Prolog
(Termos, Cuts e Negação)

Parcialmente adaptado de

http://www.learnprolognow.org/

Comparação de termos: ==/2

- O Prolog contém um predicado para comparar termos
- É o predicado da igualdade ==/2
- O predicado da igualdade ==/2 não instancia variáveis, tem um comportamento diferente do =/2

Comparação de termos: ==/2

- O Prolog contém um predicado para comparar termos
- É o predicado da igualdade ==/2
- O predicado da igualdade ==/2 não instancia variáveis, tem um comportamento diferente do =/2

```
?- a==a.
yes

?- a==b.
no

?- a=='a'.
yes

?- a==X.
X = _443
no
```

Comparação de variáveis

- Duas variáveis diferentes **não-instanciadas** não são termos iguais
- Variáveis **instanciadas** com um termo *T* são iguais a *T*

Comparação de variáveis

- Duas variáveis diferentes **não-instanciadas** não são termos iguais
- Variáveis **instanciadas** com um termo *T* são iguais a *T*

```
?- X==X.
X = _443
yes

?- Y==X.
Y = _442
X = _443
no

?- a=U, a==U.
U = _443
yes
```

Comparação de termos: \==/2

- O predicado \==/2 sucede exactamente nos casos em que ==/2 falha
- Ou seja, sucede sempre que dois termos são **diferentes**, e falha caso contrário

Comparação de termos: \==/2

- O predicado \==/2 sucede exactamente nos casos em que ==/2 falha
- Ou seja, sucede sempre que dois termos são **diferentes**, e falha caso contrário

```
?- a \== a.
no

?- a \== b.
yes

?- a \== 'a'.
no

?- a \== X.
X = _443
yes
```

Termos aritméticos

- +, -, <, >, etc são functors e expressões como 2+3 são termos complexos
- O termo 2+3 é igual ao termo +(2,3)

Termos aritméticos

- +, -, <, >, etc são functors e expressões como 2+3 são termos complexos
- O termo 2+3 é igual ao termo +(2,3)

```
?- 2+3 == +(2,3).
yes

?-  -(2,3) == 2-3.
yes

?-  (4<2) == <(4,2).
yes
```

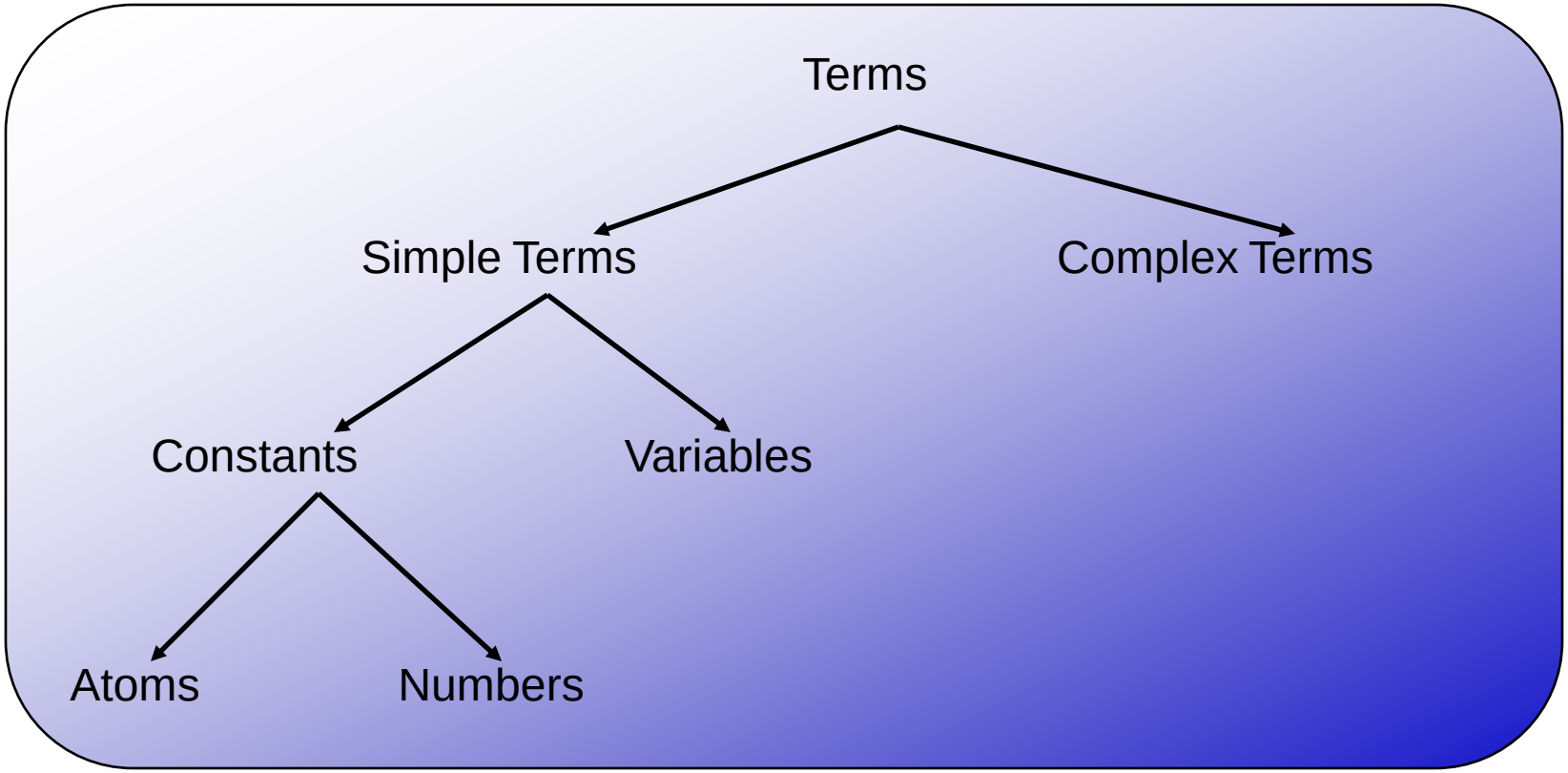
Sumário predicados de comparação

=	Unificação
\=	Negação da unificação
==	Igualdade
\==	Negação da igualdade
=:=	Igualdade aritmética
=\=	Negação da igualdade aritmética

Examinar os termos

- Existem predicados Prolog especializados em examinar termos
  - Predicados que identificam o tipo do termo
  - Predicados fornecem informação sobre a estrutura interna dos termos

Tipo de termos



Verificar o tipo de um termo

atom/1	*O argumento é um átomo?*
integer/1	*… um inteiro?*
float/1	*… um número real?*
number/1	*… um inteiro ou número real?*
atomic/1	*… uma constante?*
var/1	*… uma variável não-instanciada?*
nonvar/1	*… uma variável instanciada ou outro termo que não seja uma variável não-instanciada?*

Verificar o tipo: atom/1

?- atom(a).

yes

?- atom(7).

no

?- atom(X).

no

Verificar o tipo: atom/1

?- X=a, atom(X).

X = a

yes

?- atom(X), X=a.

no

Verificar o tipo: atomic/1

?- atomic(mia).

yes

?- atomic(5).

yes

?- atomic(loves(vincent,mia)).

no

Verificar o tipo: var/1

?- var(mia).

no

?- var(X).

yes

?- X=5, var(X).

no

Verificar o tipo: nonvar/1

?- nonvar(X).

no

?- nonvar(mia).

yes

?- nonvar(23).

yes

A estrutura dos termos

- Dado um termo complexo que informação pode ser extraída?
  - O functor
  - A aridade
  - O argumento
- O Prolog tem um predicado especializado para obter esta informação

O predicado functor/3

- O predicado functor/3 permite obter o functor e a aridade de um termo complexo

O predicado functor/3

- O predicado functor/3 permite obter o functor e a aridade de um termo complexo
  - ?- functor(friends(lou,andy),F,A).
    F = friends
    A = 2
    yes

O predicado functor/3

- O predicado functor/3 permite obter o functor e a aridade de um termo complexo
  - ?- functor(friends(lou,andy),F,A).
    F = friends
    A = 2
    yes
  - ?- functor([lou,andy,vicky],F,A).
    F = .
    A = 2
    yes

functor/3 e constantes

- O que acontece se usarmos functor/3 com constantes?

functor/3 e constantes

- O que acontece se usarmos functor/3 com constantes?
  - ?- functor(mia,F,A).
    F = mia
    A = 0
    yes

functor/3 e constantes

- O que acontece se usarmos functor/3 com constantes?
  - ?- functor(mia,F,A).
    F = mia
    A = 0
    yes
  - ?- functor(14,F,A).
    F = 14
    A = 0
    yes

functor/3 para construir termos

- O functor/3 pode ser usado para construir termos:
  - ?- functor(Term,friends,2).
    Term = friends(_,_)
    yes

Verificação de termos complexos

```
complexTerm(X):-
    nonvar(X),
    functor(X,_,A),
    A > 0.
```

Argumentos: arg/3

- O predicado arg/3 pode ser usado para obter informação sobre os argumentos de um termo complexo
- Tem três argumentos:
  - Um número *N*
  - Um termo complexo *T*
  - O *N*-ésimo argumento de *T*

Argumentos: arg/3

- O predicado arg/3 pode ser usado para obter informação sobre os argumentos de um termo complexo
- Tem três argumentos:
  - Um número *N*
  - Um termo complexo *T*
  - O *N*-ésimo argumento de *T*

```
?- arg(2,likes(lou,andy),A).
A = andy
yes
```

O Cut

- O retrocesso (backtracking) é uma característica do Prolog
- Mas o retrocesso pode ser ineficiente:
  - O Prolog pode gastar muito tempo a explorar alternativas que não interessam
  - Seria interessante ter algum tipo de controlo
- O predicado cut !/0 oferece uma forma de controlar o retrocesso

Exemplo do cut

- O cut é um predicado Prolog, logo pode ser incluído no corpo das regras:
  - Exemplo:

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

- O cut sucede sempre
- Compromete o Prolog com as escolhas que foram feitas desde a chamada do predicado que unifica com a cabeça da regra

Explicação do cut

- Para explicar o cut vamos:
  - Analisar código Prolog sem cuts e observar o seu comportamento em termos de retrocesso
  - Adicionar cuts a esse código Prolog
  - Examinar como os cuts afectam o retrocesso

Exemplo: código sem cuts

```
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 cuts

```
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).
```

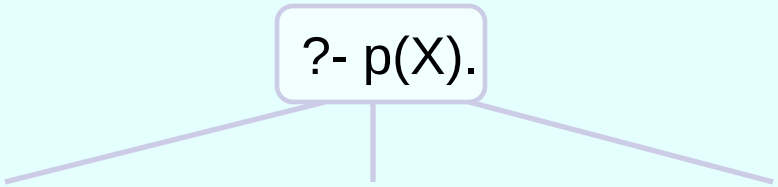
```
?- p(X).
```

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

Exemplo: código sem cuts

```
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).
```

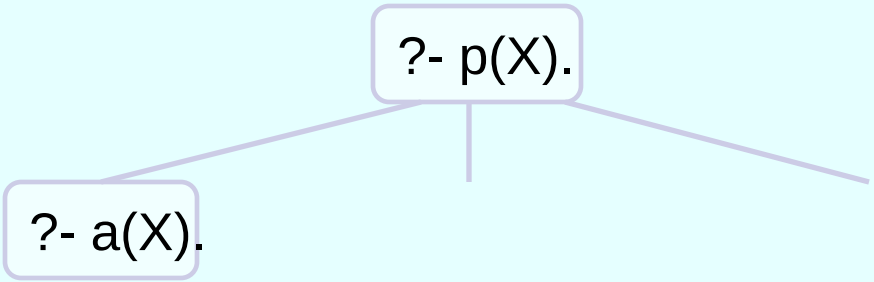
```
?- p(X).
```



Exemplo: código sem cuts

```
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).
```

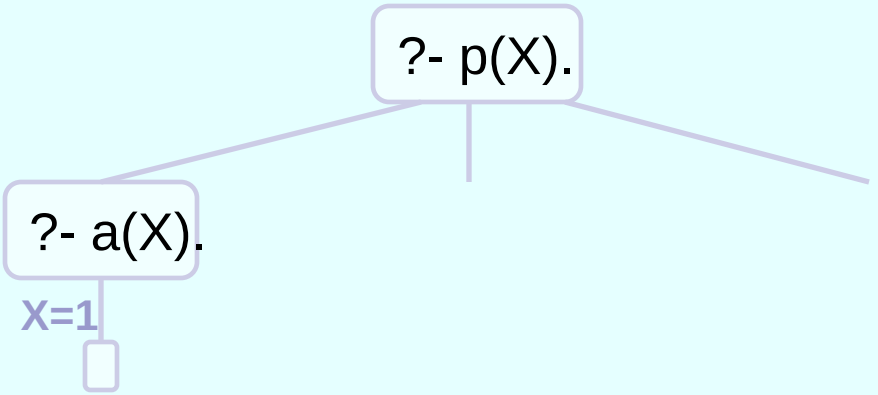
```
?- p(X).
```



Exemplo: código sem cuts

```
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).
```

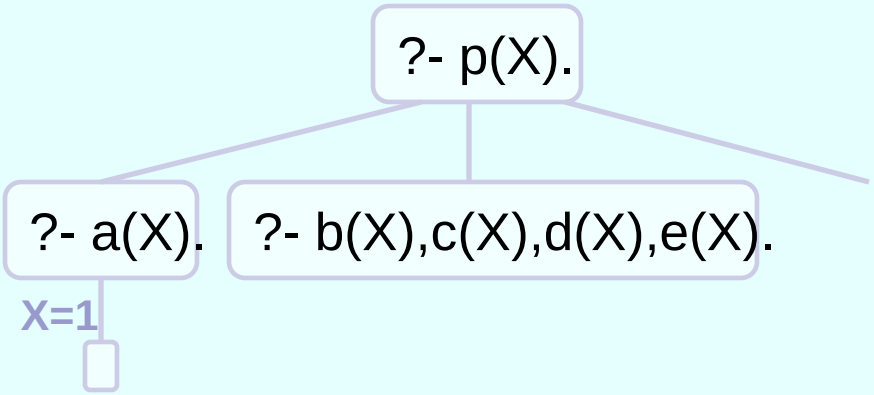
```
?- p(X).
X=1
```



Exemplo: código sem cuts

```
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).
```

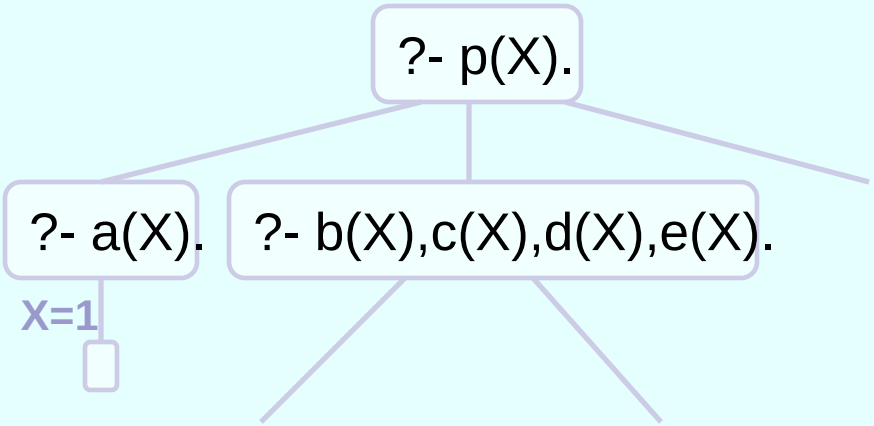
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

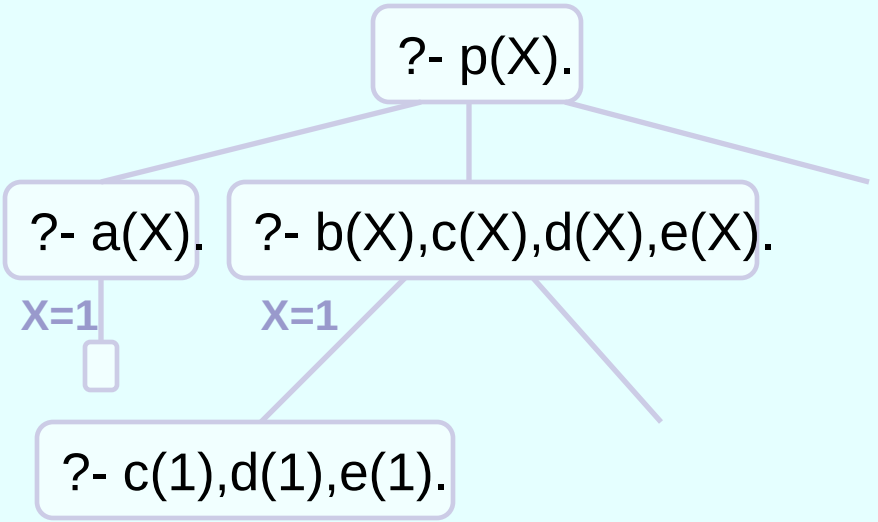
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

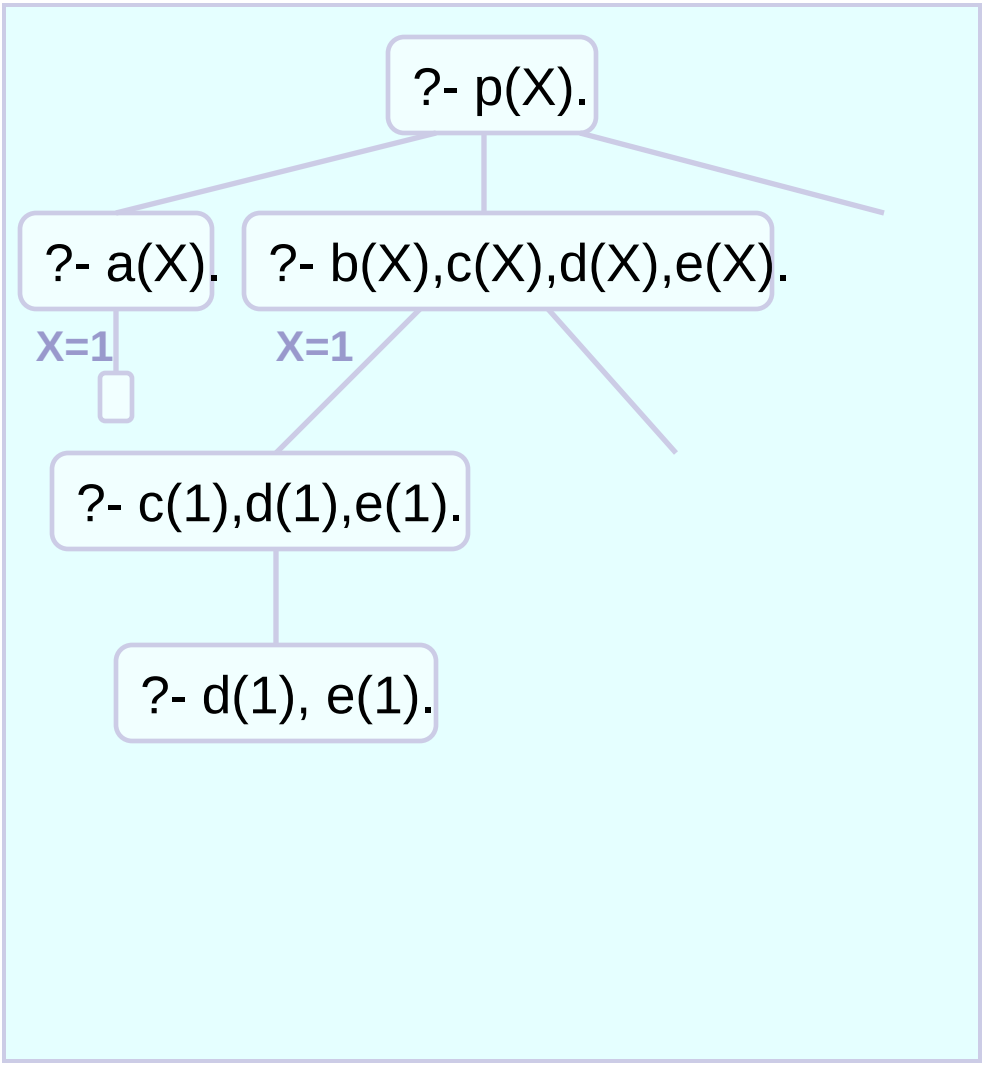
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

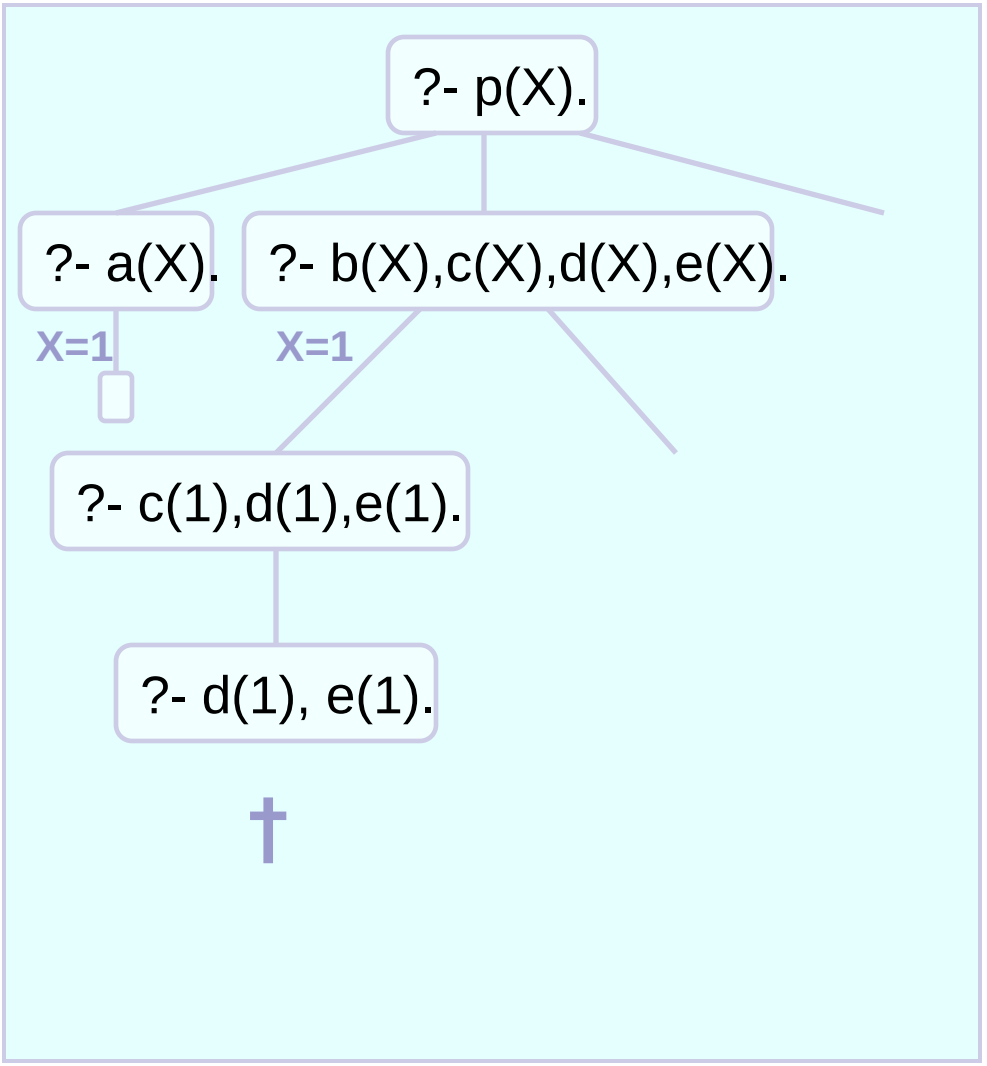
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

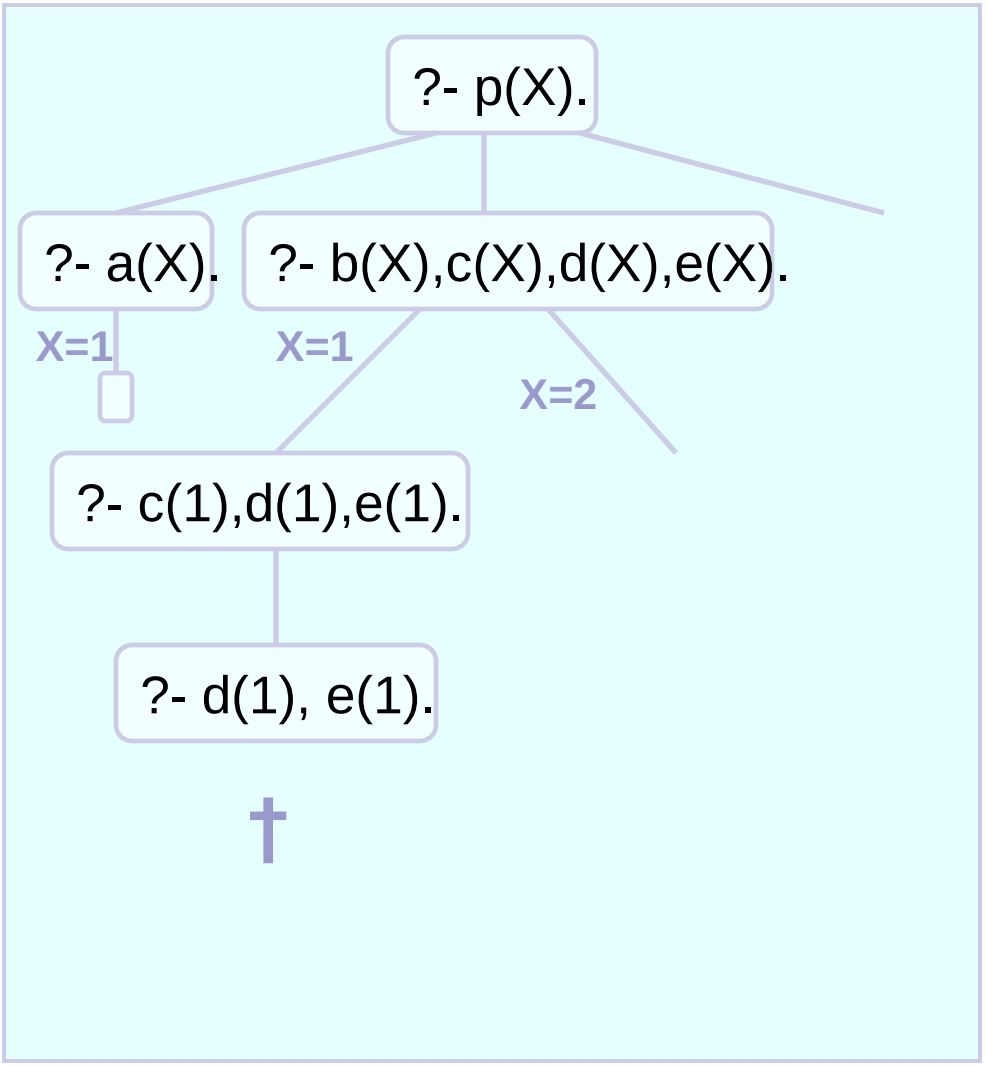
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

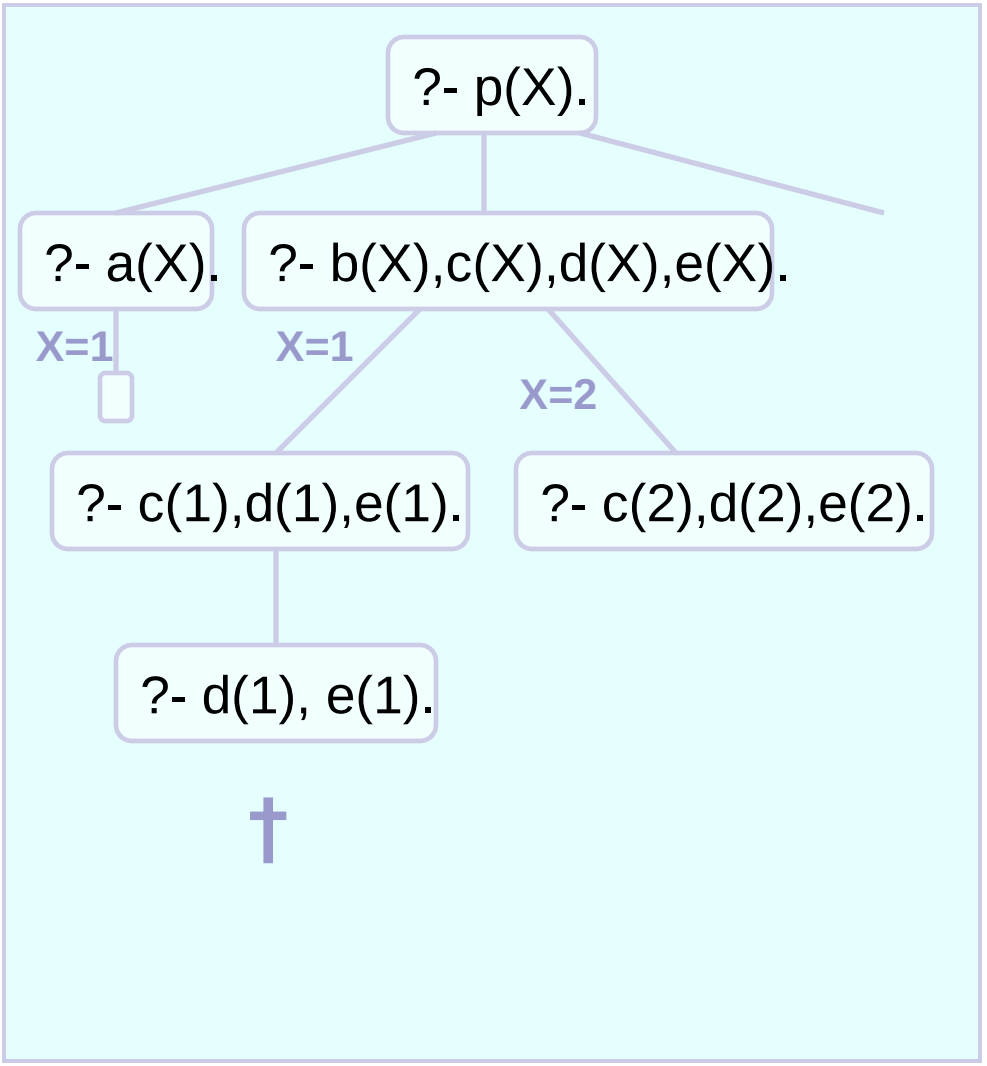
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

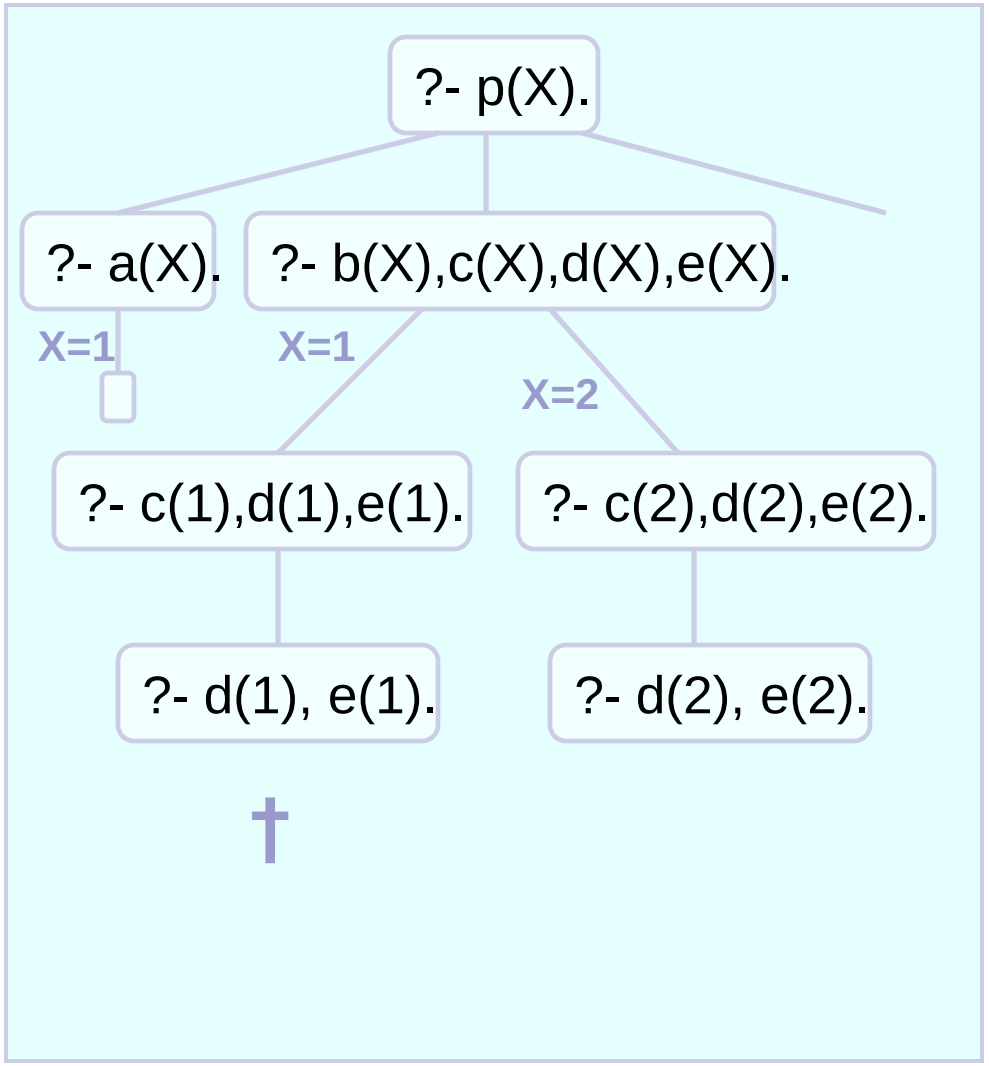
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

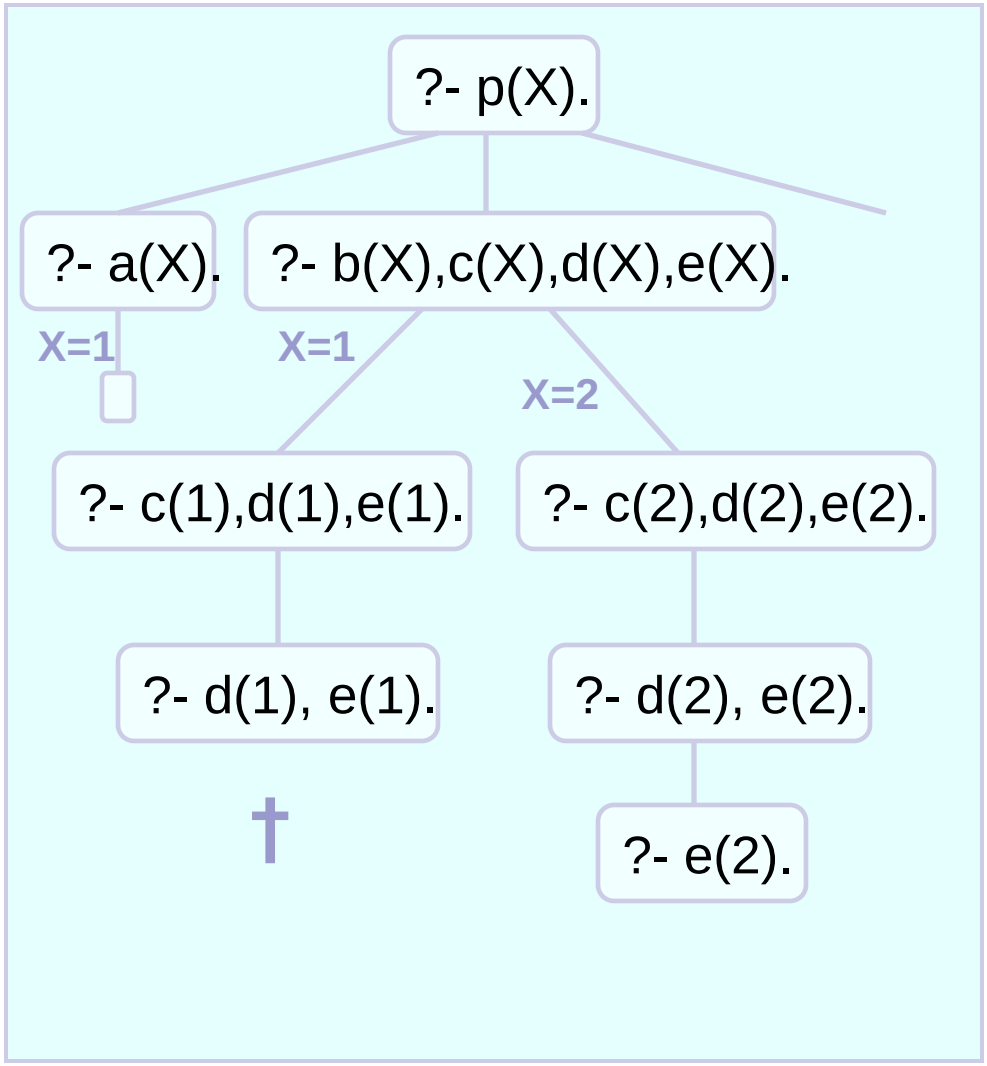
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

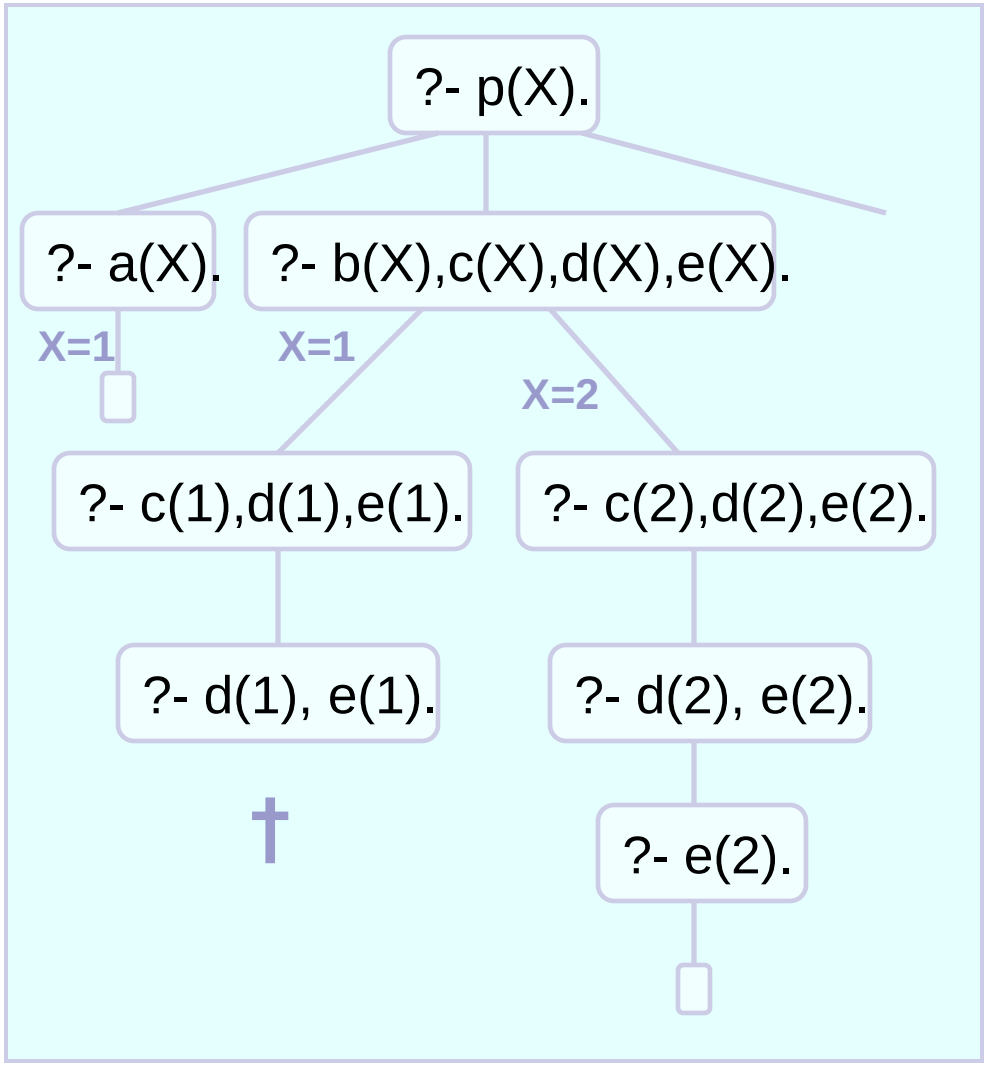
```
?- p(X).
X=1;
```



Exemplo: código sem cuts

```
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).
```

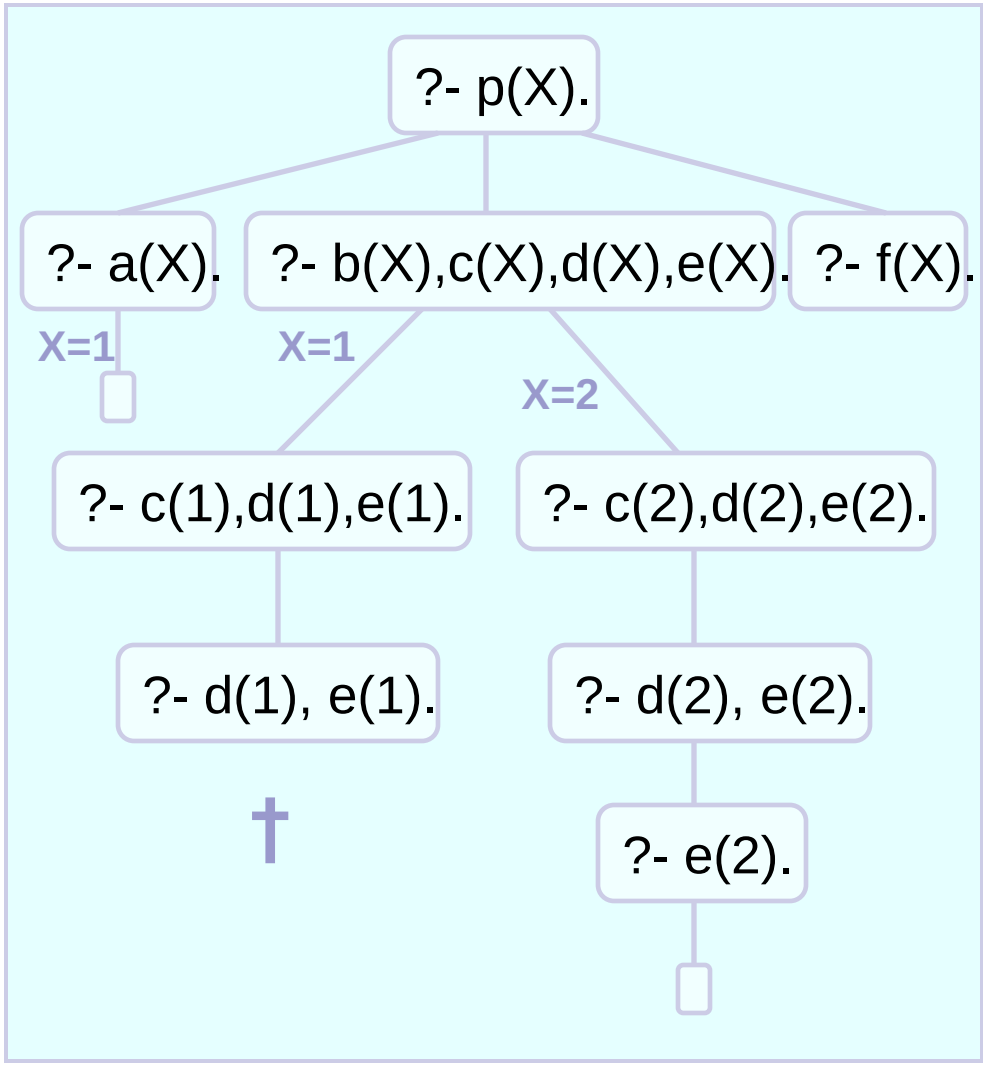
```
?- p(X).
X=1;
X=2
```



Exemplo: código sem cuts

```
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).
```

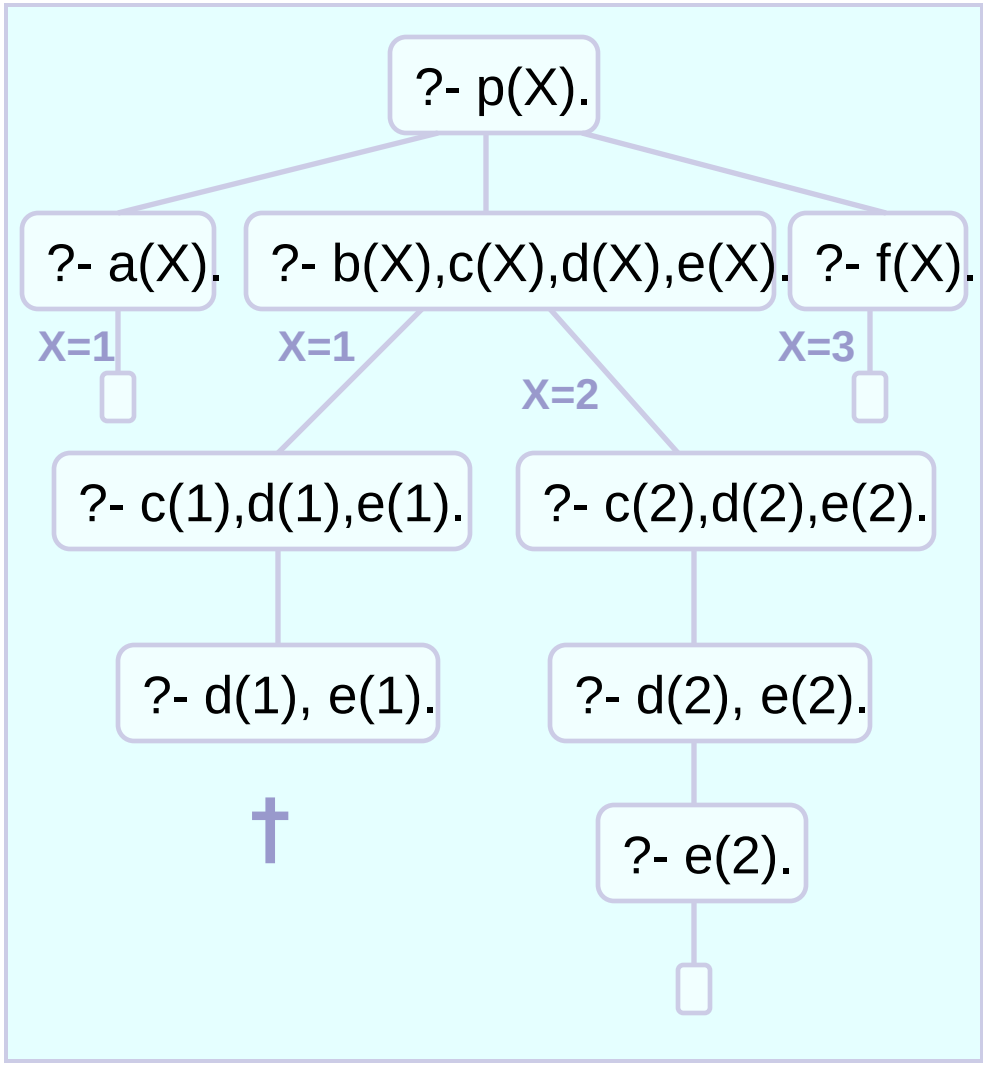
```
?- p(X).
X=1;
X=2;
```



Exemplo: código sem cuts

```
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).
```

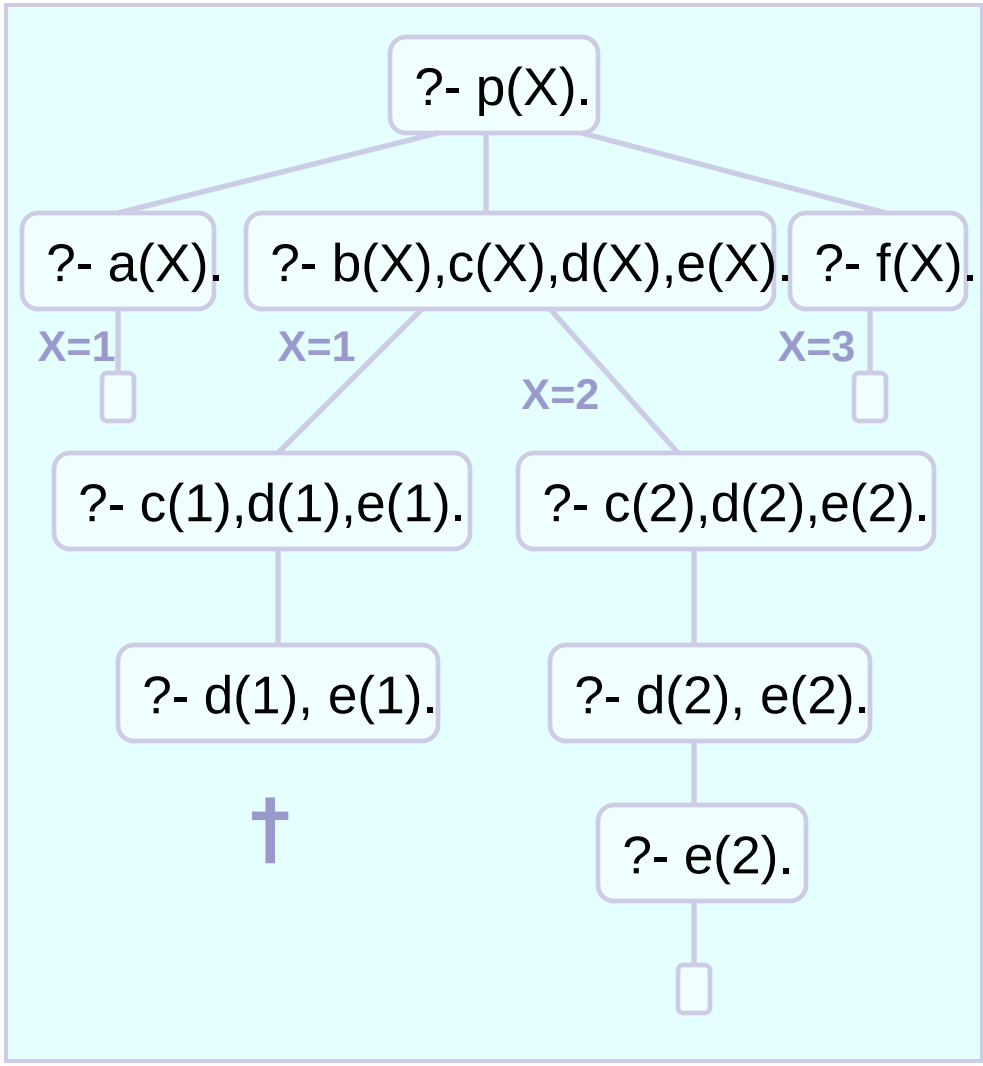
```
?- p(X).
X=1;
X=2;
X=3
```



Exemplo: código sem cuts

```
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
```



Adicionar um cut

- Vamos supor que se insere um cut na segunda cláusula:

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

- Se agora fizermos a mesma interrogação obteremos a seguinte resposta:

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

Exemplo: cut

```
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: cut

```
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).
```

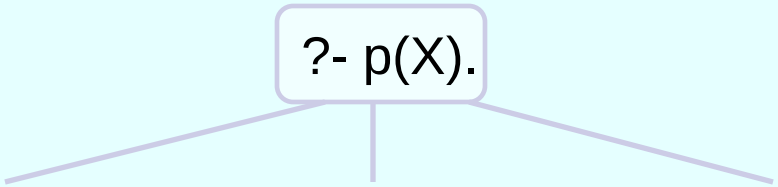
```
?- p(X).
```

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

Exemplo: cut

```
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).
```

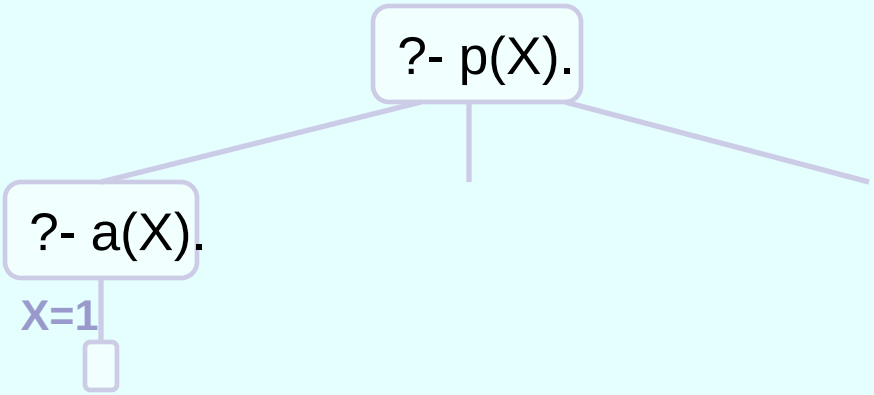
```
?- p(X).
```



Exemplo: cut

```
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).
```

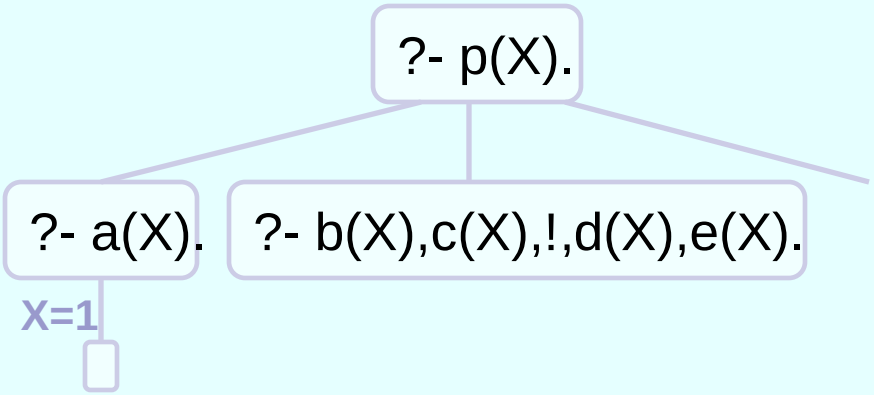
```
?- p(X).
X=1
```



Exemplo: cut

```
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).
```

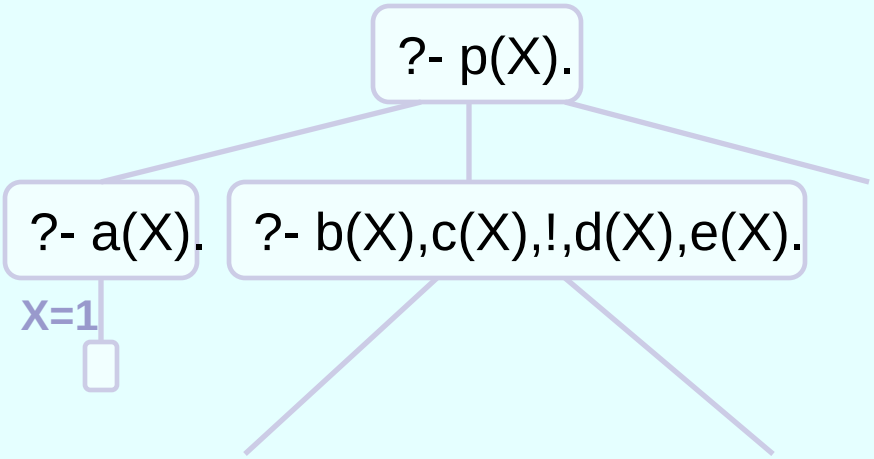
```
?- p(X).
X=1;
```



Exemplo: cut

```
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).
```

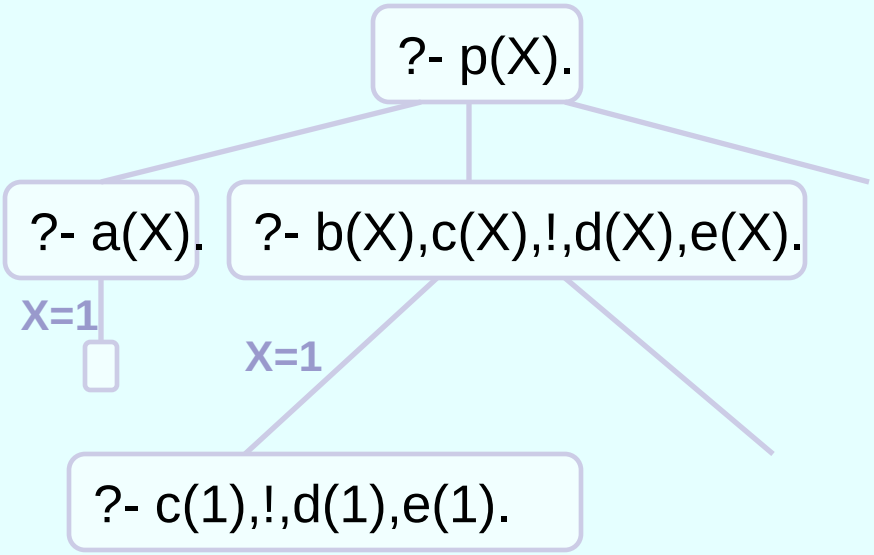
```
?- p(X).
X=1;
```



Exemplo: cut

```
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).
```

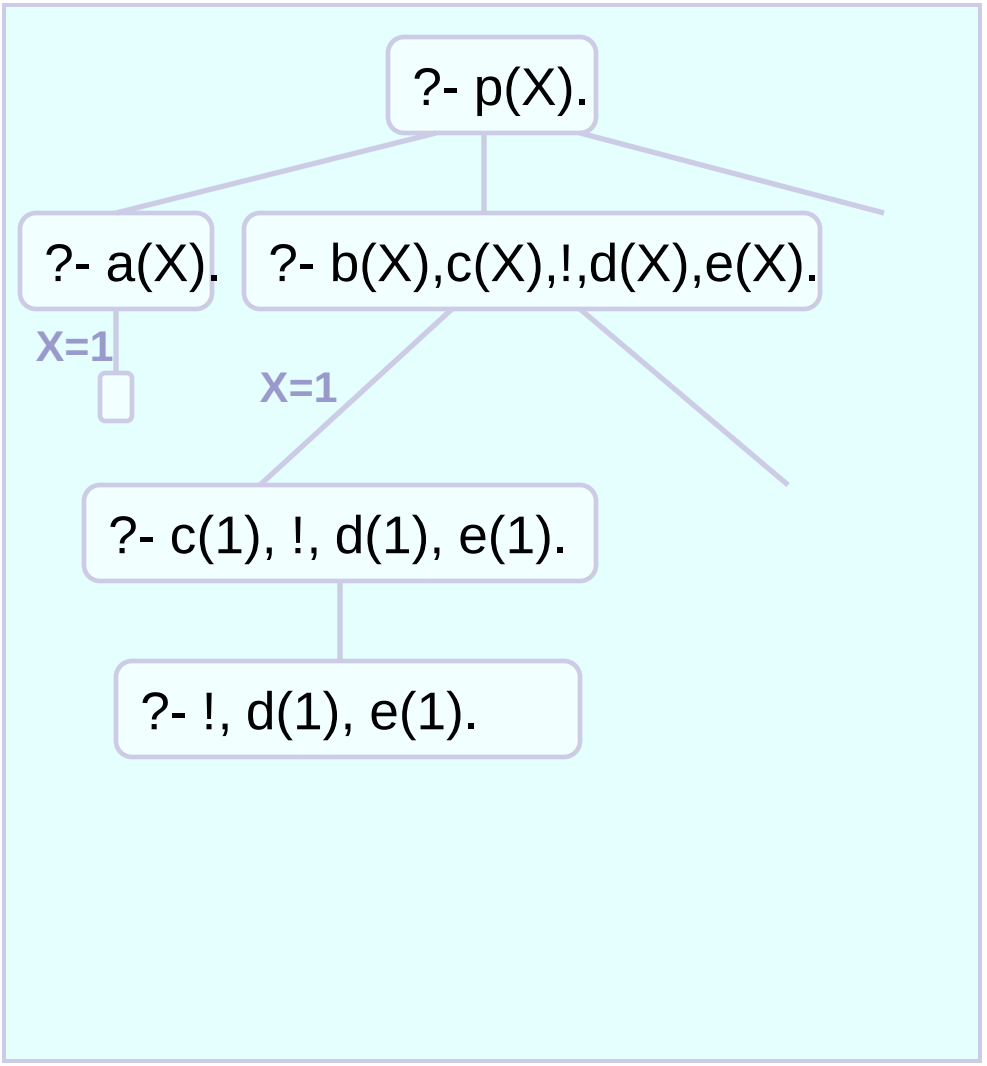
```
?- p(X).
X=1;
```



Exemplo: cut

```
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).
```

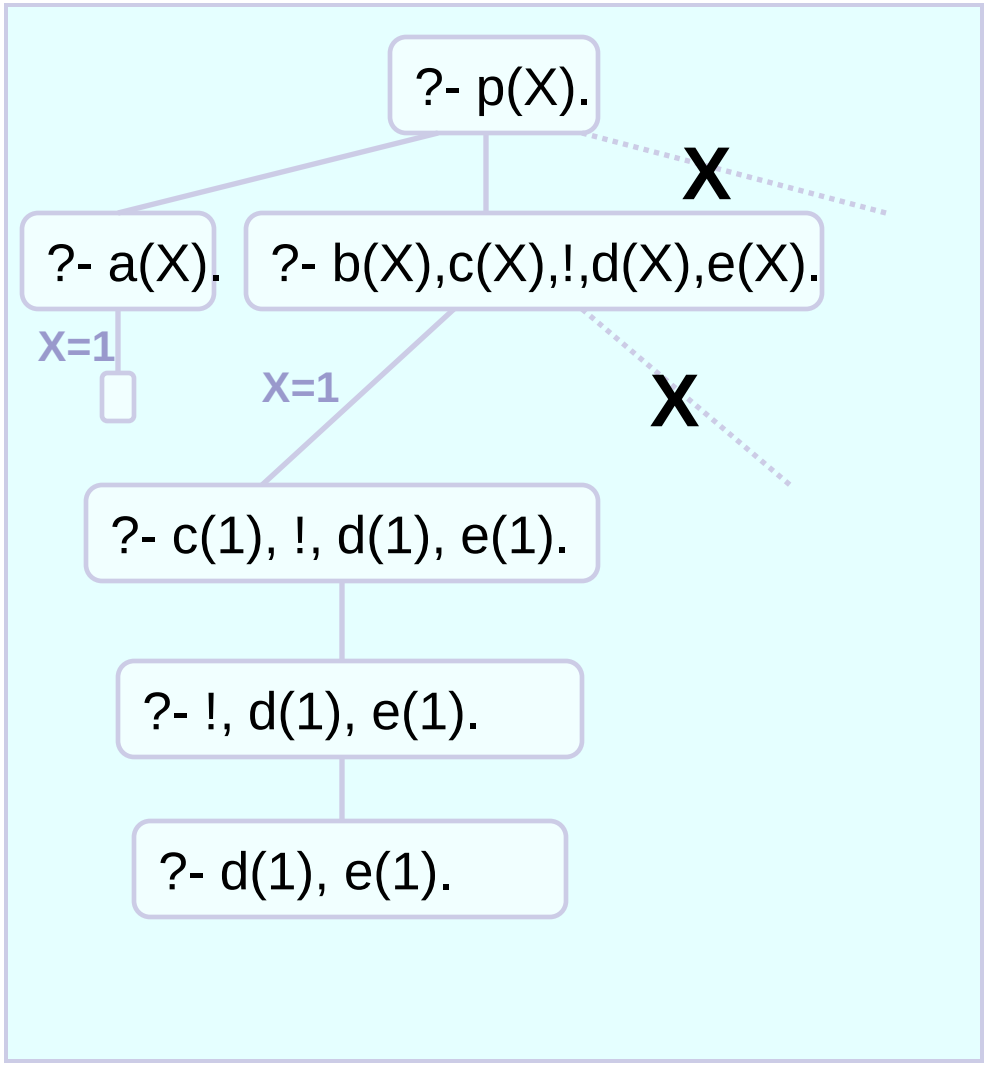
```
?- p(X).
X=1;
```



Exemplo: cut

```
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).
```

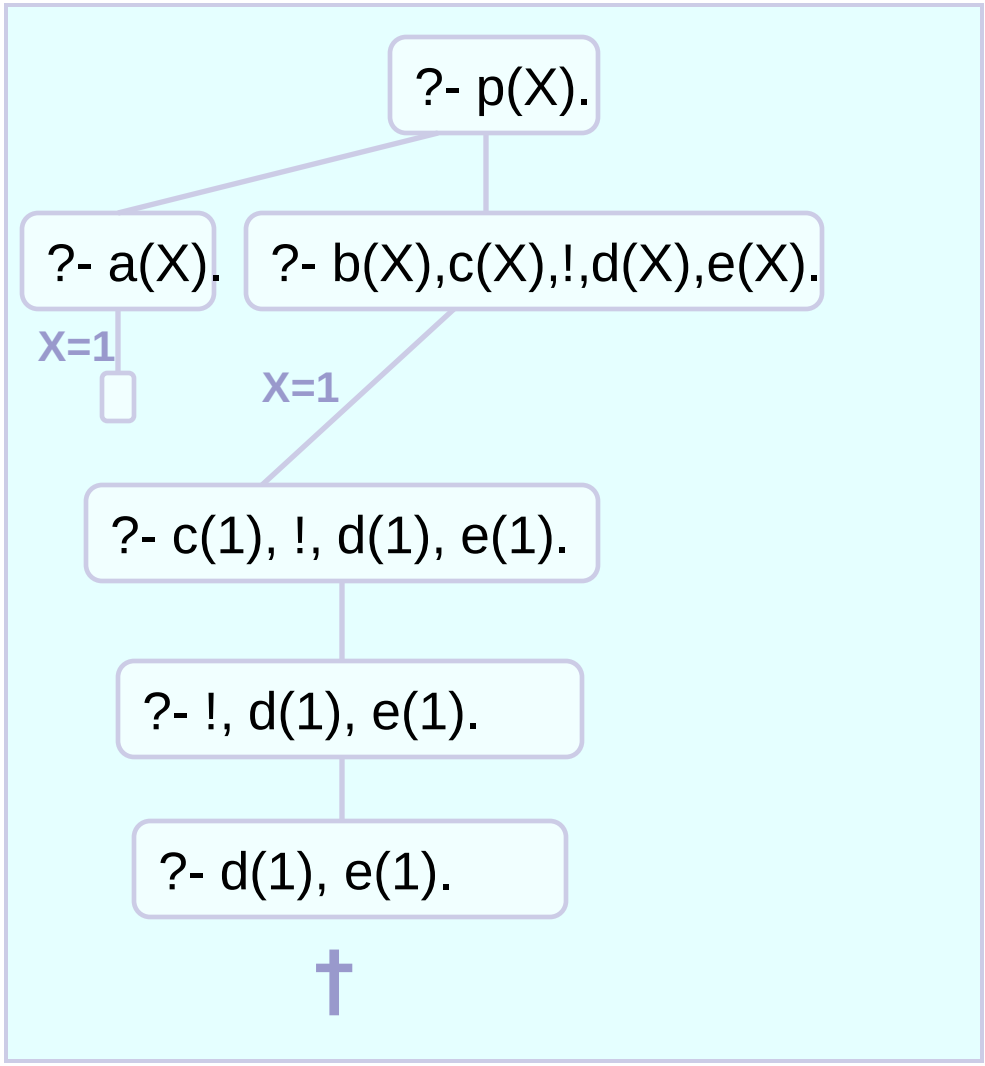
```
?- p(X).
X=1;
```



Exemplo: cut

```
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 faz o cut

- O cut apenas compromete com as escolhas feitas desde a unificação do predicado com a cabeça da cláusula que contém o cut
- Por exemplo, numa regra da forma

  q:- p_1, … , p_n, !, r_1, … , r_n.

  Quando chega ao cut compromete-se:
  - com esta cláusula em particular para q
  - com as escolhas feitas por p_1, … , p_n
  - NÃO com as escolhas feitas por r_1, … , r_n

Usar o Cut

- Considere-se o seguinte predicado max/3 que sucede se o terceiro argumento é o máximo dos dois primeiros

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

Usar o Cut

- Considere-se o seguinte predicado max/3 que sucede se o terceiro argumento é o máximo dos dois primeiros

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

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

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

Usar o Cut

- Considere-se o seguinte predicado max/3 que sucede se o terceiro argumento é o máximo dos dois primeiros

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

```
?- max(2,3,2).
no

?- max(2,3,5).
no
```

O predicado max/3

- Qual é o problema?
- Existe uma potencial ineficiência
  - Considere-se que é chamado com
    ?- max(3,4,Y).
  - Unifica correctamente Y com 4
  - Mas quando são pedidas alternativas, tenta satisfazer a segunda cláusula. Absolutamente desnecessário!

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

max/3 com cut

- Com o cut é possível resolver o problema

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

- Como funciona:
  - Se X =< Y sucede, o cut compromete com esta escolha, e a segunda cláusula de max/3 não é considerada
  - Se X =< Y falha, o Prolog prossegue pata a segunda cláusula

Cuts verdes

- Os cuts que não alteram o significado de um predicado chamam-se **cuts verdes**
- O cut em max/3 é um exemplo de um cut verde:
  - o novo código dá exactamente as mesmas respostas que a versão anterior,
  - mas é mais eficiente

Outro max/3 com cut

- Porque não se remove o corpo de segunda cláusula? Afinal, ele é redundante.

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

- Será boa ideia?

Outro max/3 com cut

- Porque não se remove o corpo de segunda cláusula? Afinal, ele é redundante.

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

- Será boa ideia?
  - ok

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

Outro max/3 com cut

- Porque não se remove o corpo de segunda cláusula? Afinal, ele é redundante.

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

- Será boa ideia?
  - ok

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

Outro max/3 com cut

- Porque não se remove o corpo de segunda cláusula? Afinal, ele é redundante.

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

- Será boa ideia?
  - oops….

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

max/3 com cut revisto

- Unificar depois de passar pelo cut

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

- Agora já funciona

```
?- max(200,300,200).
no
```

Cuts vermelhos

- Cuts que alteram o significado de um predicado chamam-se **cuts vermelhos**
- O cut do max/3 revisto é um exemplo de um cut vermelho:
  - Se se remover o cut, não se obtém um programa equivalente
- Programas com cuts vermelhos
  - Não são puramente declarativos
  - Podem ser difíceis de ler
  - Podem levar a erro de programação subtis

Outro predicado Prolog: fail/0

- Como sugere o nome, este predicado falha sempre
- Não parece muito útil!
- No entanto: quando o Prolog falha, tenta novas alternativas por retrocesso

Vincent

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

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

- A combinação cut fail
  permite codificar excepções

Vincent

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

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

- A combinação cut fail permite codificar excepções

```
?- enjoys(vincent,a).
yes
```

Vincent

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

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

- A combinação cut fail permite codificar excepções

```
?- enjoys(vincent,b).
no
```

Vincent

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

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

- A combinação cut fail permite codificar excepções

```
?- enjoys(vincent,c).
yes
```

Vincent

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

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

- A combinação cut fail permite codificar excepções

```
?- enjoys(vincent,d).
yes
```

Negação por falha

- A combinação cut-fail permite a implementação da negação por falha:

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

Vincent revisitado

```
enjoys(vincent,X):- burger(X),
                    neg(bigKahunaBurger(X)).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

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

Vincent revisitado

```
enjoys(vincent,X):- burger(X),
                    neg(bigKahunaBurger(X)).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

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

```
?- enjoys(vincent,X).
X=a
X=c
X=d
```

Outro predicado Prolog: \+

- Em Prolog o operador prefixo \+ significa negação por falha
- As preferências do Vincent podem ser representadas da seguinte forma:

```
enjoys(vincent,X):-  burger(X),
                     \+ bigKahunaBurger(X).
```

```
?- enjoys(vincent,X).
X=a
X=c
X=d
```

Negação por falha e lógica

- A negação por falha é diferente da negação em lógica
- Mudar a ordem dos predicados altera o resultado obtido:

```
enjoys(vincent,X):- \+ bigKahunaBurger(X),
                      burger(X).
```

```
?- enjoys(vincent,X).
no
```

Obter múltiplas soluções

- Podem existir muitas soluções para uma interrogação Prolog
- No entanto, o Prolog gera um solução de cada vez
- Por vezes necessitamos de obter *todas* as soluções para uma interrogação

Obter múltiplas soluções

- O Prolog tem três predicados que fazem isso: **findall**/3, **bagof**/3 e **setof**/3
- Todos estes predicados obtêm todas as soluções e juntam-nas numa lista
- Mas há diferenças importantes entre eles

findall/3

- A interrogação

```
?- findall(O,G,L).
```

devolve uma lista **L** com todos os objectos **O** que satisfazem o golo **G**

  - Sucede sempre
  - Unifica **L** com a lista vazia se **G** não pode ser satisfeito

Exemplo: findall/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(X,descend(martha,X),L).
L=[charlotte,caroline,laura,rose]
yes
```

Exemplo: findall/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(f:X,descend(martha,X),L).
L=[f:charlotte,f:caroline,f:laura,f:rose]
yes
```

Exemplo: findall/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(X,descend(rose,X),L).
L=[ ]
yes
```

Exemplo: findall/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(d,descend(martha,X),L).
L=[d,d,d,d]
yes
```

findall/3 nem sempre é o adequado

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(Chi,descend(Mot,Chi),L).
L=[charlotte,caroline,laura, rose,
    caroline,laura,rose,laura,rose,rose]
yes
```

bagof/3

- A interrogação

```
?- bagof(O,G,L).
```

devolve uma lista **L** com todos os objectos **O** que satisfazem o golo **G**

  - Só sucede se o golo **G** sucede
  - Instancia variáveis livres de **G**

Exemplo: bagof/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):-
    child(X,Y).
descend(X,Y):-
    child(X,Z),
    descend(Z,Y).
```

```
?- bagof(Chi,descend(Mot,Chi),L).
Mot=caroline
L=[laura, rose];
Mot=charlotte
L=[caroline,laura,rose];
Mot=laura
L=[rose];
Mot=martha
L=[charlotte,caroline,laura,rose];
no
```

Exemplo: bagof/3 com ^

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):-
    child(X,Y).
descend(X,Y):-
    child(X,Z),
    descend(Z,Y).
```

```
?- bagof(Chi,Mot^descend(Mot,Chi),L).
L=[charlotte, caroline, laura, rose,
    caroline,laura,rose,laura, rose, rose]
```

setof/3

- A interrogação

```
?- setof(O,G,L).
```

devolve uma lista ordenada **L** com todos os objectos **O** que satisfazem o golo **G**

  - Só sucede se o golo **G** sucede
  - Instancia variáveis livres de **G**
  - Remove duplicados de **L**
  - Ordena os elementos de **L**

Exemplo: setof/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):-
    child(X,Y).
descend(X,Y):-
    child(X,Z),
    descend(Z,Y).
```

```
?- bagof(Chi,Mot^descend(Mot,Chi),L).
L=[charlotte, caroline, laura, rose,
    caroline, laura, rose, laura, rose,
    rose]
yes

?-
```

Exemplo: setof/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):-
    child(X,Y).
descend(X,Y):-
    child(X,Z),
    descend(Z,Y).
```

```
?- bagof(Chi,Mot^descend(Mot,Chi),L).
L=[charlotte, caroline, laura, rose,
    caroline, laura, rose, laura, rose,
    rose]
yes

?- setof(Chi,Mot^descend(Mot,Chi),L).
L=[caroline, charlotte, laura, rose]
yes

?-
```