?- call(member(X,[1,2])).
X=1;
X=2;
no
```

Flexibilidad de PROLOG

La combinación de los siguientes elementos brinda mucha flexibilidad:

- la capacidad de invocar metas (**`call/1`**) y de agragar y quitar hechos y reglas (**`assert/1`** y **`retract/1`**) en forma dinámica, donde tanto metas, hechos y reglas se representan mediante estructuras.

- las facilidades para manipular (descomponer, construir, analizar) estructuras (`=../2`)

Flexibilidad de PROLOG: Ejemplo

```
suma(X,Y,Rdo):- Rdo is X+Y.
resta(X,Y,Rdo):- Rdo is X-Y.
  ⋮
pot(X,Y,Rdo):- Rdo is X^Y.

operar(Op,A,B,Rdo):- AOpB =..[Op,A,B,Rdo],
                     call(AOpB).
```

construcción de una estructura representando una consulta

invocación de la consulta

```
?-operar(pot,2,3, Rdo).
Rdo=8
yes
```

Carga de un archivo .pl

consult(+File)

• Recibe como argumento el path de un archivo conteniendo código PROLOG, y lo carga en la base de datos del intérprete.

• Un archivo .pl puede contener cláusulas especiales de la forma

```
:- Goal.
```

• **Goal** se invocará automáticamente cuando se consulte el archivo.

Carga de un archivo .pl

• La cláusula **`:-Goal`** puede ser útil para invocar predicados de inicialización o declarar predicados como dinámicos. Ejemplo:

programa.pl

```
:- setup.
:- dynamic p/1, q/2.
setup:- …
p(a).
q(b).
```

Más info sobre estos predicados y otros:

On Line Manual del SWI PROLOG

FIN