# 7.11 O operador de corte

Representado pelo átomo especial `!`.

Pode ser usado como literal no corpo de uma cláusula.

**O que faz?**
Impede a geração de determinados ramos da árvore SLD.

**Vantagem:**
Evita gerar ramos que sabemos não levar a soluções.

**Desvantagem:**
Alteração inadvertida da semântica declarativa de um programa.

# 7.11 O operador de corte

**Como funciona?**

Literal ! tem sempre sucesso e compromete o PROLOG com todas as escolhas feitas desde que o objectivo foi unificado com a cabeça da regra que contém o corte até ao literal !.

# 7.11 O operador de corte

Por exemplo, suponhamos que o PROLOG usa a regra

```
p :- a_c_1, ..., a_c_n, !, d_c_1 ,..., d_c_m.
```

para satisfazer um objectivo p.

Uma vez atingido o corte o PROLOG não pode

- fazer novas escolhas para `p, a_c_1, ..., a_c_n`;

Pode no entanto

- explorar outras alternativas para `d_c_1, ..., d_c_m`;
- explorar outras alternativas para escolhas feitas antes de atingir o objectivo `p`.

# 7.11 O operador de corte

## Exemplo - Programa

```
p(X) :- q(X).
p(X) :- r(X), t(X).
p(X) :- u(X).

q(1).
r(1).
r(2).
t(2).
u(3).
```

## Exemplo - Objectivo

```
?- p(X).
X = 1 ;
X = 2 ;
X = 3.
```

# 7.11 O operador de corte

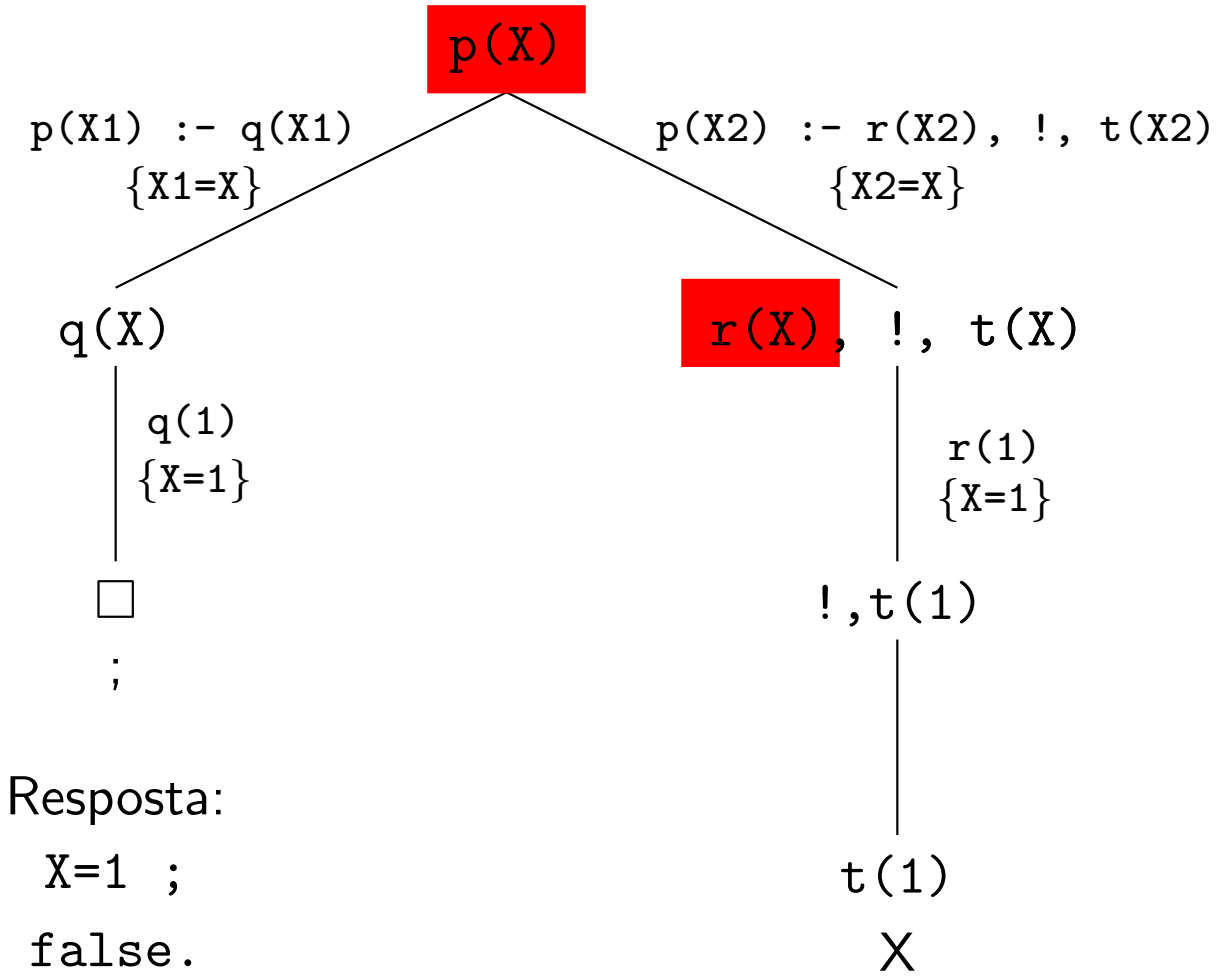
## Exemplo - Programa com corte

```
p(X) :- q(X).
p(X) :- r(X), !, t(X).
p(X) :- u(X).

q(1).
r(1).
r(2).
t(2).
u(3).
```

## Exemplo - Objectivo

```
?- p(X).
???
```

# 7.11 O operador de corte




Não é possível fazer outras escolhas nos nós dentro dos rectângulos vermelhos.

# 7.11 O operador de corte

## Predicado `diferenca` errado

```
diferenca([], _, []).
diferenca([P | R], L2, D) :-  membro(P, L2),
                              diferenca(R, L2, D).
diferenca([P | R], L2, [P | D]) :- diferenca(R, L2, D).
```

## Utilização

```
?- diferenca([a,b,c,d,e],[b,d],D).
D = [a, c, e] ;
D = [a, c, d, e] ;
D = [a, b, c, e] ;
D = [a, b, c, d, e].
```

## Problema

A 2ª regra pode ser usada, mesmo que `membro(P, L2)` se verifique.

# 7.11 O operador de corte

Vamos usar o corte para impedir a utilização da 2ª regra, quando `membro(P, L2)` se verificar.

Onde colocar o corte? A seguir ao literal `membro(P, L2)`.

**Predicado `diferenca` corrigido**

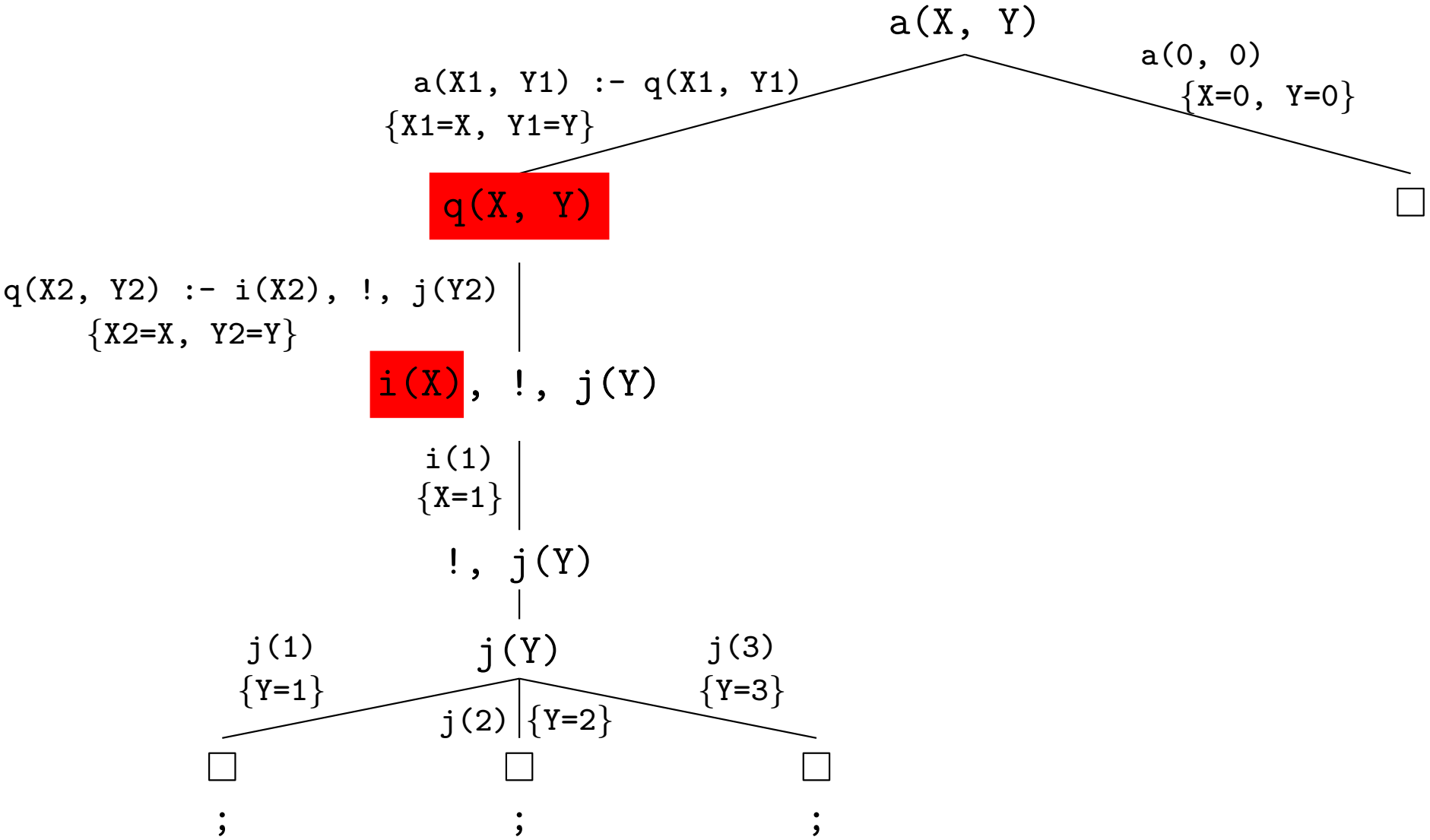
```
diferenca([], _, []).
diferenca([P | R], L2, D) :-  membro(P, L2),
                              !,
                              diferenca(R, L2, D).
diferenca([P | R], L2, [P | D]) :- diferenca(R, L2, D).

?- diferenca([a,b,c,d,e],[b,d],D).
D = [a, c, e].
```

# 7.11 O operador de corte

**Exemplo 7.11.3 (Utilização do operador de corte)**

```
a(X, Y) :- q(X, Y).

a(0, 0).

q(X, Y) :- i(X), !, j(Y).

q(5, 5).

i(1).
i(2).
j(1).
j(2).
j(3).
```

# 7.11 O operador de corte



**Exemplo 7.11.4 (Divisão de uma lista)**

`parte(L, N, L1, L2)` afirma que
os elementos da lista `L1` são os elementos de `L` menores do que `N` e
os elementos da lista `L2` são os elementos de `L` maiores do que `N`.

Por exemplo,
`parte([2, 3, 5, 7, 1], 4, [2, 3, 1], [5, 7])`

**Programa**

```
parte([], _, [], []).

parte([P | R], N, [P | R1], L2) :- P < N,
                                   parte(R, N, R1, L2).

parte([P | R], N, L1, [P | R2]) :- P >= N,
                                   parte(R, N, L1, R2).
```

# 7.11 O operador de corte

## Utilização

```
?- parte([2, 3, 5, 7, 1], 4, L1, L2).
L1 = [2, 3, 1],
L2 = [5, 7] ;
false.

?- parte(L, 4, [3], [6,7,8]).
L = [3, 6, 7, 8] ;
L = [6, 3, 7, 8] ;
L = [6, 7, 3, 8] ;
L = [6, 7, 8, 3] ;
false.

?- parte(L, 4, [5], [6,7,8]).
false.
```

# 7.11 O operador de corte

Programa correcto, no entanto mesmo quando a 1ª regra tem sucesso (`P < N`), o PROLOG vai tentar usar a 2ª regra, que falhará sempre.

Como indicar que, no caso de `P < N` ter sucesso, não deve ser experimentada a 2ª regra?

**Programa com corte**

```
parte([], _, [], []) :- !.

parte([P | R], N, [P | R1], L2) :- P < N,
                                   !,
                                   parte(R, N, R1, L2).

parte([P | R], N, L1, [P | R2]) :- P >= N,
                                   !,
                                   parte(R, N, L1, R2).
```

# 7.11 O operador de corte

A utilização do operador de corte na 2ª regra (após `P >= N`) é desnecessária, sendo utilizado apenas por uma questão de simetria entre as duas regras.

Por que é desnecessária? Porque não há mais cláusulas na definição do predicado `parte`.

# 7.11 O operador de corte

## Utilização do programa com corte

```
?- parte([2, 3, 5, 7, 1], 4, L1, L2).
L1 = [2, 3, 1],
L2 = [5, 7].

?- parte(L, 4, [3], [6,7,8]).
L = [3, 6, 7, 8].
```

# 7.11 O operador de corte

Poderíamos pensar:

A 2ª regra só é usada se a 1ª falhar.

A 1ª só falha se `P < N` falhar.

Logo, a 2ª só é usada se `P >= N`, e este teste é desnecessário.

Argumento válido, mas ... a 2ª premissa não é verdadeira:

A 1ª regra pode falhar porque a sua cabeça,

```
parte_com_erro([P | R], N, [P | R1], L2)
```

não unifica com o objectivo.

# 7.11 O operador de corte

**Programa errado:**

```
parte_com_erro([], _, [], []).

parte_com_erro([P | R], N, [P | R1], L2) :-
                        P < N,
                        !,
                        parte_com_erro(R, N, R1, L2).

parte_com_erro([P | R], N, L1, [P | R2]) :-
                        parte_com_erro(R, N, L1, R2).
```

# 7.11 O operador de corte

## Utilização

```
?- parte_com_erro([2, 3, 5, 7, 1], 4, L1, L2).
L1 = [2, 3, 1],
L2 = [5, 7].

?- parte_com_erro(L, 4, [3], [6,7,8]).
L = [3, 6, 7, 8] ;
false.

?- parte_com_erro([1,2], 4, [], L2).
L2 = [1, 2].
```

**Exemplo 7.11.5 ("*quicksort*")**

Algoritmo de ordenação "*quicksort*":

1. considerar um dos elementos a ordenar, o pivô
2. dividir os restantes em dois grupos, o grupo dos elementos menores e o grupo dos elementos maiores que o pivô
3. ordenar os dois grupos
4. colocar o pivô entre os grupos ordenados

# 7.11 O operador de corte

## Programa

```
quicksort([], []).

quicksort([P | R], L) :-
        parte(R, P, Menores, Maiores),
        quicksort(Menores, Menores_ord),
        quicksort(Maiores, Maiores_ord),
        junta(Menores_ord, [P | Maiores_ord], L).
```

## Utilização

```
?- quicksort([6,2,3,1,8,4,7,5],L).
L = [1, 2, 3, 4, 5, 6, 7, 8].
```

**Exemplo 7.11.6 (Junção de listas ordenadas)**

Predicado `junta_ord`

`junta_ord(L1,L2,L3)` significa que `L3` é a lista ordenada que resulta da junção das listas ordenadas `L1` e `L2`.

## Programa

```
junta_ord(L, [], L) :- !.

junta_ord([], L, L) :- !.

junta_ord([P1 | R1], [P2 | R2], [P1 | R]) :-
                              P1 < P2,
                              !,
                              junta_ord(R1, [P2 | R2], R).

junta_ord([P1 | R1], [P2 | R2], [P1 | R]) :-
                              P1 = P2,
                              !,
                              junta_ord(R1, R2, R).

junta_ord([P1 | R1], [P2 | R2], [P2 | R]) :-
                              P1 > P2,
                              !,
                              junta_ord([P1 | R1], R2, R).
```

## Utilização

```
?- junta_ord([1, 3, 5], [2, 4, 6], L).
L = [1, 2, 3, 4, 5, 6].

?- junta_ord([1, 3, 5], [2, 4, 5, 6], L).
L = [1, 2, 3, 4, 5, 6].

?- junta_ord([1, 3, 5], [2, 4, 6, 6], L).
L = [1, 2, 3, 4, 5, 6, 6].
```

**Exemplo 7.11.7 (Perigos do corte)**

`menor(V1, V2, V3)` afirma que `V3` é o menor dos elementos `V1` e `V2`.

```
menor(X, Y, X) :- X =< Y.

menor(X, Y, Y) :- X > Y.
```

Devemos introduzir um corte? Sim, por uma questão de eficiência.

```
menor_1(X, Y, X) :- X =< Y, !.

menor_1(X, Y, Y) :- X > Y.
```

**Exemplo 7.11.7 (Perigos do corte)**

Podemos eliminar o teste da 2ª regra?

```
menor_2_com_erro(X, Y, X) :- X =< Y, !.

menor_2_com_erro(_, Y, Y).
```

Não. Mesmo problema que vimos com o predicado `parte`.
A 1ª regra pode falhar porque a cabeça não unifica com objectivo:

```
?- menor_2_com_erro(5, 10, 10).
true.
```

**Exemplo 7.11.7 (Perigos do corte)**

Se quisermos eliminar o teste na 2ª regra, temos de evitar que ela falhe porque a cabeça não unifica com objectivo.

```
menor_3(X, Y, Z) :- X =< Y,
                    !,
                    Z = X.

menor_3(_, Y, Y).
```

# 7.12 O falhanço forçado

Predicado `fail/0`.

Força a geração de um nó falhado.

Útil quando usado em conjunto com o corte.

# 7.12 O falhanço forçado

**Obtenção de todas as respostas a um objectivo:**
Até agora usámos ; a seguir a cada resposta, cujo efeito é gerar um nó falhado.

Usando `fail`, temos

```
?- <objectivo>, writeln(<objectivo>), fail.
```

**Exemplo**

```
?- membro(X,[1,2,3]),writeln(membro(X,[1,2,3])),fail.
membro(1,[1,2,3])
membro(2,[1,2,3])
membro(3,[1,2,3])
false.
```

# 7.12 O falhanço forçado

**Exemplo (listas disjuntas)**

`disjuntas(L1,L2)` significa que L1 e L2 são disjuntas, isto é, não têm elementos em comum.

**Algoritmo:**

$$disjuntas(L1, L2) = \begin{cases} verdadeiro & \text{se L1} = [\,] \\ falso & \text{se primeiro(L1)} \in \text{L2} \\ disjuntas(resto(L1), L2) & \text{senão} \end{cases}$$

disjuntas([2,3,7], [1,3,5]) = disjuntas([3,7], [1,3,5]) = falso

disjuntas([2,3,7], [5]) =
disjuntas([3,7], [5]) =
disjuntas([7], [5]) =
disjuntas([ ], [5]) =
verdadeiro

# 7.12 O falhanço forçado

## Definição do predicado `disjuntas/2`

```
disjuntas([], _) :- !.
disjuntas(_, []) :- !.
disjuntas([P1 | _], L2) :-  membro(P1,L2), !, fail.
disjuntas([_ | R1], L2) :- disjuntas(R1, L2).
```

## Utilização

```
?- disjuntas([2,3,7], [1,3,5]).
false.

?- disjuntas([2,3,7], [5]).
true.
```

# 7.13 A negação em PROLOG

Combinação do corte com o falhanço forçado permite definir a *negação por falhanço*.

Diferente da negação lógica. Corresponde à negação usada na hipótese do mundo fechado.

**Hipótese do mundo fechado:**

Assume que tudo o que não é derivável a partir de um programa é falso.

# 7.13 A negação em PROLOG

`not(literal)` **não** significa $\neg$ `literal`, mas antes que `literal` não é derivável ($\nvdash$) a partir do programa.

Por esta razão a negação por falhanço é representada por `\+`.

Predicado `\+` pode ser definido do seguinte modo:

```
\+(Lit) :- Lit, !, fail.
\+(Lit).
```

Negação por falhanço pode ser usada na representação de regras com excepções, isto é regras do tipo "Normalmente ...".

**Exemplo 7.13.1 (Entidades voadoras):**

Consideremos as regras:

- "Os pinguins são aves".
- "Normalmente as aves voam, a não ser que sejam pinguins".

A 1ª é uma regra universal, não tem excepções:
se soubermos que um `X` é um pinguim, podemos concluir que é uma ave.

A 2ª é uma regra com excepções:
Se soubermos que um `X` é uma ave, podemos concluir que voa, desde que não saibamos que é um pinguim.

# 7.13 A negação em PROLOG

**Exemplo 7.13.1 (Entidades voadoras):**

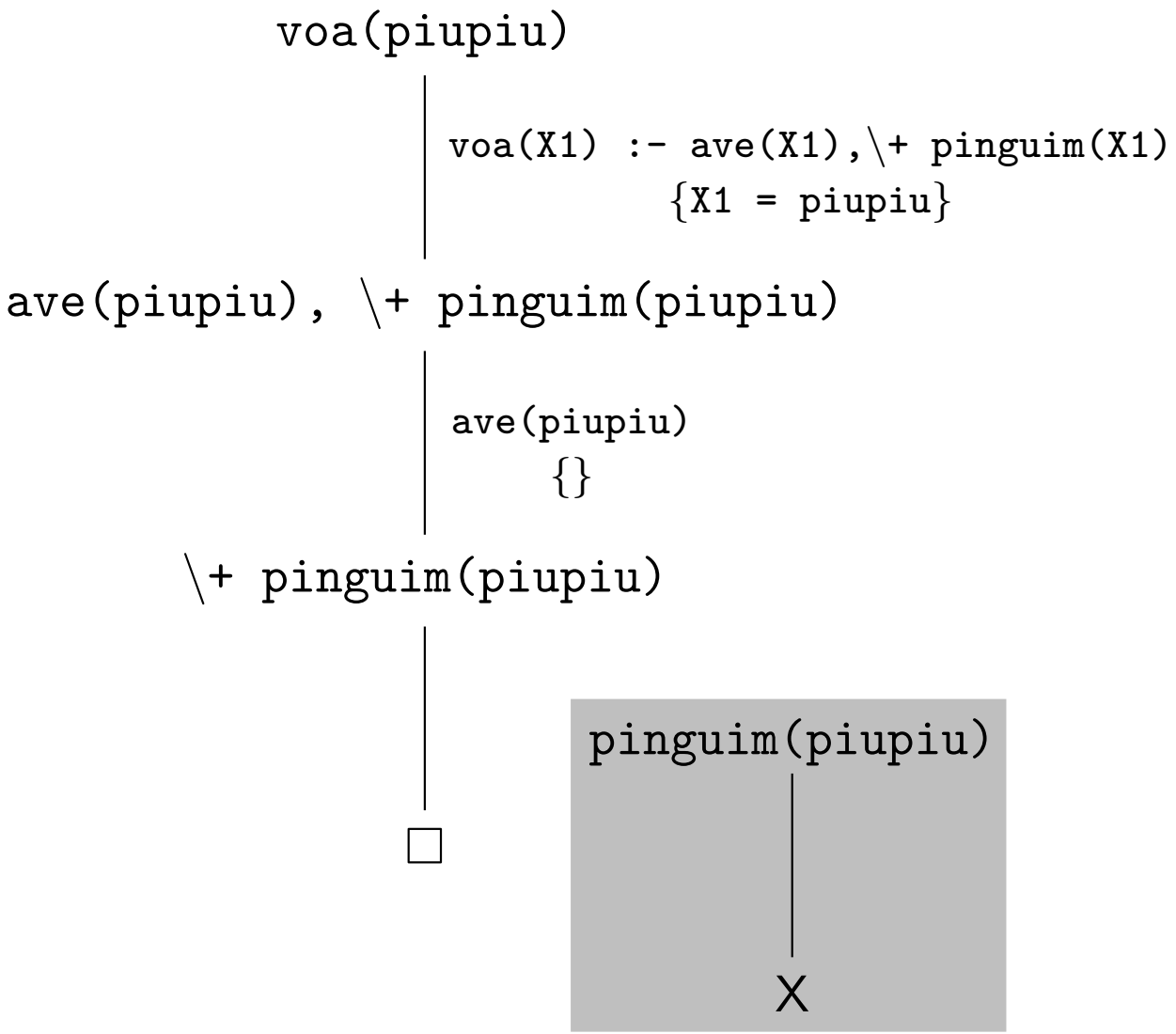
```
pinguim(gelido).

ave(piupiu).
ave(X) :- pinguim(X).

voa(X) :-  ave(X), \+ pinguim(X).
```

# 7.13 A negação em PROLOG

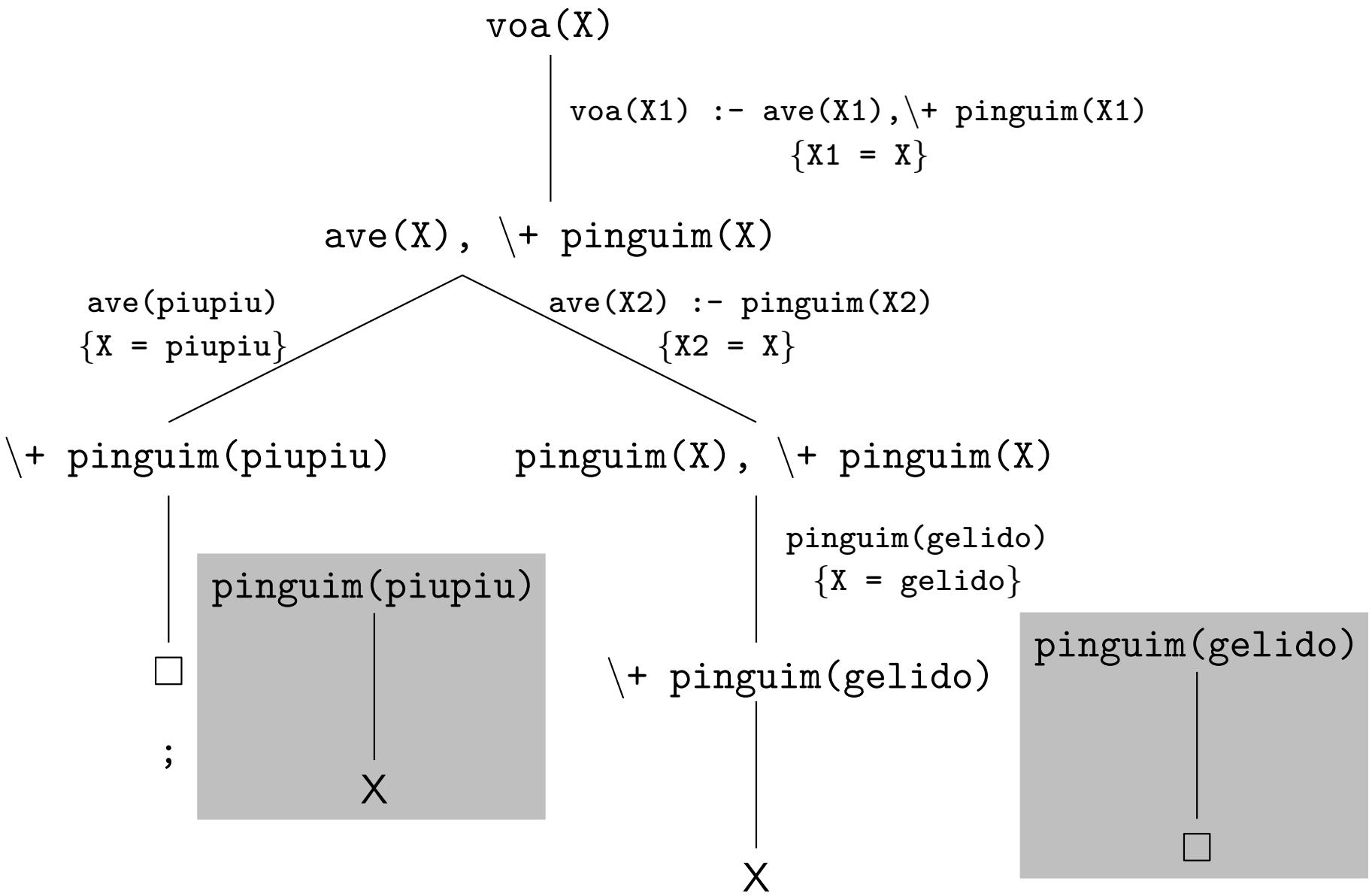
**Utilização:**

```
?- voa(gelido).
false.

?- voa(piupiu).
true .

?- voa(X).
X = piupiu ;
false.

?- \+ voa(X).
false.

?- \+ pinguim(X).
false.
```

# 7.13 A negação em PROLOG



# 7.13 A negação em PROLOG



# 7.13 A negação em PROLOG

A negação por falhanço não funciona correctamente para objectivos não chãos, como mostram os dois últimos exemplos.

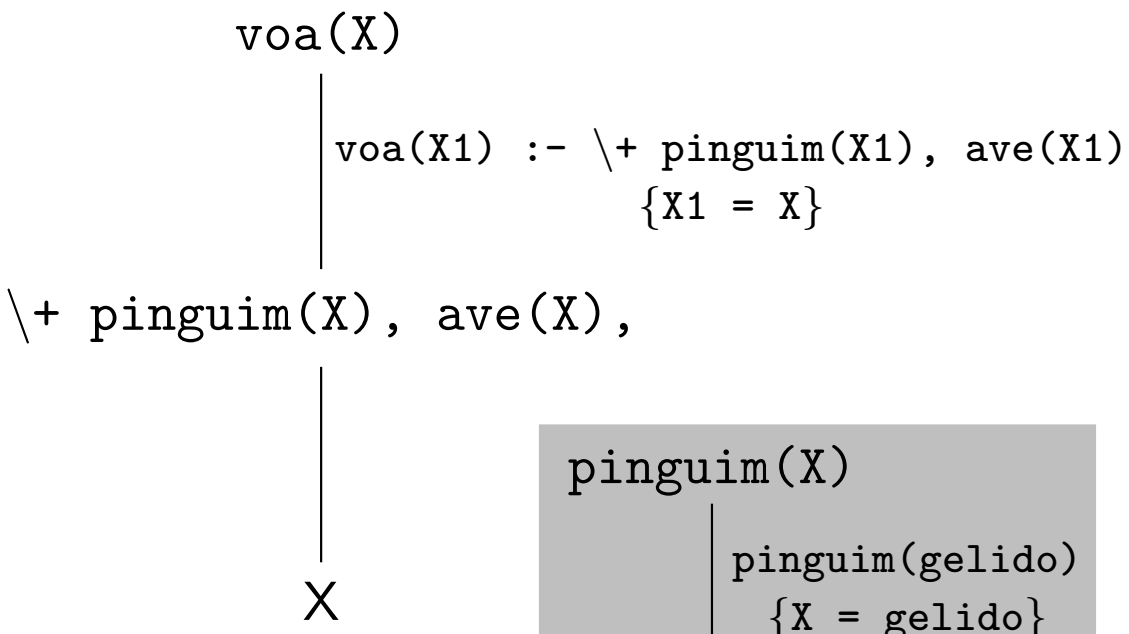
Por essa razão, se mudássemos a regra para:

```
voa(X) :- \+ pinguim(X), ave(X).
```

Obteríamos:

```
?- voa(X).
false.
```

# 7.13 A negação em PROLOG



# 7.13 A negação em PROLOG

**Exemplo (listas disjuntas - versão2)**

`disjuntas(L1,L2)` significa que L1 e L2 são disjuntas, isto é, **NÃO** têm elementos em comum.

```
disjuntas(L1,L2) :-  \+ (membro(E,L1),membro(E,L2)).
```

# 7.15 Execução forçada

Numa *regra*

$$\langle \text{literal} \rangle \ :- \ \langle \text{literais} \rangle .$$

é possível o literal cabeça ser "*nada*". A regra

$$:- \ \langle \text{literais} \rangle .$$

pode ser interpretada como significando
"para provar "*nada*", prove os literais a seguir ao símbolo :- ",
levando à execução forçada dos literais que surgem após o símbolo ":-".

# 7.15 Execução forçada

Utilização num programa permite a execução de código, sem ser necessário introduzir nenhum objectivo.

**Exemplo:**

```
:- writeln('Este programa ....').

:- [fich1], [fich2].
```

Quando o ficheiro contendo estas regras é carregado no PROLOG:

- É escrita a mensagem `Este programa .....`
- São carregados os ficheiros `fich1` e `fich2`.