

Cuts, Negación, Términos

Objetivos

- Explicar como se realiza el control del backtracking de Prolog mediante el cut.
- Introducir la negación en Prolog
- Acceso a las componentes estructurales de los términos complejos

Cut

- El backtracking es un rasgo característico de Prolog
- Se produce cada vez que se llega a un punto de falla en el árbol-SLD, es de hecho parte del mecanismo para recorrer el árbol-SLD
- El backtracking puede implicar ineficiencia si se exploran posibilidades que no van a conducir a algo de interés.
- El predicado cut !/0 permite controlar el backtracking, realizando una poda en el árbol-SLD

Ejemplo de cut

- El cut es un predicado del sistema, lo podemos usar en el cuerpo de las reglas.

- Ejemplo:

$p(X) \text{:- } b(X), c(X), !, d(X), e(X).$

- Cut (!) es un objetivo que siempre tiene éxito.
- Fija las opciones que fueron tomadas desde la invocación del predicado padre.

Cut, ejemplo

- Veremos el funcionamiento del intérprete en términos de:
 - Ver la conducta en términos de backtracking de código Prolog sin cut
 - Ver la conducta del mismo código con cuts agregados, en lo que refiere al backtracking.

Ejemplo: código sin 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).
```

Ejemplo: código sin 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).
```

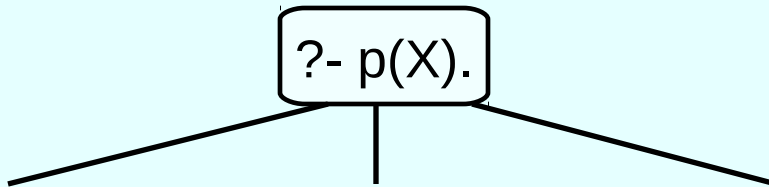
```
?- p(X).
```

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

Ejemplo: código sin 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).
```



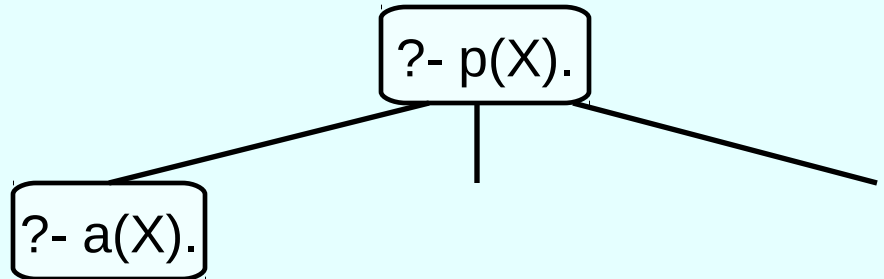
?- p(X).

The diagram shows a rectangular box containing the text '?- p(X)'. Three lines extend from the bottom of the box: one to the left, one straight down, and one to the right, representing connections to other parts of a query graph.

Ejemplo: código sin 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).
```

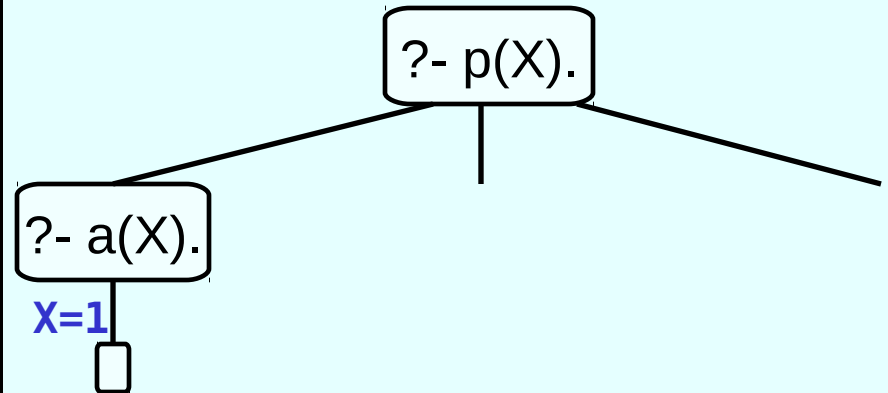
```
?- p(X).
```



Ejemplo: código sin 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).
```

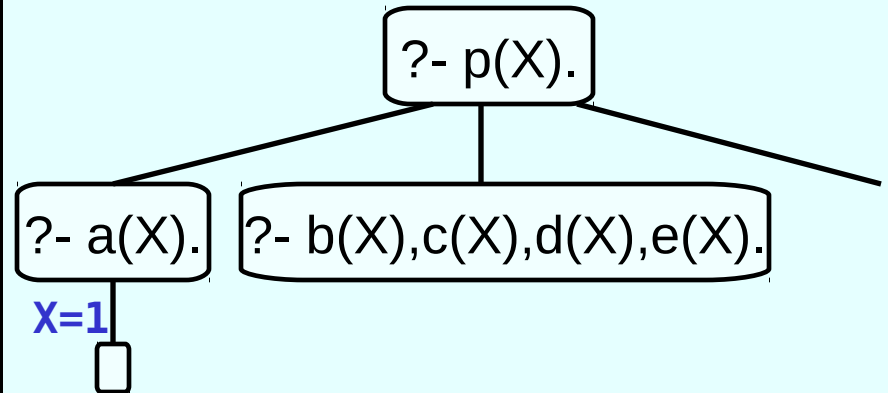
```
?- p(X).  
X=1
```



Ejemplo: código sin 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).
```

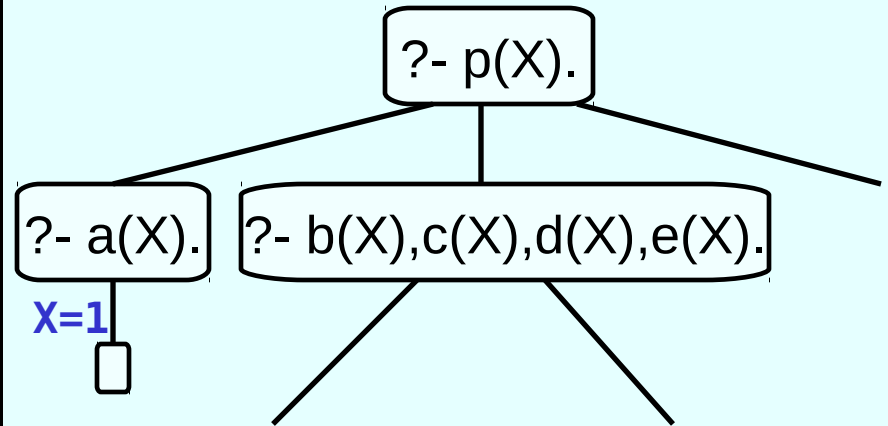
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

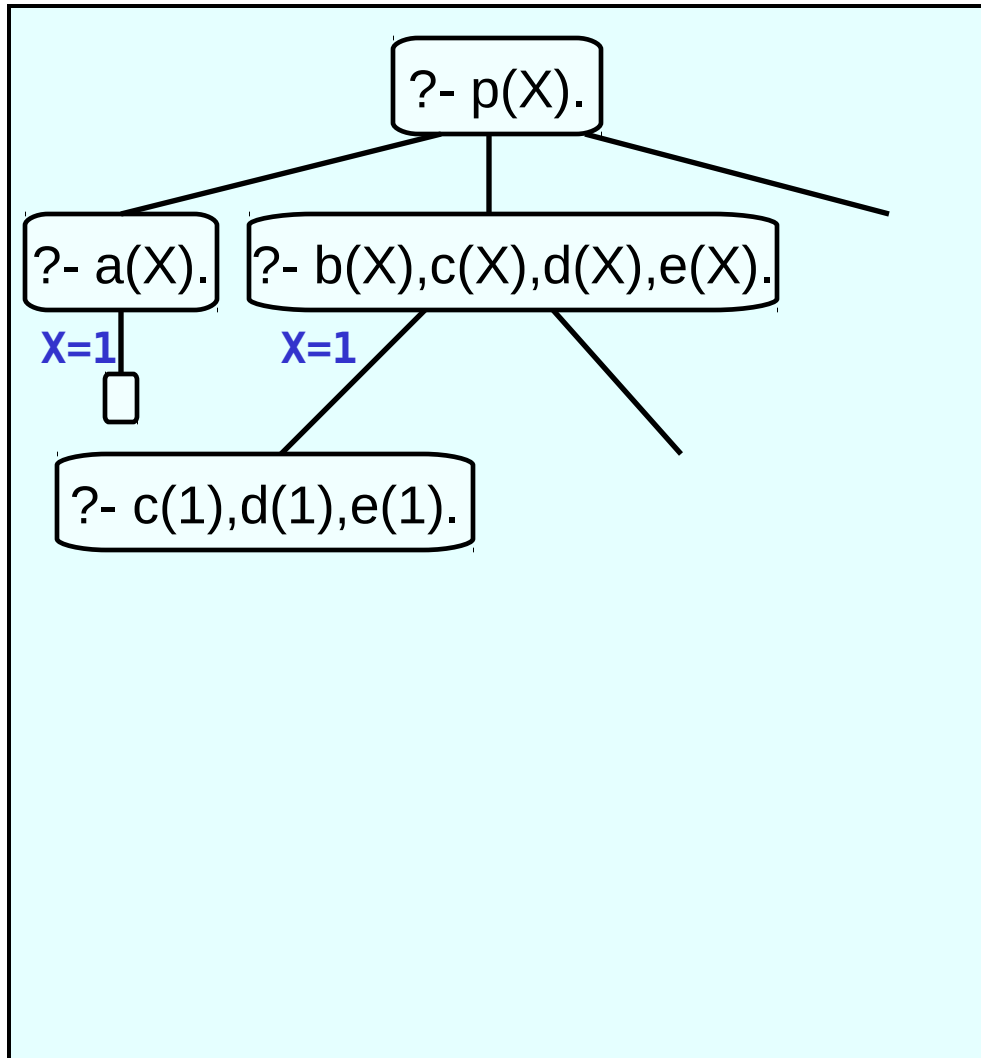
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

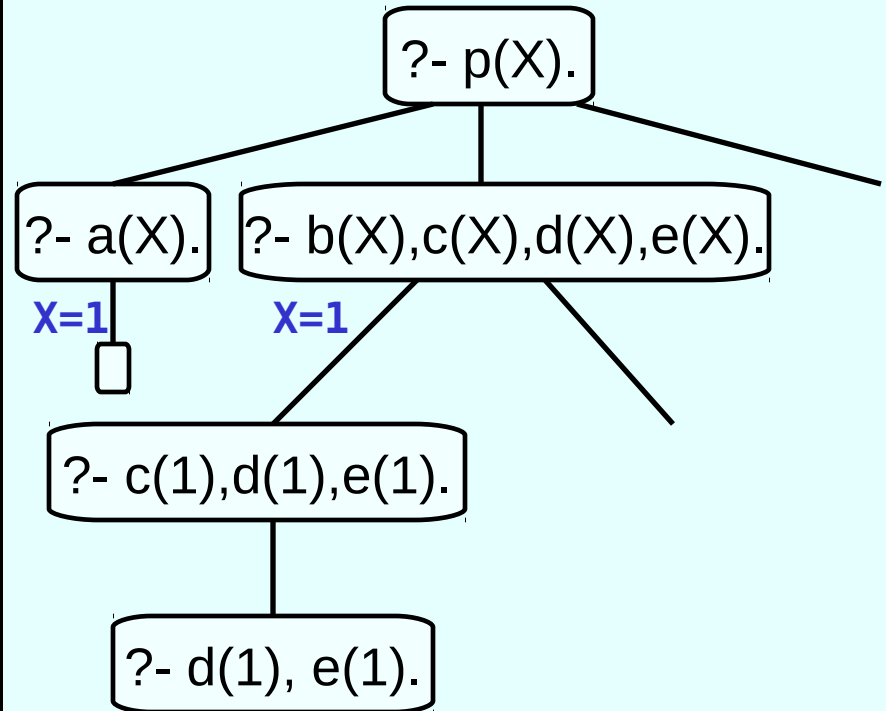
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

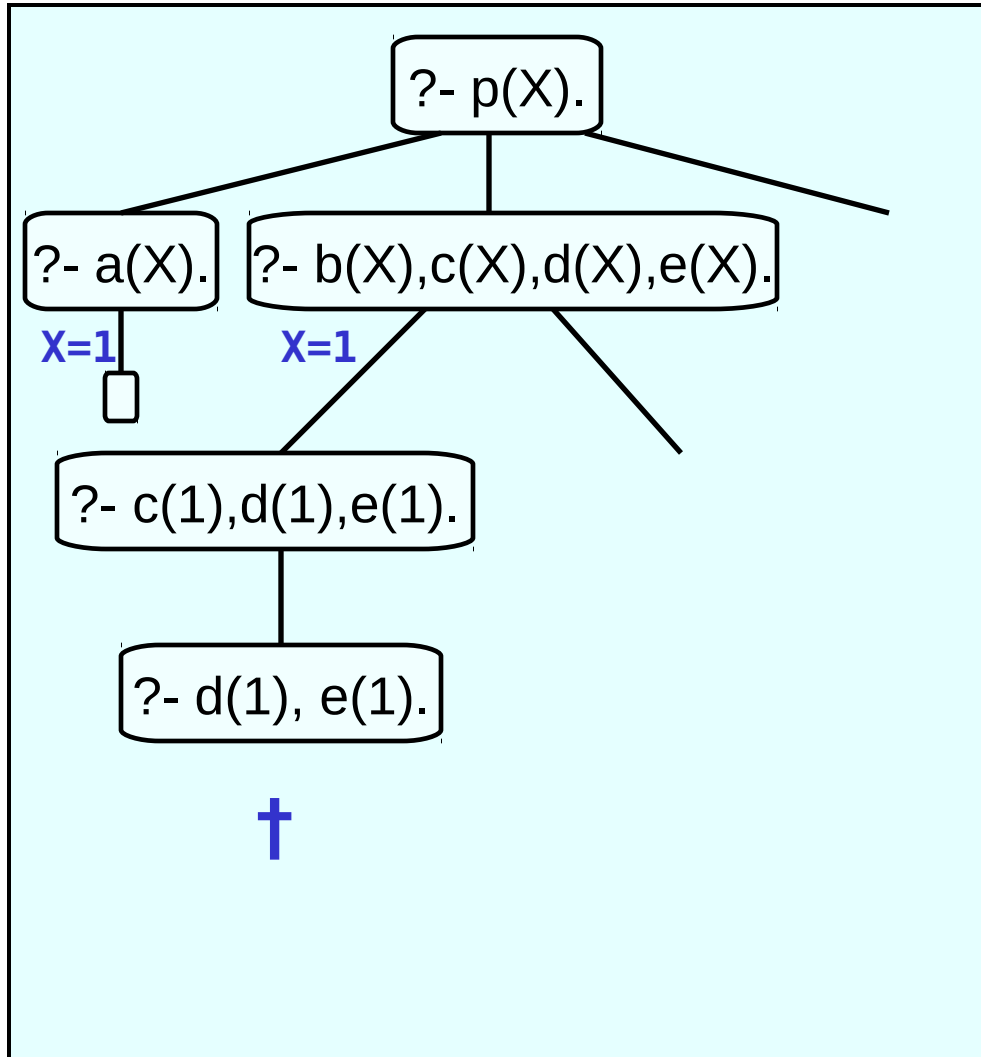
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

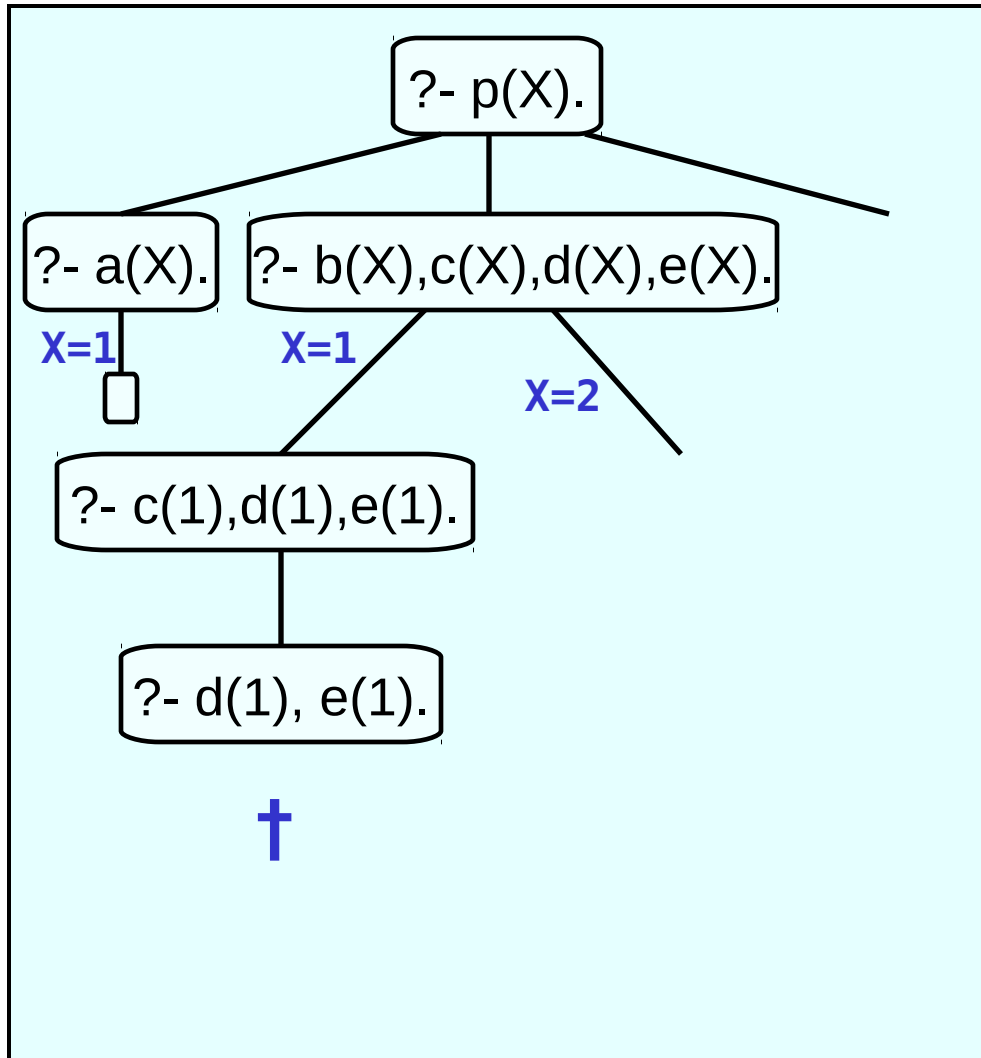
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

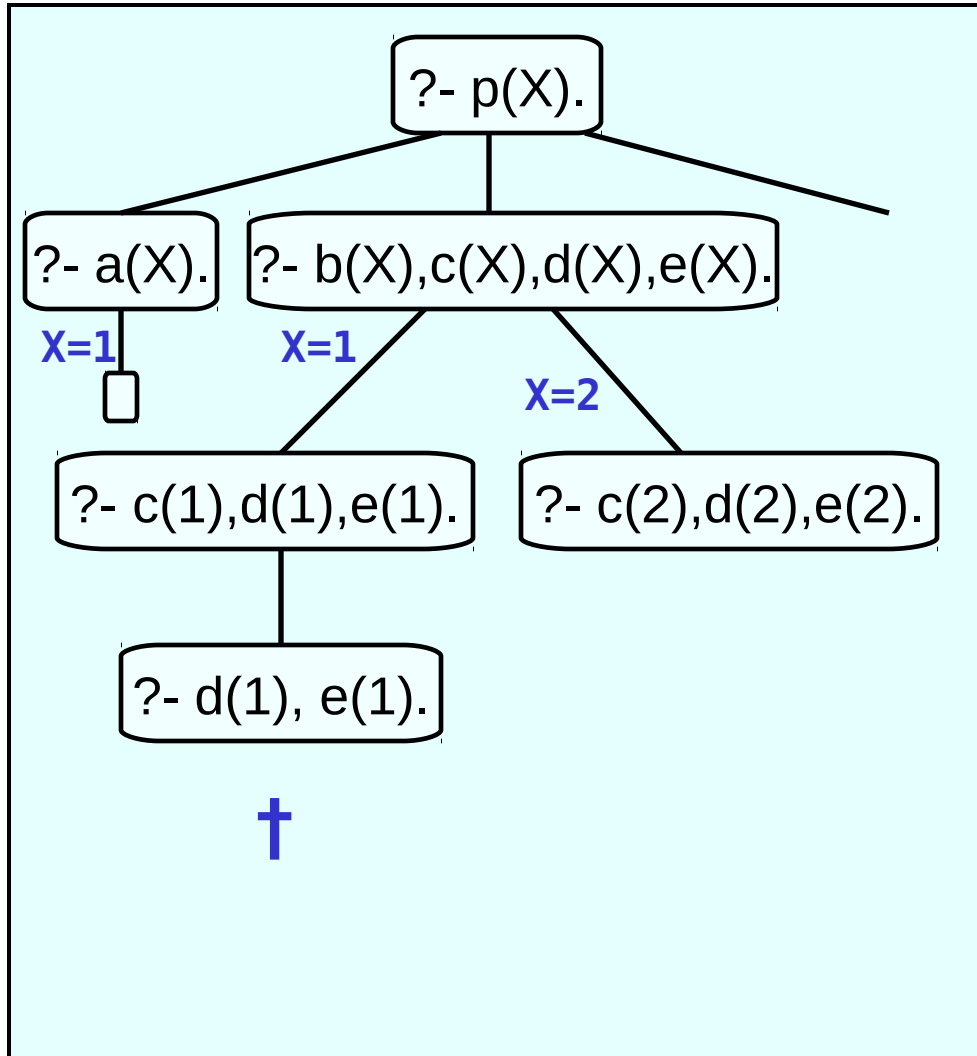
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

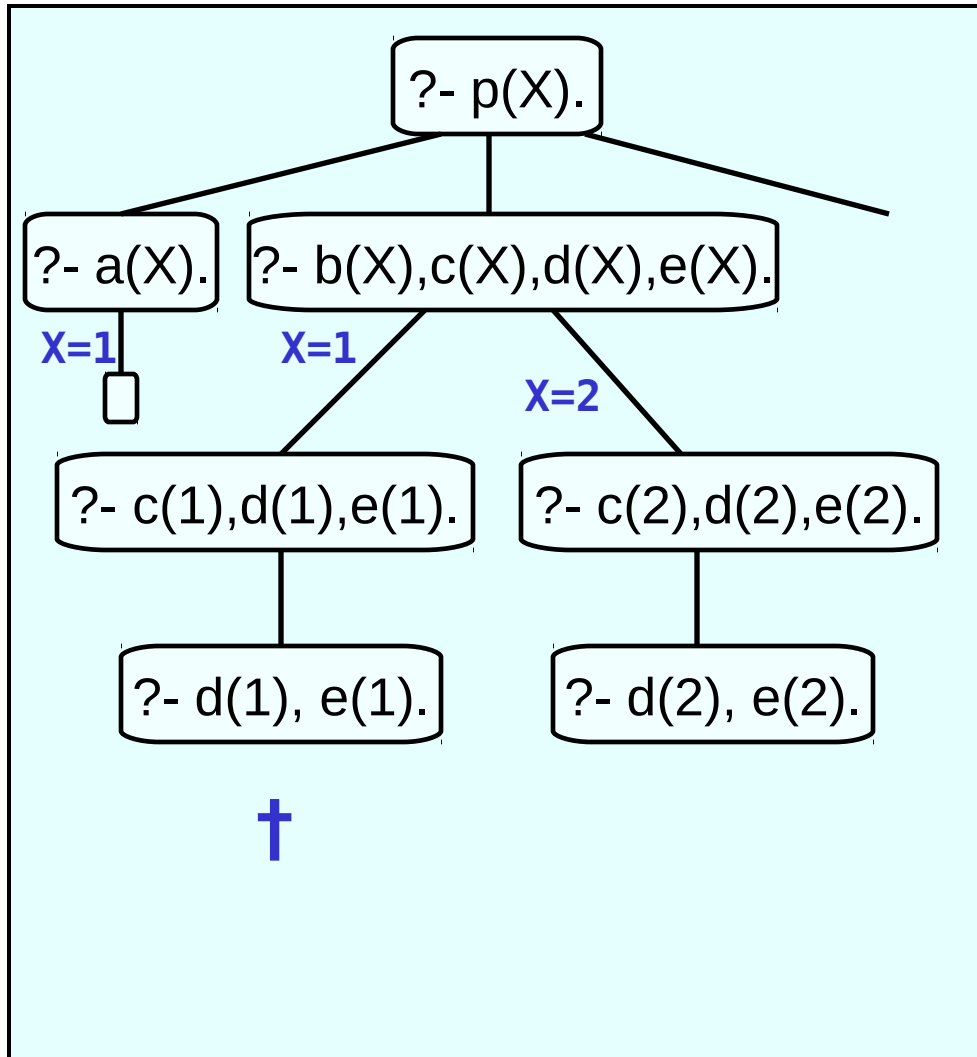
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

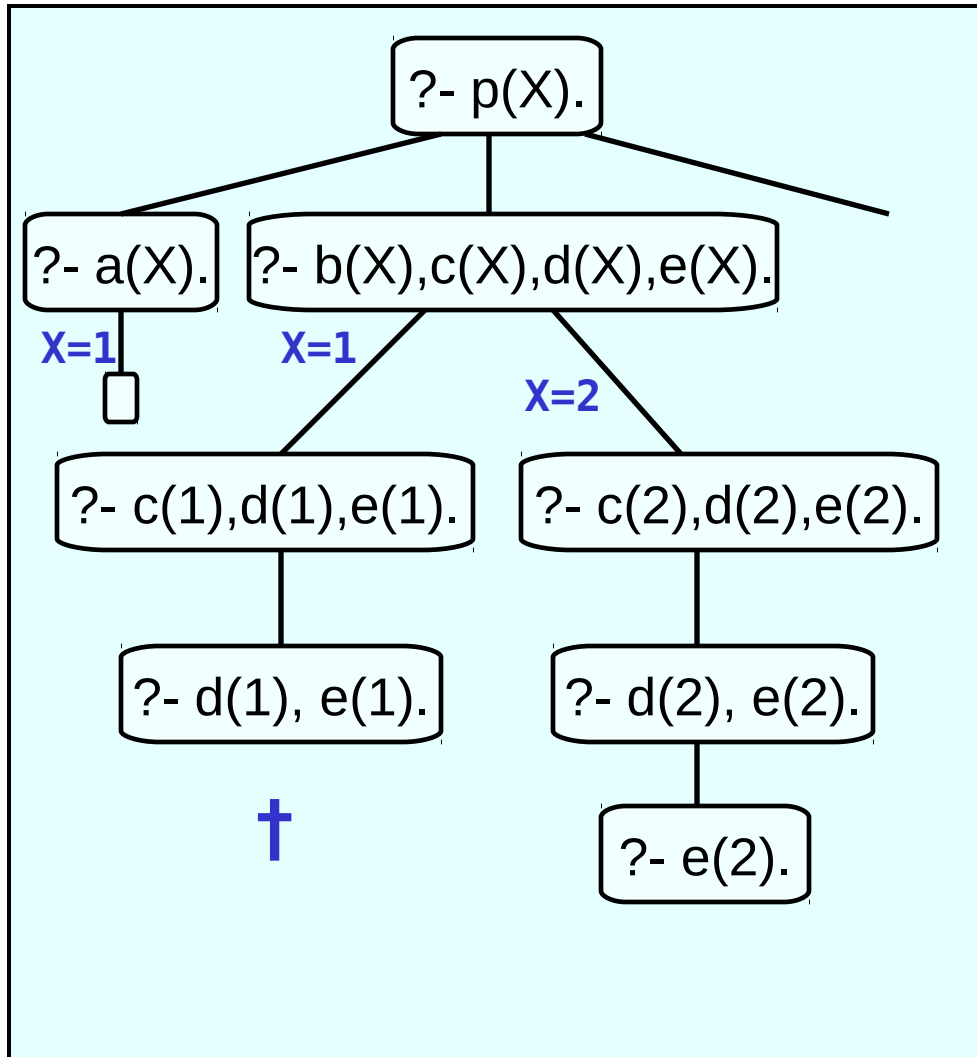
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

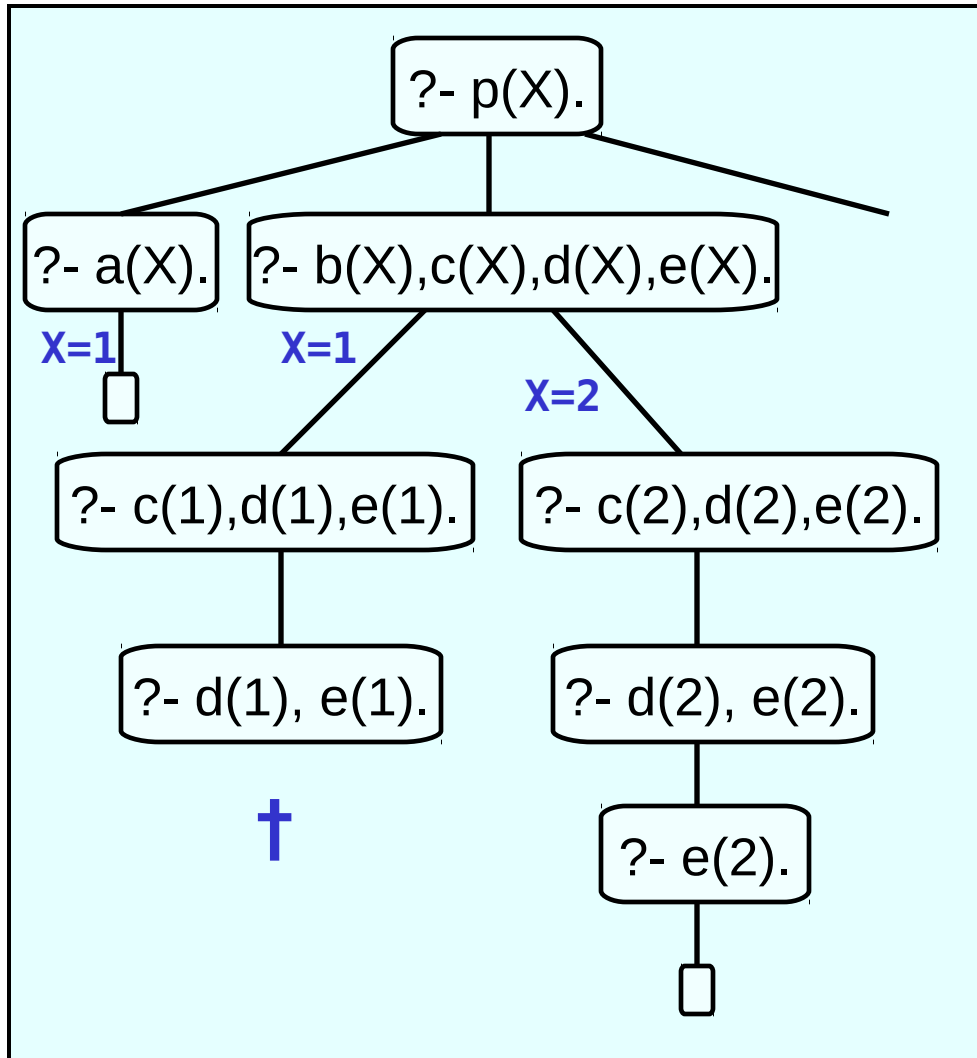
```
?- p(X).  
X=1;
```



Ejemplo: código sin 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).
```

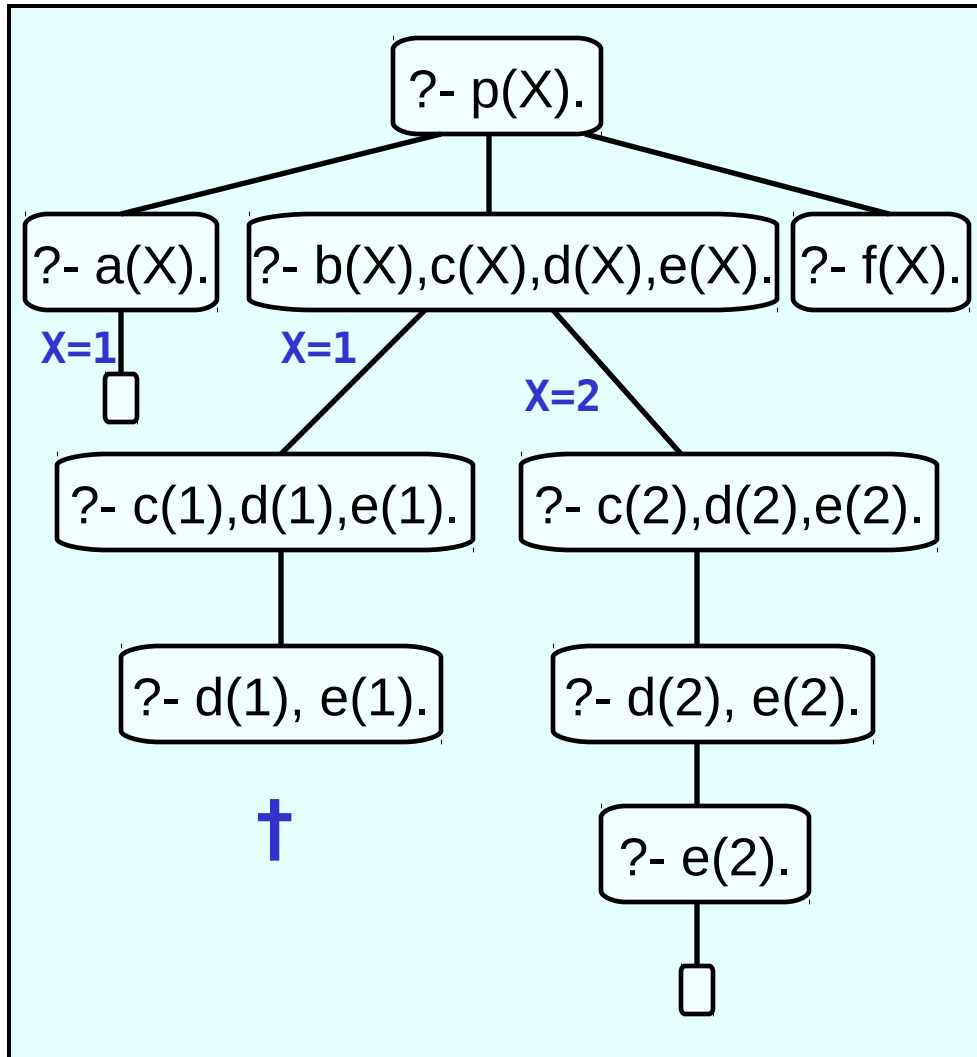
```
?- p(X).  
X=1;  
X=2
```



Ejemplo: código sin 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).
```

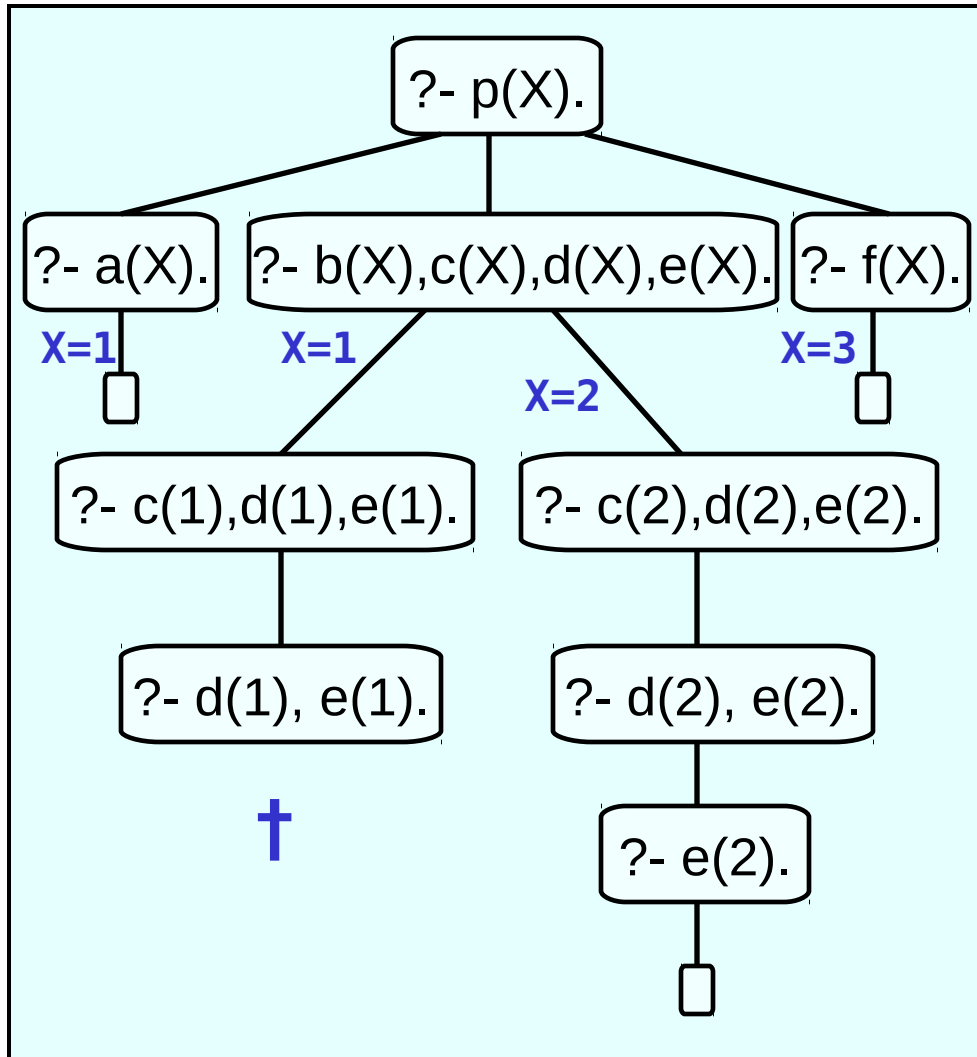
```
?- p(X).  
X=1;  
X=2;
```



Ejemplo: código sin 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).
```

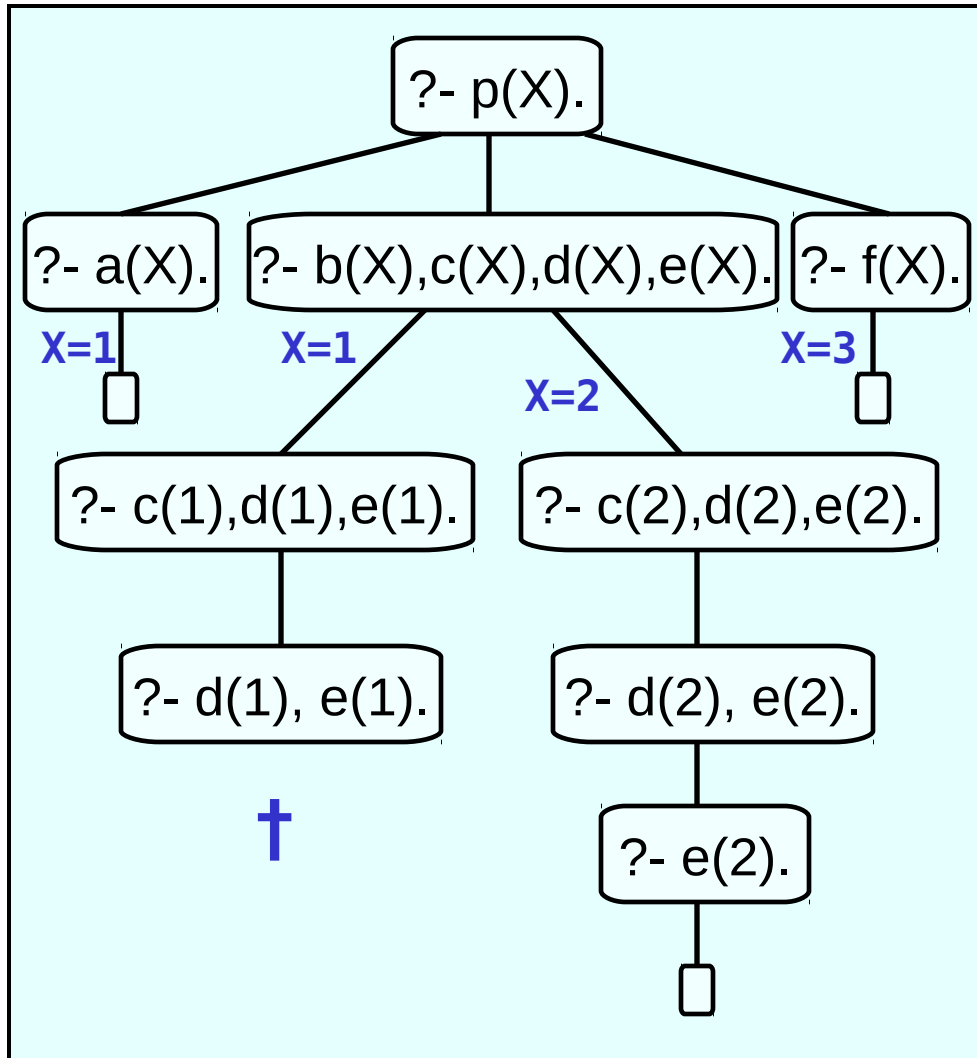
```
?- p(X).  
X=1;  
X=2;  
X=3
```



Example: cut-free code

```
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
```



Agregando un cut

- Supongamos que insertamos un cut en la 2da cláusula

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

- Si planteamos la misma consulta obtendremos la siguiente respuesta:

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

Ejemplo: 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).
```


Ejemplo: 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).
```

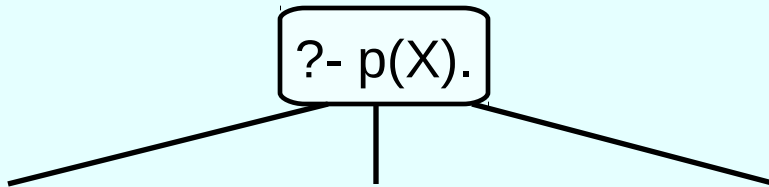
```
?- p(X).
```

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

Ejemplo: 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).
```

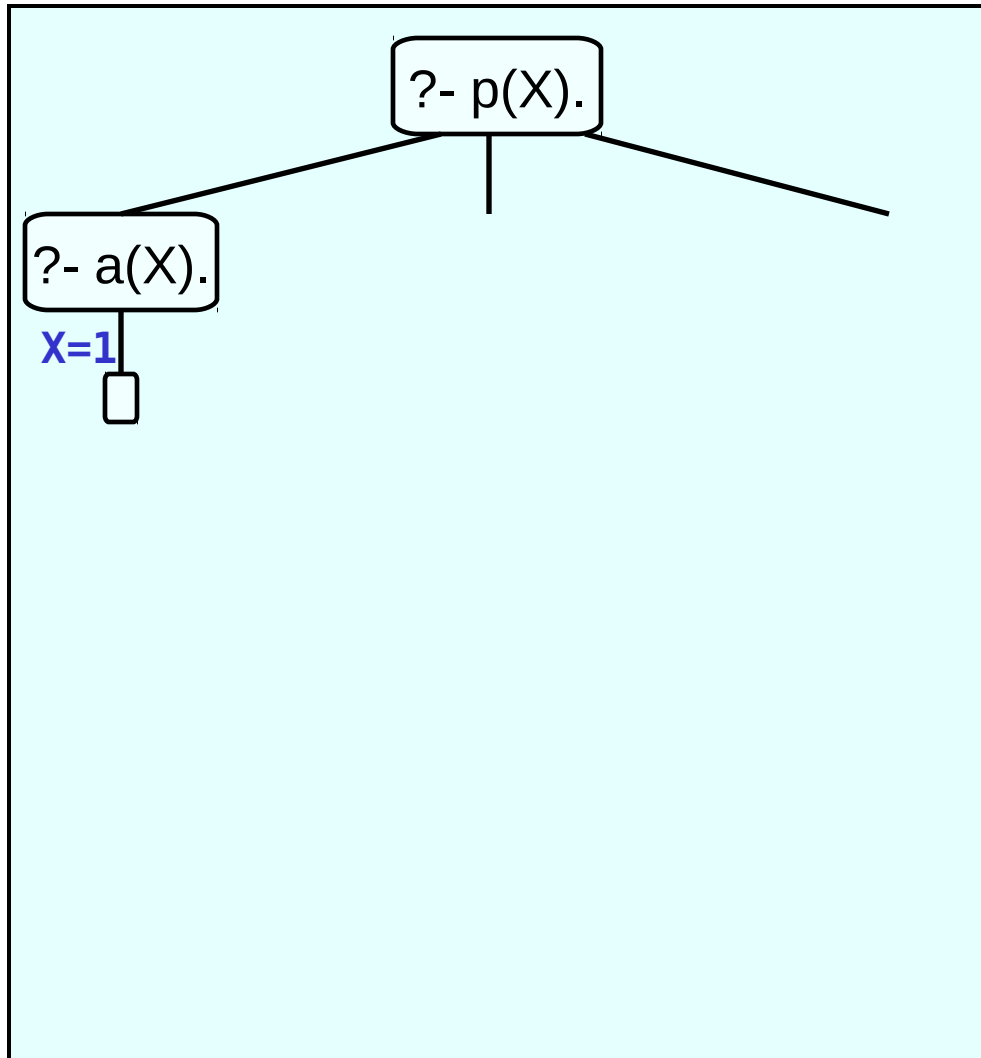


?- p(X).

Ejemplo: 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).
```

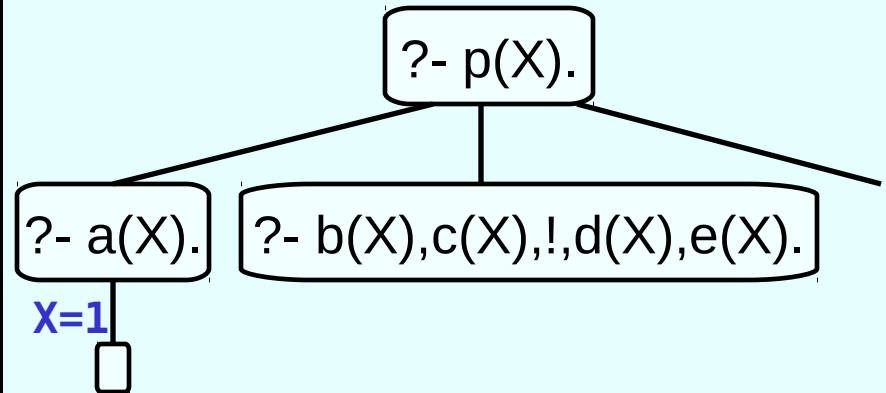
```
?- p(X).  
X=1
```



Ejemplo: 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).
```

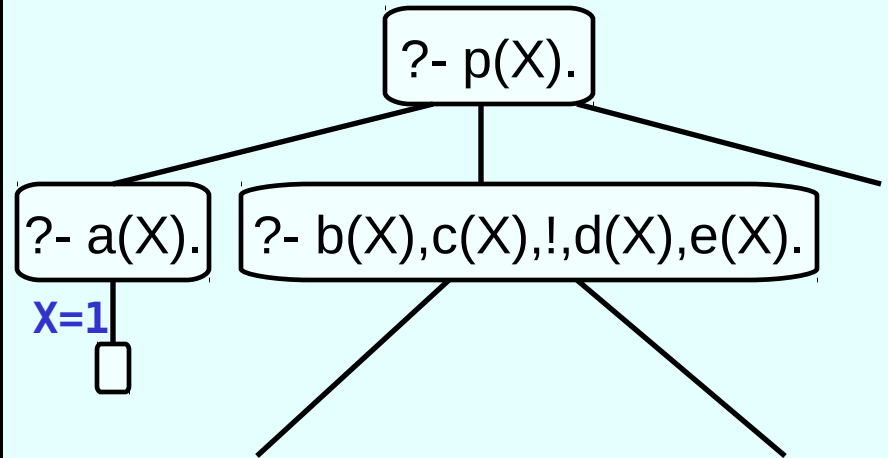
```
?- p(X).  
X=1;
```



Ejemplo: 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).
```

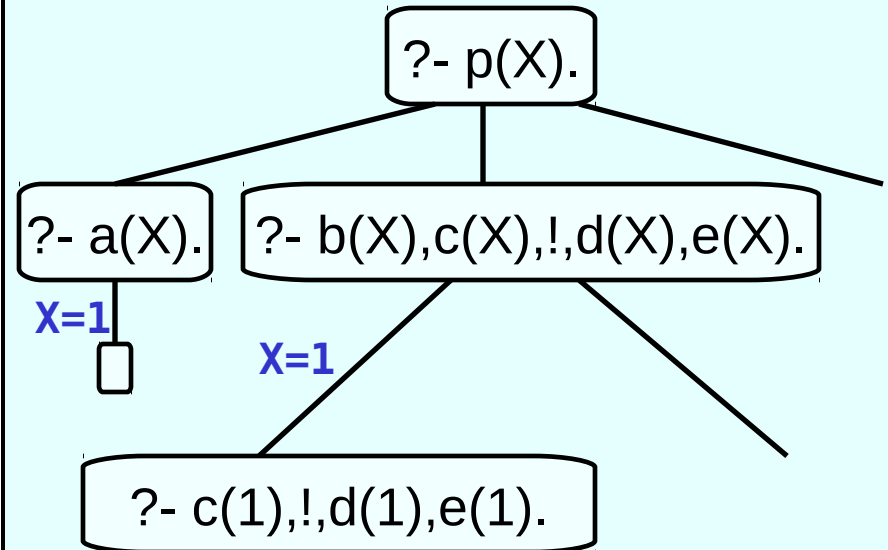
```
?- p(X).  
X=1;
```



Ejemplo: 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).
```

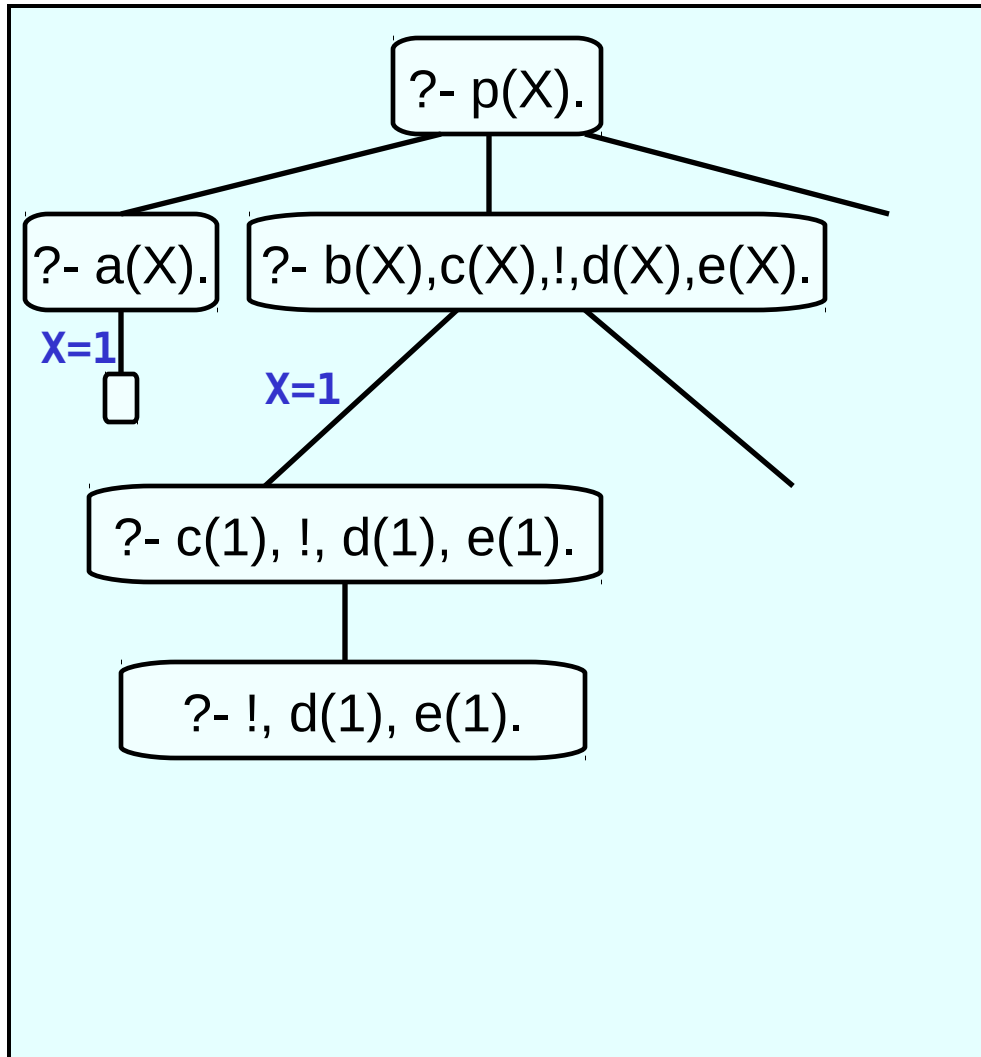
```
?- p(X).  
X=1;
```



Ejemplo: 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).
```

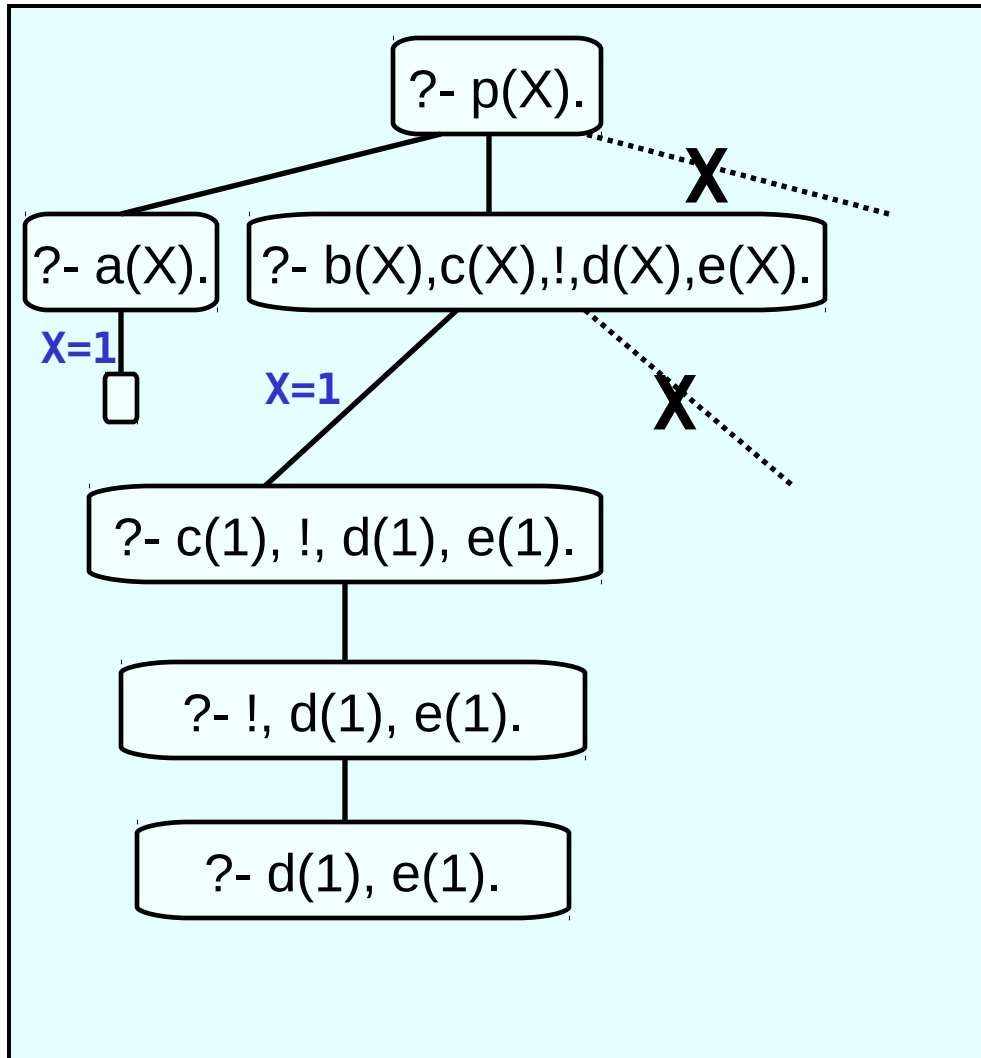
```
?- p(X).  
X=1;
```



Ejemplo: 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).
```

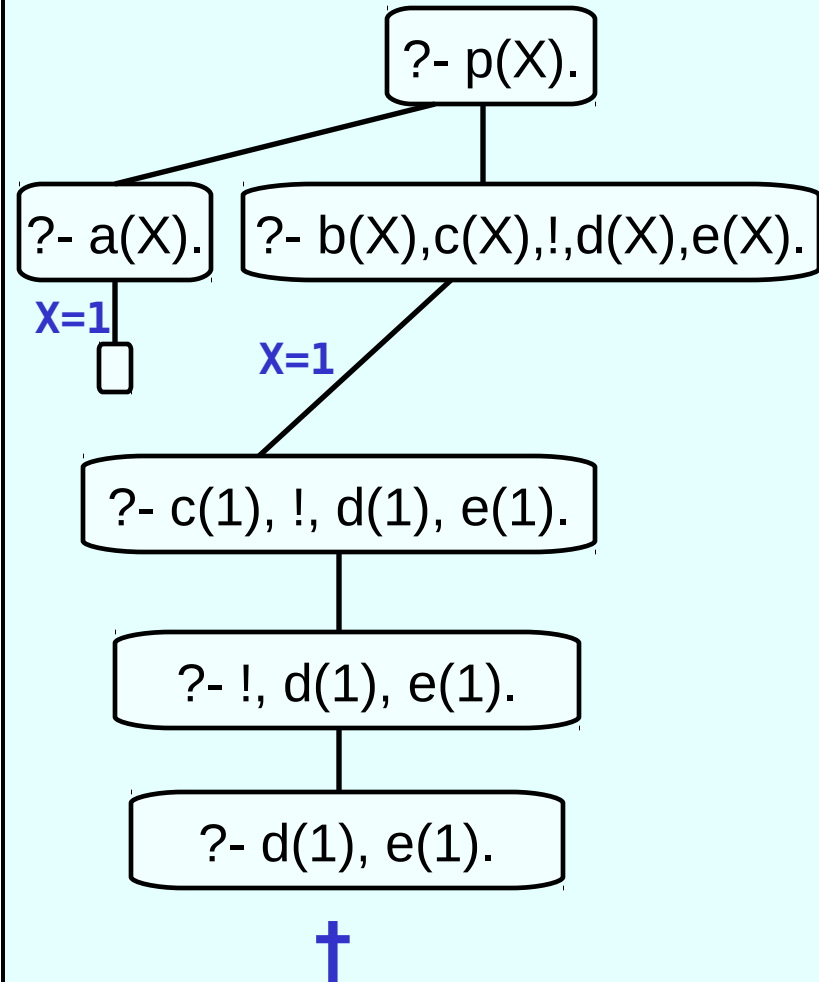
```
?- p(X).  
X=1;
```



Ejemplo: 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
```



Qué hace el cut

- El cut elimina puntos de opción en el árbol-
SLD
- Por ejemplo, en una regla de la forma $q:- p_1, \dots, p_n, !, r_1, \dots, r_n$.

el cut deja fijas las opciones en cuanto a:

- Esta cláusula particular de q
- Las opciones hechas por p_1, \dots, p_n
- NO afecta el funcionamiento en cuanto a r_1, \dots, r_n

Usos de Cut

- Considere el siguiente predicado max/3 que tiene éxito si el 3er argumento es el máximo de los 2 primeros.

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

Usos de Cut

- Considere el siguiente predicado max/3 que tiene éxito si el 3er argumento es el máximo de los 2 primeros.

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

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

```
yes
```

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

```
yes
```

Usos de Cut

- Considere el siguiente predicado max/3 que tiene éxito si el 3er argumento es el máximo de los 2 primeros.

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

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

```
no
```

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

```
no
```

El predicado max/3

- Cuál es el problema?
- Hay una ineficiencia potencial
 - Considerar la consulta `?- max(3,4,Y).`
 - Unifica correctamente Y con 4
 - Pero si pedimos más soluciones, tratará de satisfacer la 2da cláusula. Esto es completamente inútil!

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

max/3 con cut

- Con la ayuda del cut lo arreglamos fácil

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

- Funciona del siguiente modo:
 - Si $X \leq Y$ es exitoso, el cut fija la opción tomada para max, y la segunda cláusula de max/3 no se considera
 - Si $X \leq Y$ falla, Prolog considera la opción de la 2da cláusula

Cuts verdes

- Cuts que no cambian el significado de un programa se llaman **cuts verdes**
- El cut en $\text{max}/3$ es un ejemplo de cut verde:
 - el nuevo código tiene exactamente la misma respuesta que la versión anterior,
 - pero es más eficiente

Otro max/3 con cut

- Por qué no eliminar el cuerpo de la 2da cláusula? En principio, parece redundante.

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

- Está bien?

Otro max/3 con cut

- Por qué no eliminar el cuerpo de la 2da cláusula? En principio, parece redundante.

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

- Está bien?
 - ok

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

Otro max/3 con cut

- Por qué no eliminar el cuerpo de la 2da cláusula? En principio, parece redundante.

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

- Está bien?
 - ok

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

Otro max/3 con cut

- Por qué no eliminar el cuerpo de la 2da cláusula? En principio, parece redundante.

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

- Está bien?
 - oops....

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

max/3 con cut revisado

- Unificación después de pasar el cut

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

- Esta versión funciona

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

Cuts rojos

- Los cuts que cambian el significado de un programa son llamados **cuts rojos**
- El cut en $\text{max}/3$ revisado es un ejemplo de cut rojo:
 - Si sacamos el cut, no tenemos un programa equivalente
- Los programas que contienen cuts rojos
 - No son completamente declarativos
 - Pueden ser difíciles de entender
 - Pueden involucrar errores sutiles, difíciles de percibir

Otro predicado del sistema: **fail/0**

- Como lo sugiere el nombre, este es un predicado que siempre falla.
- No parece muy útil !!
- Pero:
 - cuando Prolog falla, hace backtracking

!,fail

```
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).
```

- La combinación cut,fail permite codificar excepciones

!,fail

```
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).
```

- La combinación cut,fail permite codificar excepciones

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

!,fail

```
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).
```

- La combinación cut,fail permite codificar excepciones

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

!,fail

```
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).
```

- La combinación cut,fail permite codificar excepciones

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

!,fail

```
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).
```

- La combinación cut,fail permite codificar excepciones

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

Negación por falla

- La combinación cut,fail nos da algún modo de negación
- La llamamos negación por falla, y su definición es:

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

Negación por falla

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

- En términos del árbol-SLD, la negación es exitosa cuando el árbol es un árbol **finitamente fallado**

Negación por falla

```
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).
```

Negación por falla

```
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
```


Otro predicado del sistema: \+

- La negación por falla se usa a menudo, está predefinida
- En Prolog standard el operador prefijo \ + significa negación por falla
- Podríamos codificar entonces:

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

```
?- enjoys(vincent,X).
```

```
X=a
```

```
X=c
```

```
X=d
```

Negación por falla y lógica

- La negación por falla es distinta de la negación lógica
- Si cambiamos el orden de los objetivos en el cuerpo, tenemos una conducta distinta

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

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

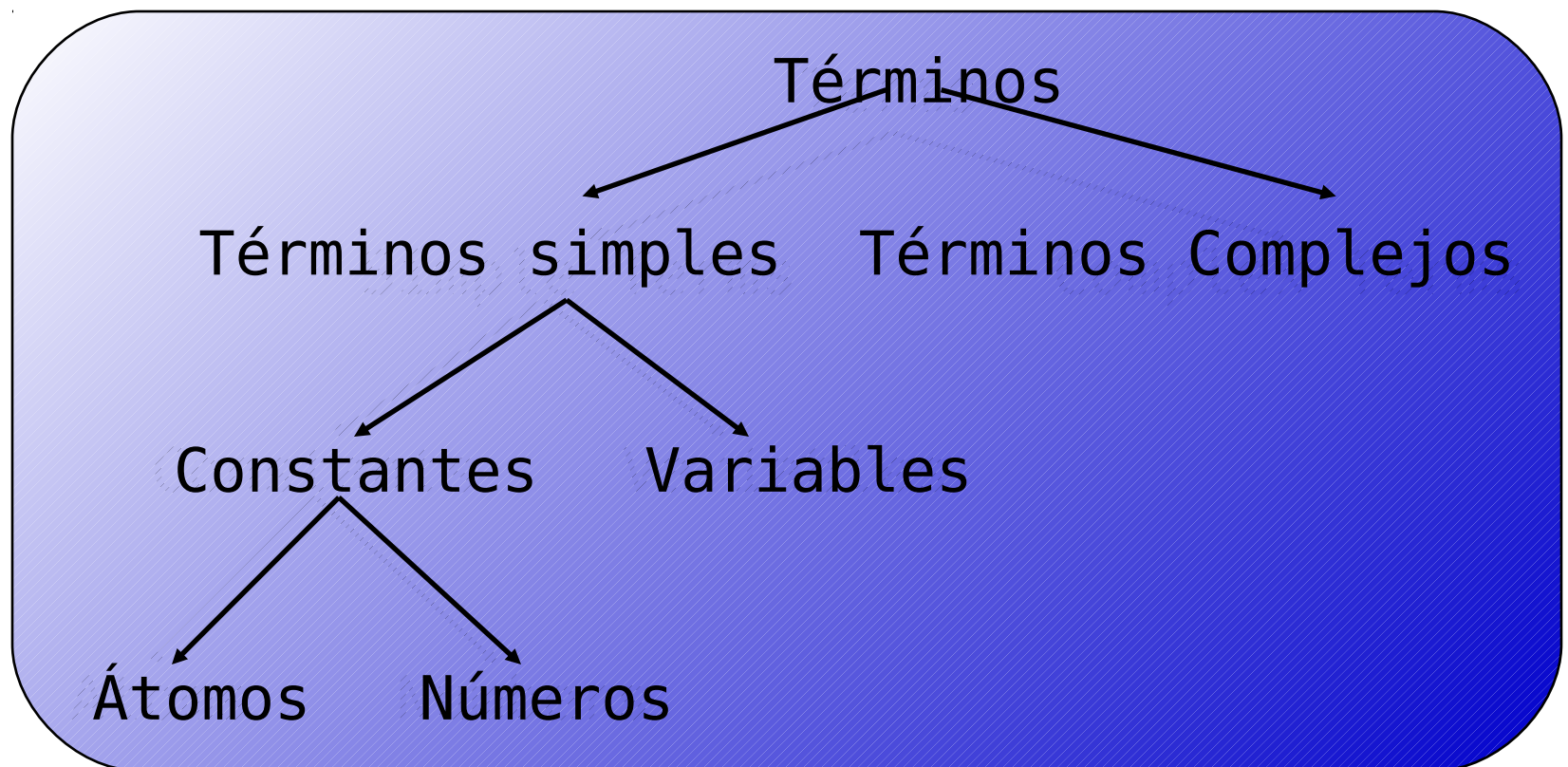
Negación por falla

- Algunos ejemplos

$\text{disjuntos}(+X, +Y) \leftarrow$ Las listas X y Y representan 2 conjuntos disjuntos

diferencia de conjuntos

Términos Prolog



Chequeando el tipo de un término

atom/1	<i>El argumento es un átomo?</i>
integer/1	<i>... un entero?</i>
float/1	<i>... un número en pto. flotante?</i>
number/1	<i>... un entero o pto. flotante?</i>
atomic/1	<i>... una constante?</i>
var/1	<i>... una variable sin instanciar?</i>
nonvar/1	<i>... una variable instanciada u otro término que no es una variable?</i>

atom/1

?- atom(a).

yes

?- atom(7).

no

?- atom(X).

no

atom/1

?- X=a, atom(X).

X = a

yes

?- atom(X), X=a.

no

atomic/1

?- atomic(mia).

yes

?- atomic(5).

yes

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

no

var/1

?- var(mia).

no

?- var(X).

yes

?- X=5, var(X).

no

nonvar/1

?- nonvar(X).

no

?- nonvar(mia).

yes

?- nonvar(23).

yes

La estructura de los términos

- Dado un término complejo, existen predicados del sistema que permiten reconocer:
 - El functor
 - La aridad
 - Los argumentos

El predicado functor/3

- El predicado functor/3 da el functor y la aridad de un término complejo.

El predicado functor/3

- Nos da el functor y la aridad de un término complejo
- functor(Term,F,Aridad)
 - ?- functor(friends(lou,andy),F,A).
F = friends
A = 2
yes

El predicado functor/3

- Nos da el functor y la aridad de un término complejo
- functor(Term,F,Aridad)
 - ?- functor([lou,andy,vicky],F,A).
F = .
A = 2
yes

functor/3 y constantes

- Qué pasa si usamos functor con constantes?

functor/3 y constantes

- Qué pasa si usamos functor con constantes?
 - ?- functor(mia,F,A).
 - F = mia
 - A = 0
 - yes

functor/3 y constantes

- Qué pasa si usamos functor con constantes?

❑?- functor(mia,F,A).

F = mia

A = 0

yes

❑?- functor(14,F,A).

F = 14

A = 0

yes

functor/3 para construir términos

- Se puede usar también para construir términos
 - ?- functor(Term,friends,2).
Term = friends(__,__)
yes

Chequeo de término complejo

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

Argumentos: arg/3

- Arg/3 aísla los argumentos de un término complejo
- Tres argumentos:
 - Un natural N
 - Un término complejo T
 - El N -ésimo argumento de T

Argumentos: arg/3

- Arg/3 aísla los argumentos de un término complejo
- Tres argumentos:
 - Un natural N
 - Un término complejo T
 - El N -ésimo argumento de T

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

El predicado univ =../2

- Es infijo
- Terminó =.. [Functor,Arg1,Arg2,...]
- Se satisface si la lista del lado derecho tiene como 1er elemento el functor y luego los argumentos, en orden.