

Aula 9: Cortes e Negação

- Teoria
 - Explicar como controlar o comportamento de retrocesso do Prolog com a ajuda do predicado de corte
 - Introduzir negação
 - Explicar como o corte pode ser empacotado em uma forma mais estruturada: a negação como falha

O Corte

- O retrocesso (*backtracking*) é um traço característico do Prolog
- Mas o retrocesso pode levar a ineficiência:
 - Prolog pode gastar tempo explorando possibilidades que levam a lugar nenhum
 - Seria bom ter algum controle
- O predicado de corte **!/0** oferece um modo de controlar o retrocesso

Exemplo de corte

- O corte é um predicado Prolog, assim podemos incluí-lo no corpo de uma regra:
 - Exemplo:

`p(X):- b(X), c(X), !, d(X), e(X).`

- O corte é um objetivo que sempre é bem sucedido
- Ele restringe o Prolog às escolhas feitas desde que o objetivo pai foi chamado.

Explicando o corte

- Para explicar o corte, nós
 - Olharemos um trecho de programa Prolog sem corte e veremos o que ele faz em termos de retrocesso.
 - Adicionaremos cortes a este trecho Prolog
 - Examinaremos a mesma parte do código já com os cortes adicionados e olharemos como os cortes afetam o retrocesso

Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

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

Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

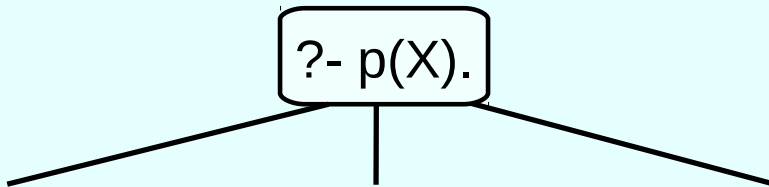
```
?- p(X).
```

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

Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

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



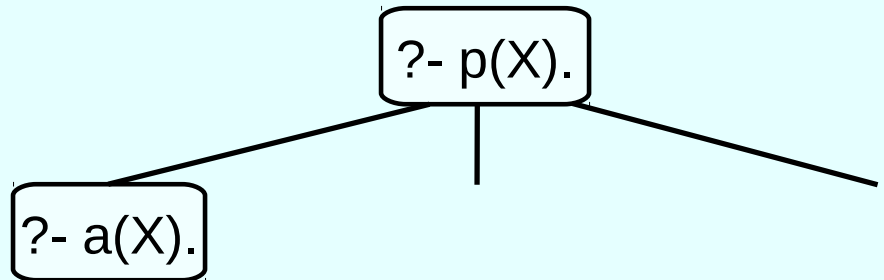
```
graph TD; A[?- p(X).] --- B[ ]; A --- C[ ]; A --- D[ ]
```

?- p(X).

Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

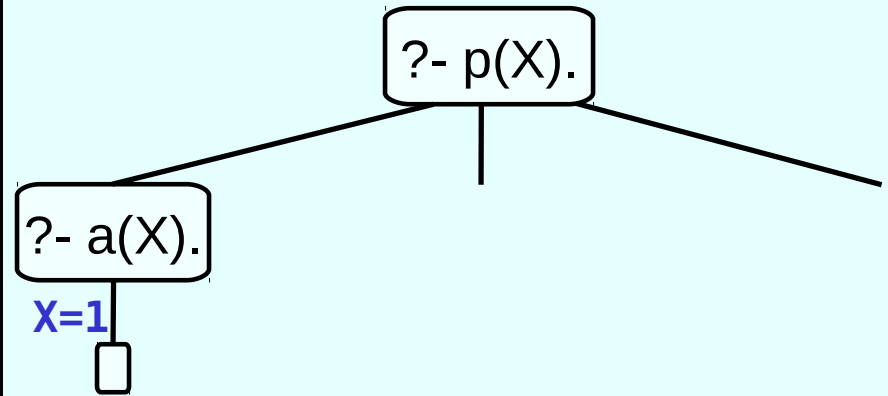
```
?- p(X).
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

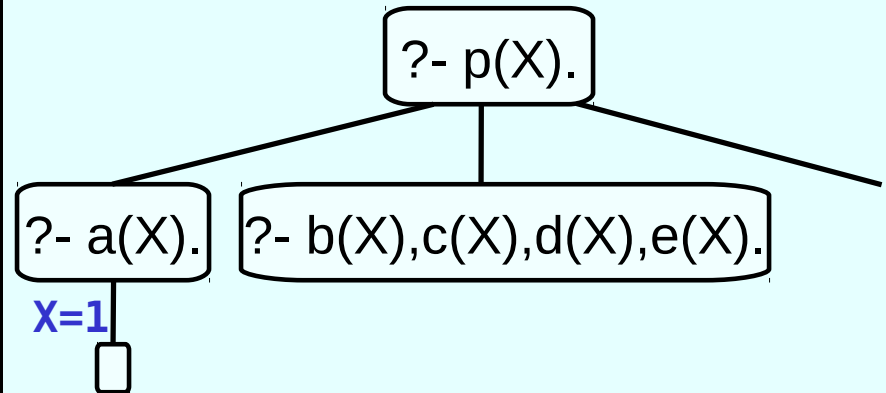
```
?- p(X).  
X=1
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

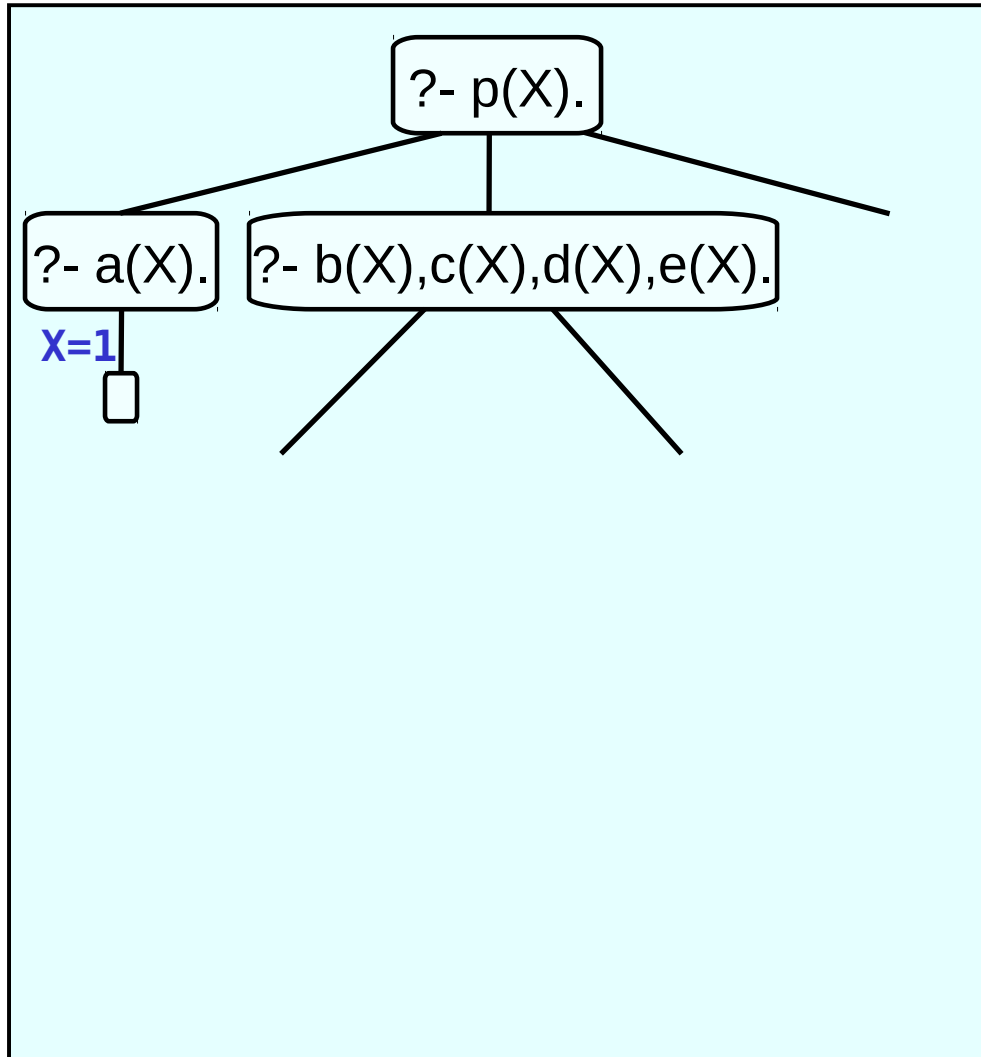
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

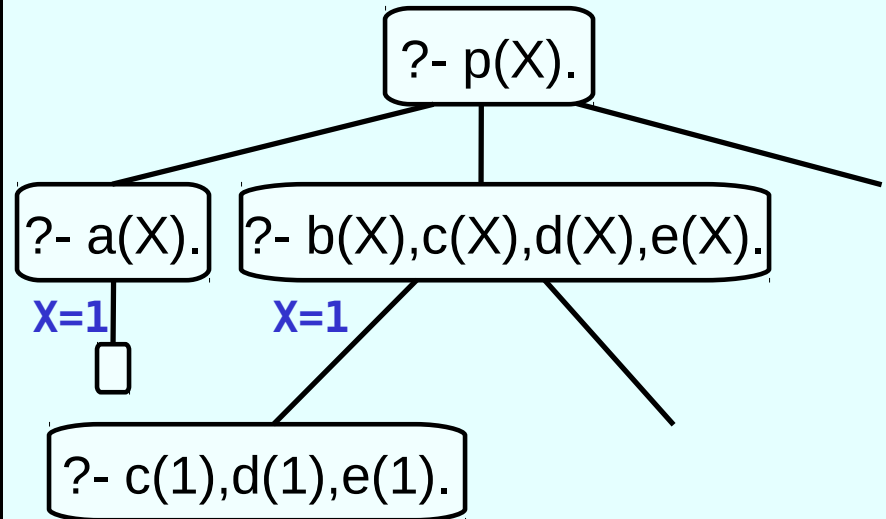
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

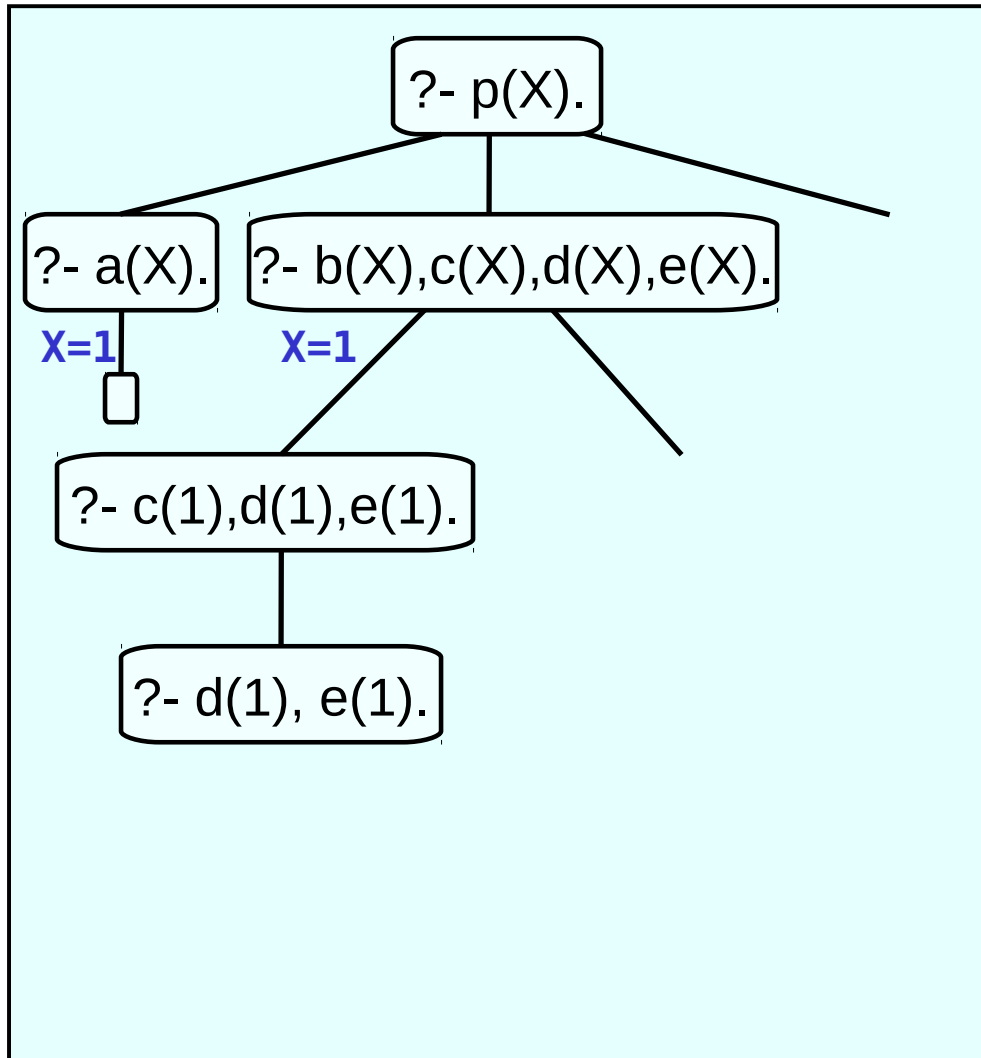
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

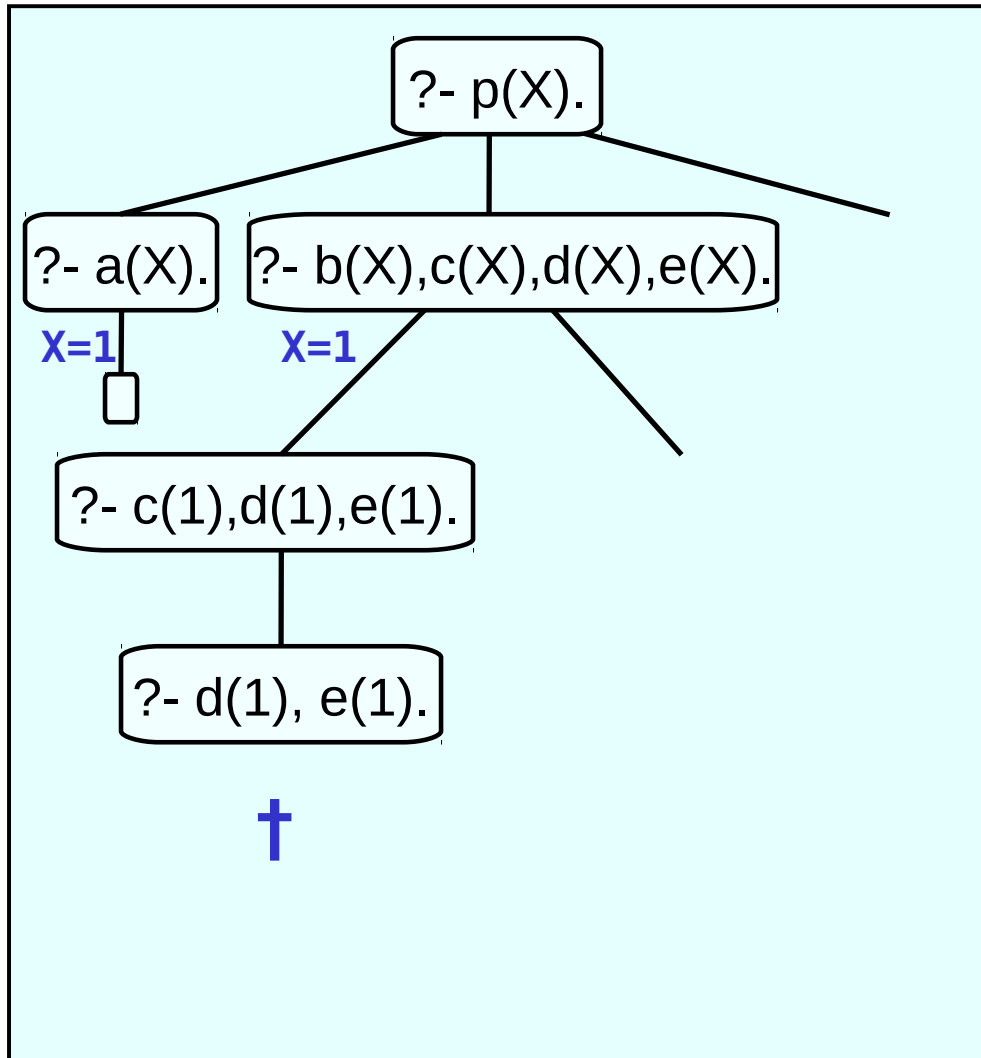
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

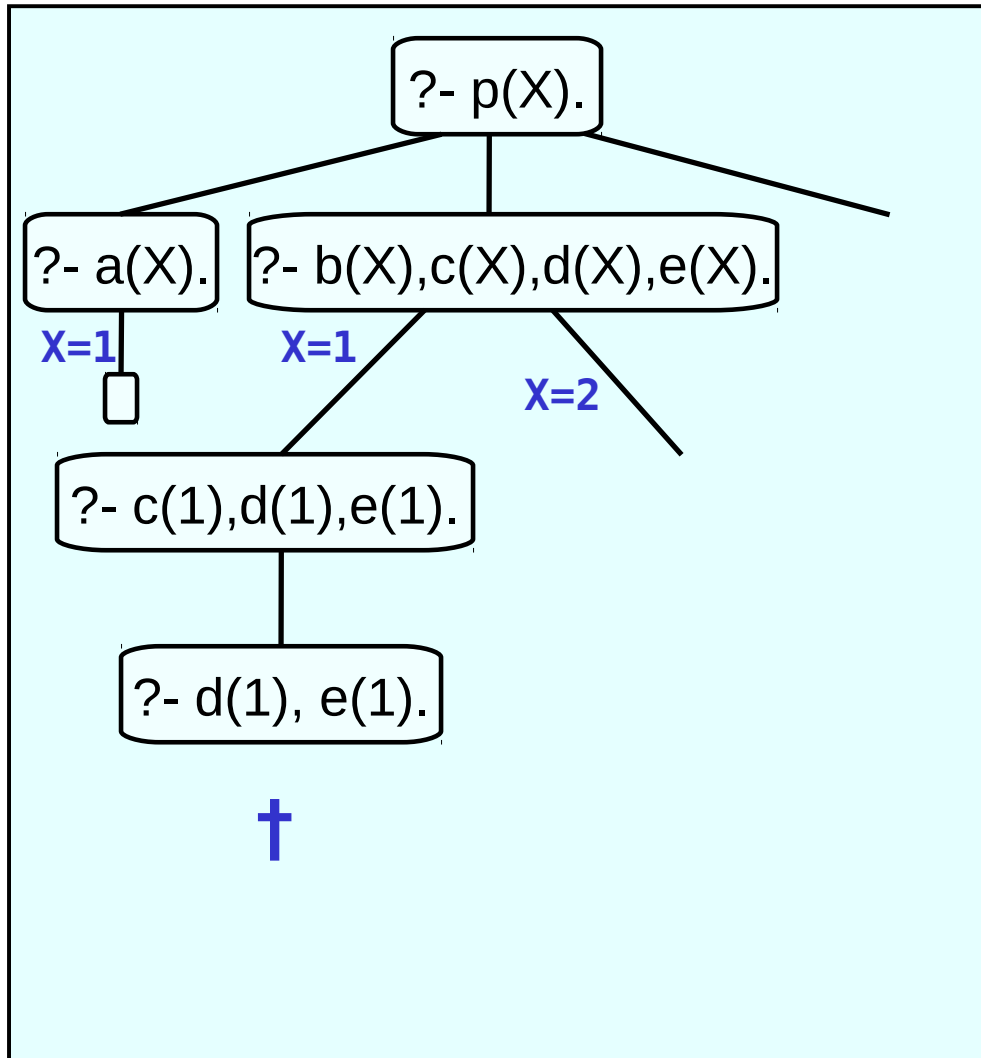
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

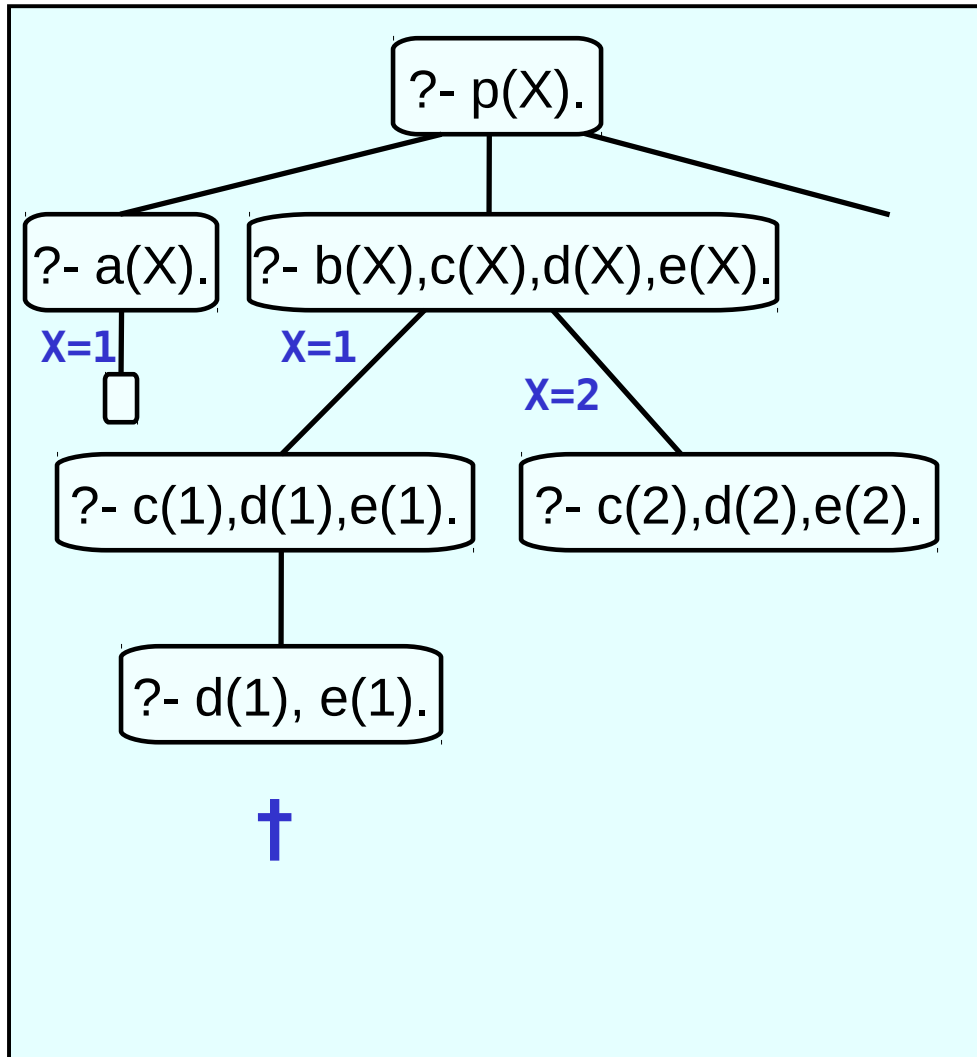
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

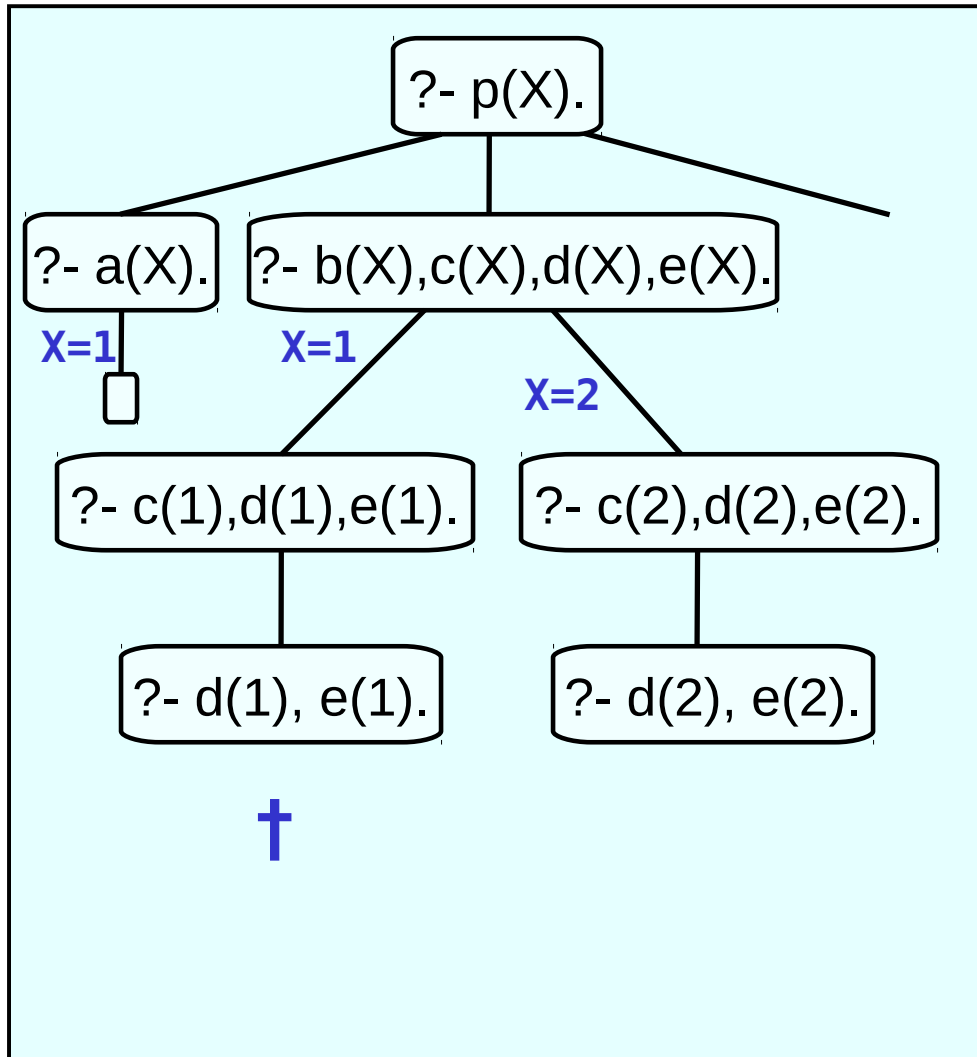
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

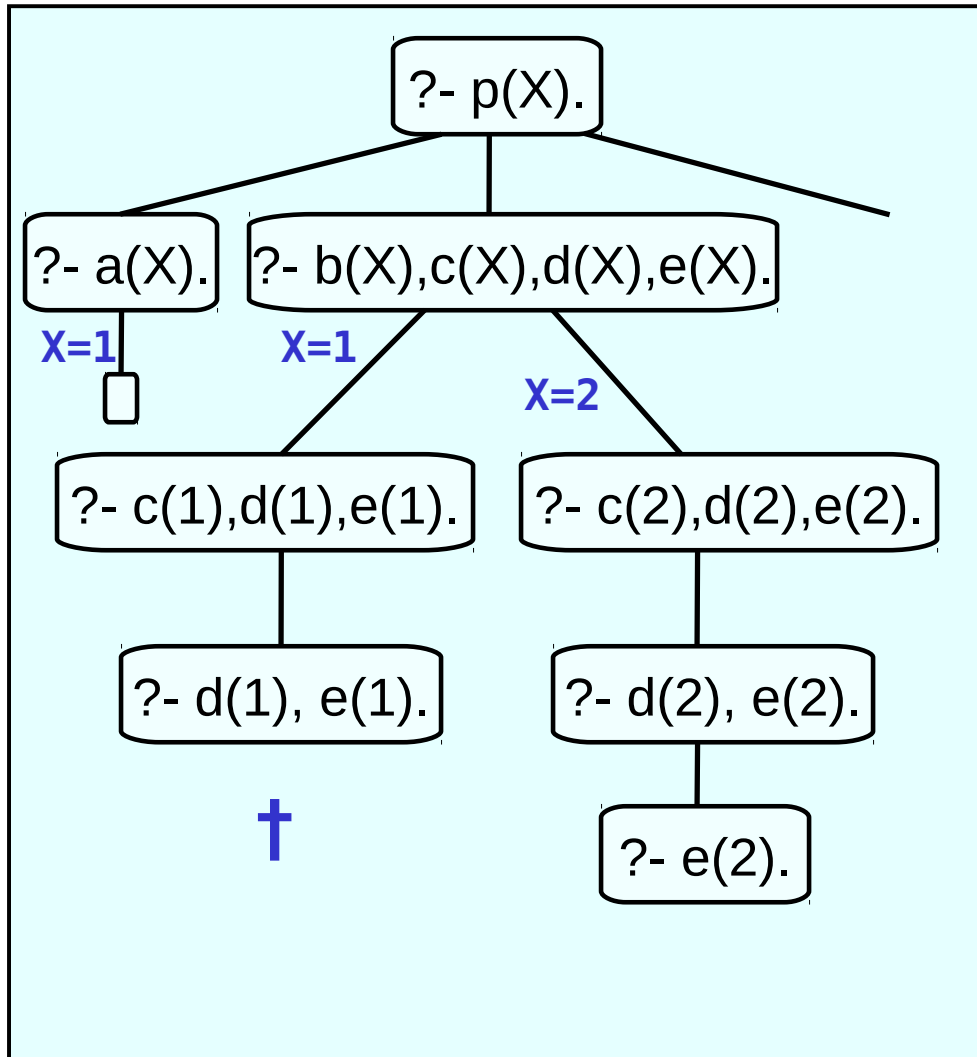
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

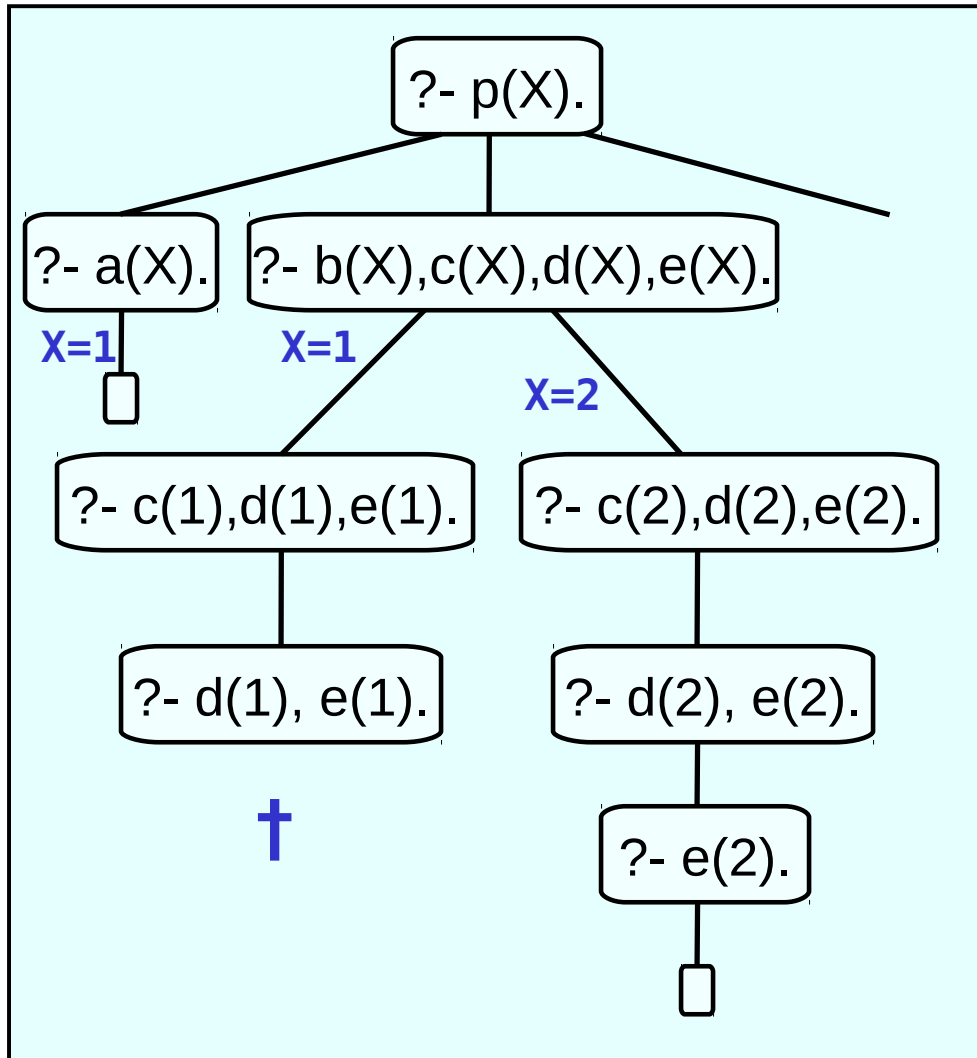
```
?- p(X).  
X=1;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

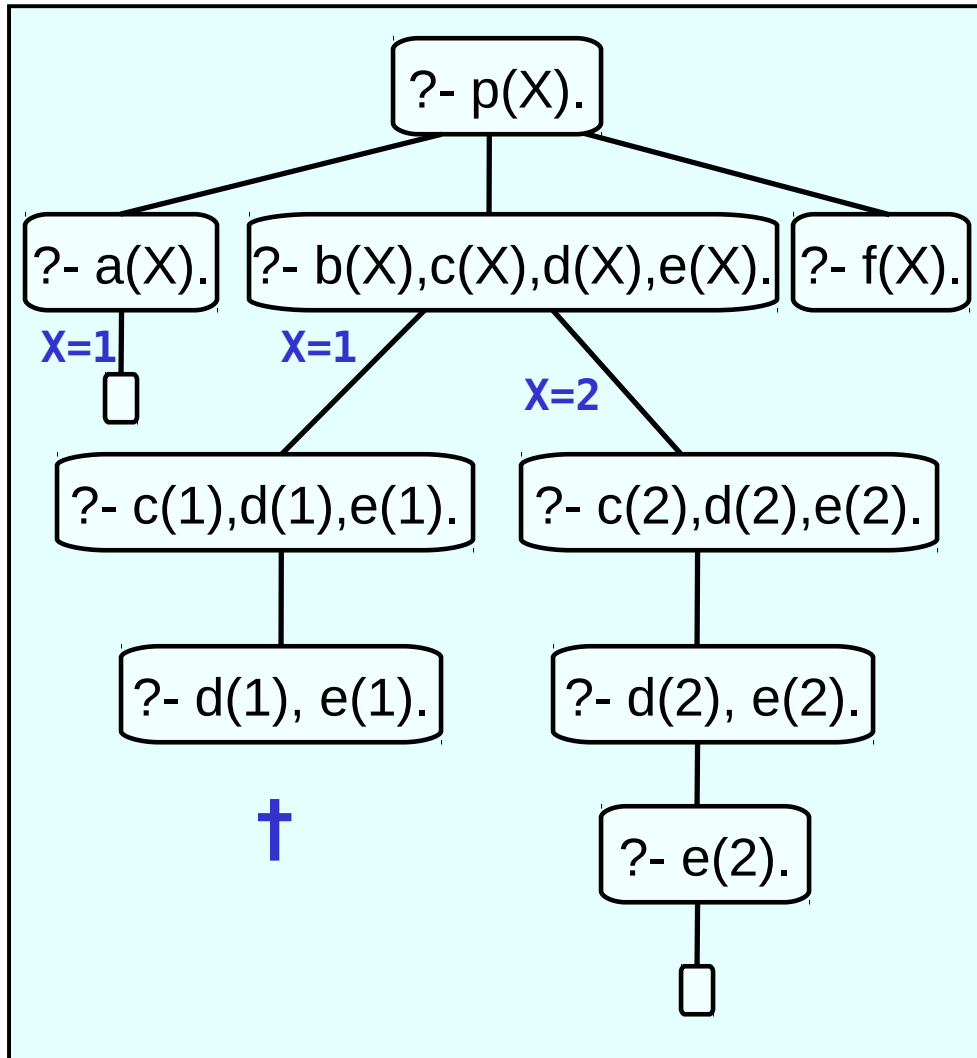
```
?- p(X).  
X=1;  
X=2
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

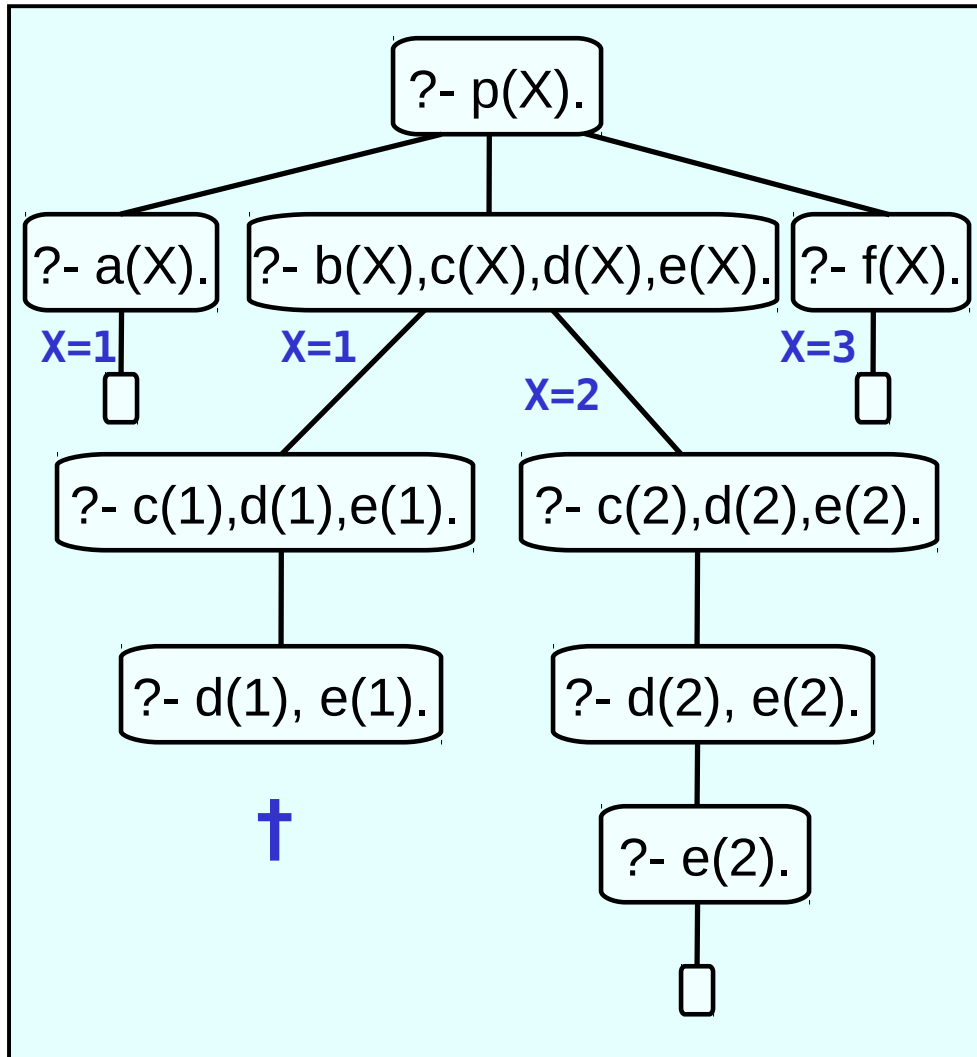
```
?- p(X).  
X=1;  
X=2;
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

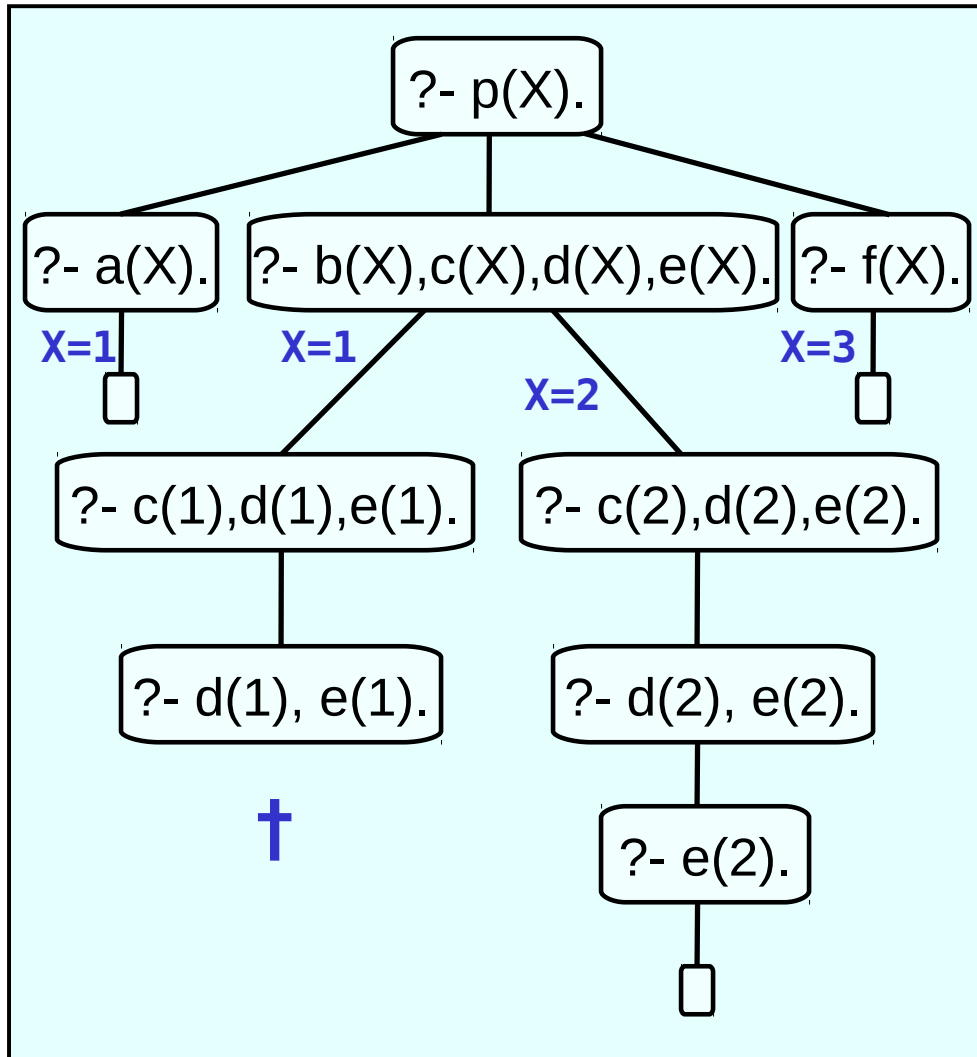
```
?- p(X).  
X=1;  
X=2;  
X=3
```



Exemplo: código sem corte

```
p(X):- a(X).  
p(X):- b(X), c(X), d(X), e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

```
?- p(X).  
X=1;  
X=2;  
X=3;  
no
```



Adicionando um corte

- Suponha que adicionemos um corte na segunda cláusula:

```
p(X):- b(X), c(X), !, d(X), e(X).
```

- Se nós repetirmos a mesma consulta anterior, obteremos a seguinte resposta:

```
?- p(X).  
X=1;  
no
```

Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

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


Exemplo: corte

p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).

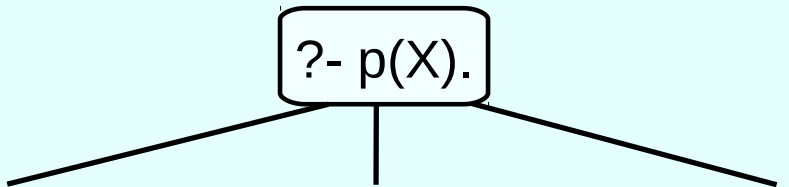
?- p(X).

?- p(X).

Exemplo: corte

p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).

?- p(X).



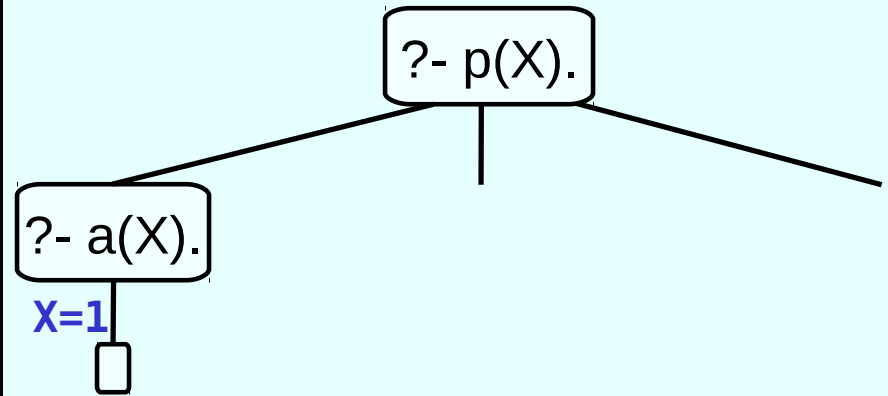
?- p(X).

The diagram shows a rectangular box containing the text '?- p(X)'. From the bottom of this box, three lines extend downwards and outwards, representing the branches of a search tree or a call stack in a Prolog interpreter.

Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

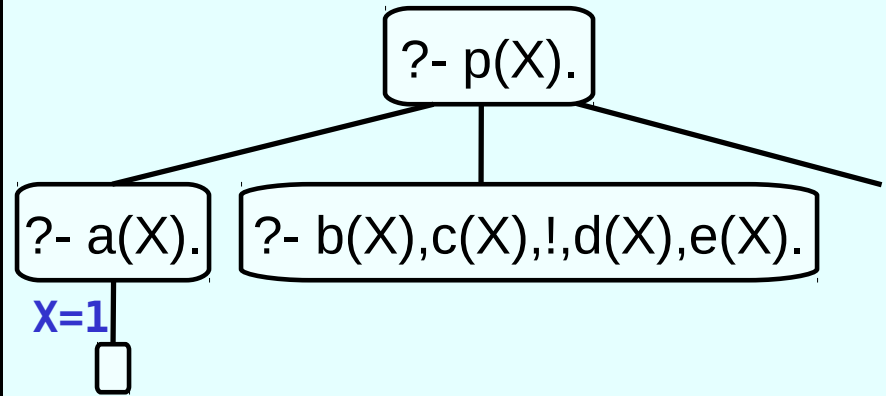
```
?- p(X).  
X=1
```



Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

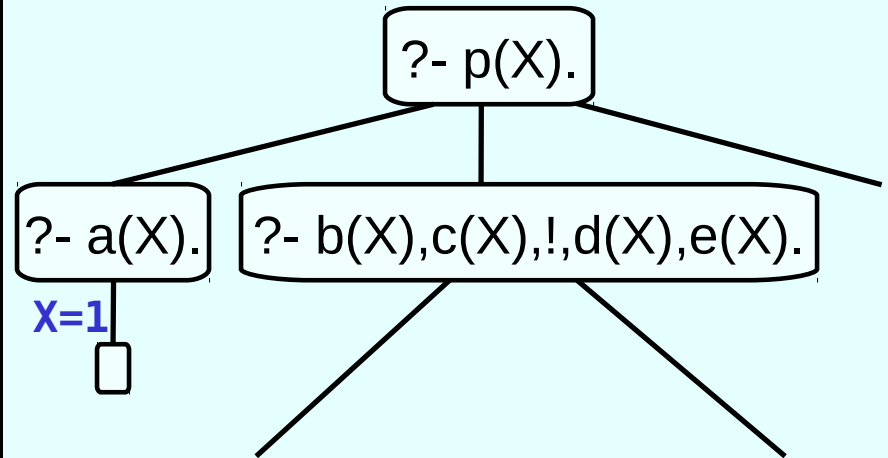
```
?- p(X).  
X=1;
```



Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

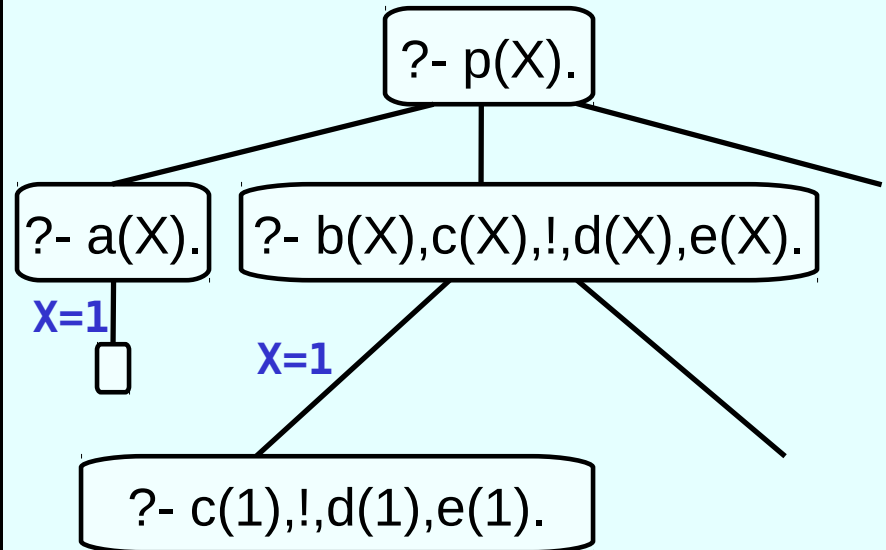
```
?- p(X).  
X=1;
```



Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

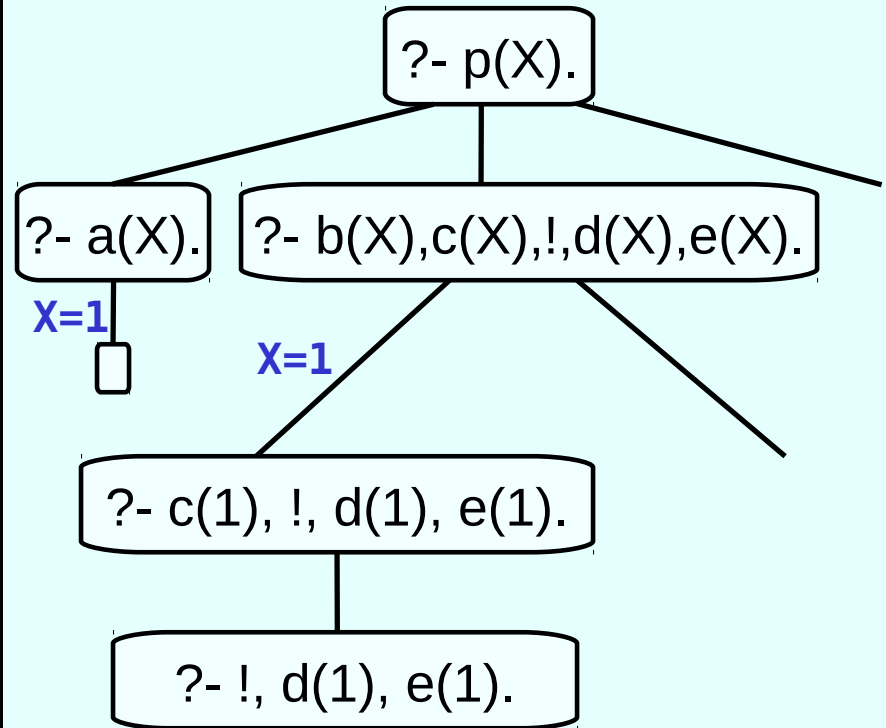
```
?- p(X).  
X=1;
```



Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

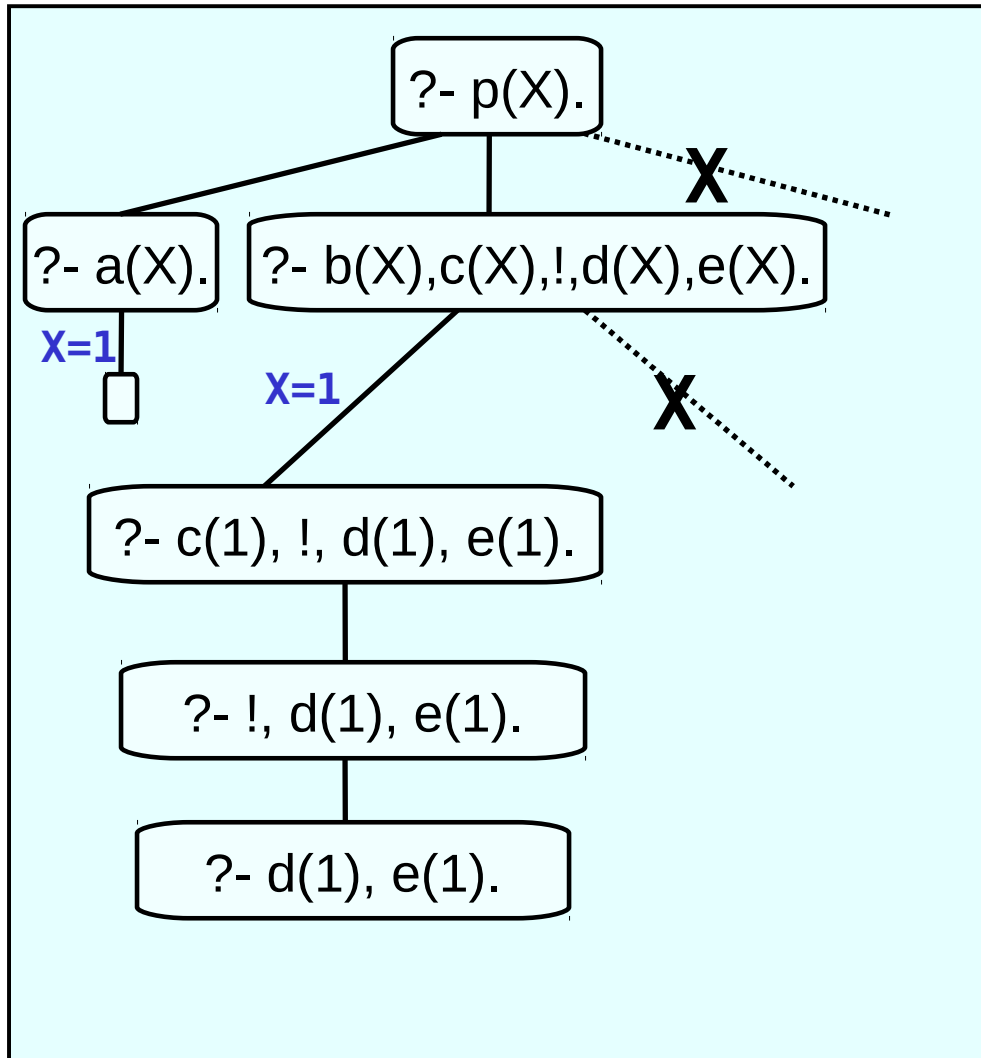
```
?- p(X).  
X=1;
```



Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

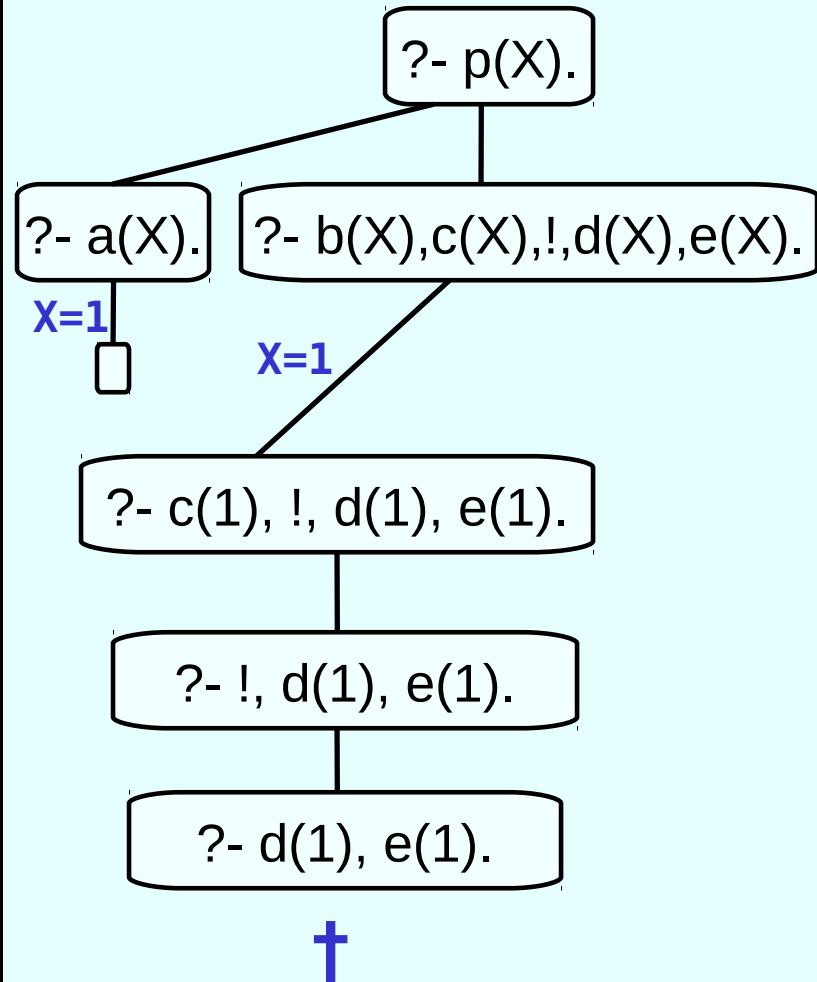
```
?- p(X).  
X=1;
```



Exemplo: corte

```
p(X):- a(X).  
p(X):- b(X),c(X),!,d(X),e(X).  
p(X):- f(X).  
a(1).  
b(1).  b(2).  
c(1).  c(2).  
d(2).  
e(2).  
f(3).
```

```
?- p(X).  
X=1;  
no
```



O que o corte faz?

- O corte somente restringe-nos às escolhas já feitas desde que o objetivo pai foi unificado com o lado esquerdo da cláusula contendo o corte.
- Por exemplo, em uma regra da forma

$q:- p_1, \dots, p_n, !, r_1, \dots, r_n.$

Quando alcançarmos o corte ele nos restringe:

- a esta cláusula particular de q
- às escolhas feitas por p_1, \dots, p_n
- Mas, NÃO às escolhas feitas por r_1, \dots, r_n

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.  
max(X,Y,X):- X > Y.
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,3).
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X <= Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,3).  
true
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X <= Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,3).
```

```
true
```

```
?- max(7,3,7).
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,3).
```

```
true
```

```
?- max(7,3,7).
```

```
true
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,2).
```


Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,2).  
false
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X <= Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,2).
```

```
false
```

```
?- max(2,3,5).
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.  
max(X,Y,X):- X > Y.
```

```
?- max(2,3,2).  
false
```

```
?- max(2,3,5).  
false
```

O predicado max/3

- Qual é o problema?
- Existe uma ineficiência em potencial
 - Suponha que ele é chamado com ?-max(3,4,Y).
 - Ele corretamente unificará Y com 4
 - Mas, quando lhe é solicitado mais soluções, ele tentará satisfazer a segunda cláusula. E isto é completamente sem sentido!

```
max(X,Y,Y):- X =< Y.  
max(X,Y,X):- X > Y.
```

max/3 com corte

- Com a ajuda do corte isto é fácil de consertar

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

- Note como isto funciona:
 - Se $X \leq Y$ sucede, o corte nos restringe a esta escolha e a segunda cláusula de max/3 não é considerada
 - Se $X \leq Y$ falha, Prolog segue para a segunda cláusula

Cortes verdes

- Cortes que não alteram o significado de um predicado são chamados de **cortes verdes**
- O corte em $\max/3$ é um exemplo de corte verde:
 - O novo código produz exatamente as mesmas respostas que a versão antiga.
 - Mas é mais eficiente!

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

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

- Quão bom é isto?

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

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

- Quão bom é isto?

```
?- max(200,300,X).
```


Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

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

- Quão bom é isto?
 - ok

```
?- max(200,300,X).  
X=300
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

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

- Quão bom é isto?

```
?- max(400,300,X).
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

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

- Quão bom é isto?
 - ok

```
?- max(400,300,X).  
X=400
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

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

- Quão bom é isto?

```
?- max(200,300,200).
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

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

- Quão bom é isto?
 - ôpa!

```
?- max(200,300,200).  
true
```

max/3 revisado com corte

- Unificação após cruzar o corte

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

- Isto de fato funciona

```
?- max(200,300,200).
```

max/3 revisado com corte

- Unificação após cruzar o corte

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

- Isto de fato funciona

```
?- max(200,300,200).  
false
```

Cortes Vermelhos

- Cortes que alteram o significado de um predicado são chamados de cortes vermelhos
- O corte no max/3 revisado é um exemplo de corte vermelho:
 - Se retirarmos o corte, nós não obteremos um programa equivalente ao original
- Programas contendo cortes vermelhos
 - Não são completamente declarativos
 - Podem ser difíceis de ler
 - Podem levar a erros sutis de programação

Um outro predicado pré-construído: **fail/0**

- Como o nome sugere, esta é uma meta que falhará imediatamente quando o Prolog tentar prová-la.
- Isto pode não parecer muito útil
- Mas, lembre-se: quando Prolog falha, ele tenta retroceder

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com **fail** permite codificar exceções

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com **fail** permite codificar exceções

```
?- aprecia(vicente,a).
```

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com **fail** permite codificar exceções

```
?- aprecia(vicente,a).  
true
```

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com fail permite codificar exceções

```
?- aprecia(vicente,b).
```

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com fail permite codificar exceções

```
?- aprecia(vicente,b).  
false
```

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com fail permite codificar exceções

```
?- aprecia(vicente,c).
```

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com fail permite codificar exceções

```
?- aprecia(vicente,c).  
true
```


Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com fail permite codificar exceções

```
?- aprecia(vicente,d).
```

Vicente e hambúrgueres

```
aprecia(vicente,X):- bigKahunaBurger(X), !, fail.  
aprecia(vicente,X):- hamburguer(X).
```

```
hamburguer(X):- bigMac(X).  
hamburguer(X):- bigKahunaBurger(X).  
hamburguer(X):- whopper(X).
```

```
bigMac(a).  
bigKahunaBurger(b).  
bigMac(c).  
whopper(d).
```

- A combinação de corte com fail permite codificar exceções

```
?- aprecia(vicente,d).  
true
```

Negação como falha

- A combinação de corte com **fail** parece estar a nos oferecer alguma forma de negação
- Isto é chamado de **negação como falha**, e é definido como segue:

```
neg(Objetivo):- Objetivo, !, fail.  
neg(Objetivo).
```

Vicente e hambúrgueres revisitado

```
aprecia(vicente,X):- hamburguer(X),  
                    neg(bigKahunaBurger(X)).
```

```
hamburguer(X):- bigMac(X).
```

```
hamburguer(X):- bigKahunaBurger(X).
```

```
hamburguer(X):- whopper(X).
```

```
bigMac(a).
```

```
bigKahunaBurger(b).
```

```
bigMac(c).
```

```
whopper(d).
```

Vicente e hambúrgueres revisitado

```
aprecia(vicente,X):- hamburguer(X),  
                    neg(bigKahunaBurger(X)).
```

```
hamburguer(X):- bigMac(X).
```

```
hamburguer(X):- bigKahunaBurger(X).
```

```
hamburguer(X):- whopper(X).
```

```
bigMac(a).
```

```
bigKahunaBurger(b).
```

```
bigMac(c).
```

```
whopper(d).
```

```
?- aprecia(vicente,X).
```

Vicente e hambúrgueres revisitado

```
aprecia(vicente,X):- hamburguer(X),  
                    neg(bigKahunaBurger(X)).
```

```
hamburguer(X):- bigMac(X).
```

```
hamburguer(X):- bigKahunaBurger(X).
```

```
hamburguer(X):- whopper(X).
```

```
bigMac(a).
```

```
bigKahunaBurger(b).
```

```
bigMac(c).
```

```
whopper(d).
```

```
?- aprecia(vicente,X).  
X=a;
```

Vicente e hambúrgueres revisitado

```
aprecia(vicente,X):- hamburguer(X),  
                    neg(bigKahunaBurger(X)).
```

```
hamburguer(X):- bigMac(X).
```

```
hamburguer(X):- bigKahunaBurger(X).
```

```
hamburguer(X):- whopper(X).
```

```
bigMac(a).
```

```
bigKahunaBurger(b).
```

```
bigMac(c).
```

```
whopper(d).
```

```
?- aprecia(vicente,X).
```

```
X=a;
```

```
X=c;
```

Vicente e hambúrgueres revisitado

```
aprecia(vicente,X):- hamburguer(X),  
                    neg(bigKahunaBurger(X)).
```

```
hamburguer(X):- bigMac(X).
```

```
hamburguer(X):- bigKahunaBurger(X).
```

```
hamburguer(X):- whopper(X).
```

```
bigMac(a).
```

```
bigKahunaBurger(b).
```

```
bigMac(c).
```

```
whopper(d).
```

```
?- aprecia(vicente,X).
```

```
X=a;
```

```
X=c;
```

```
X=d
```


Mais um predicado pré-construído: \+

- Como a negação como falha é usada com frequência, não há necessidade de defini-la
- No Prolog padrão, o operador prefixo \+ significa a negação como falha
- Assim, poderemos definir as preferências de Vicente como segue:

```
aprecia(vicente,X):- hamburger(X),  
                      \+ bigKahunaBurger(X).
```

```
?- aprecia(vicente,X).
```

Mais um predicado pré-construído: **\+**

- Como a negação como falha é usada com frequência, não há necessidade de defini-la
- No Prolog padrão, o operador prefixo **\+** significa a negação como falha
- Assim, poderemos definir as preferências de Vicente como segue:

```
aprecia(vicente,X):- hamburguer(X),  
                    \+ bigKahunaBurger(X).
```

```
?- aprecia(vicente,X).  
X=a;
```

Mais um predicado pré-construído: **\+**

- Como a negação como falha é usada com frequência, não há necessidade de defini-la
- No Prolog padrão, o operador prefixo **\+** significa a negação como falha
- Assim, poderemos definir as preferências de Vicente como segue:

```
aprecia(vicente,X):- hamburger(X),  
                      \+ bigKahunaBurger(X).
```

```
?- aprecia(vicente,X).  
X=a;  
X=c;
```

Mais um predicado pré-construído: \+

- Como a negação como falha é usada com frequência, não há necessidade de defini-la
- No Prolog padrão, o operador prefixo \+ significa a negação como falha
- Assim, poderemos definir as preferências de Vicente como segue:

```
aprecia(vicente,X):- hamburger(X),  
                      \+ bigKahunaBurger(X).
```

```
?- aprecia(vicente,X).  
X=a  
X=c  
X=d
```

Negação como falha e lógica

- A negação como falha não é a negação lógica
- Alterando a ordem das metas no programa Vicente e hambúrgueres produz um comportamento diferente:

```
aprecia(vicente,X):- \+ bigKahunaBurger(X),  
                    hamburger(X).
```

```
?- aprecia(vicente,X).  
false
```

Próxima aula

- Manipulação de banco de dados e coleta de soluções
 - Discussão de manipulação de dados em Prolog
 - Introdução de predicados pré-construídos que nos permitem coletar todas as soluções em uma única lista