Unification et résolution
ELFE, séance 2

- Sujets:
  - unification
  - algorithme d'unification de Prolog
  - recherche de preuve de Prolog
    - arbre de résolution

- Exos
  - LPN, chapitre 2

Buts

- Discuter **l'unification** de Prolog
  - Montrer comment l'unification de Prolog diffère de l'unification habituelle en logique formelle

- Expliquer la **stratégie de recherche** de Prolog dans la tentative de déduire des nouvelles information de précédentes, en appliquant le *modus ponens*.

L’unification

- Revenons sur l’exemple où nous disions que

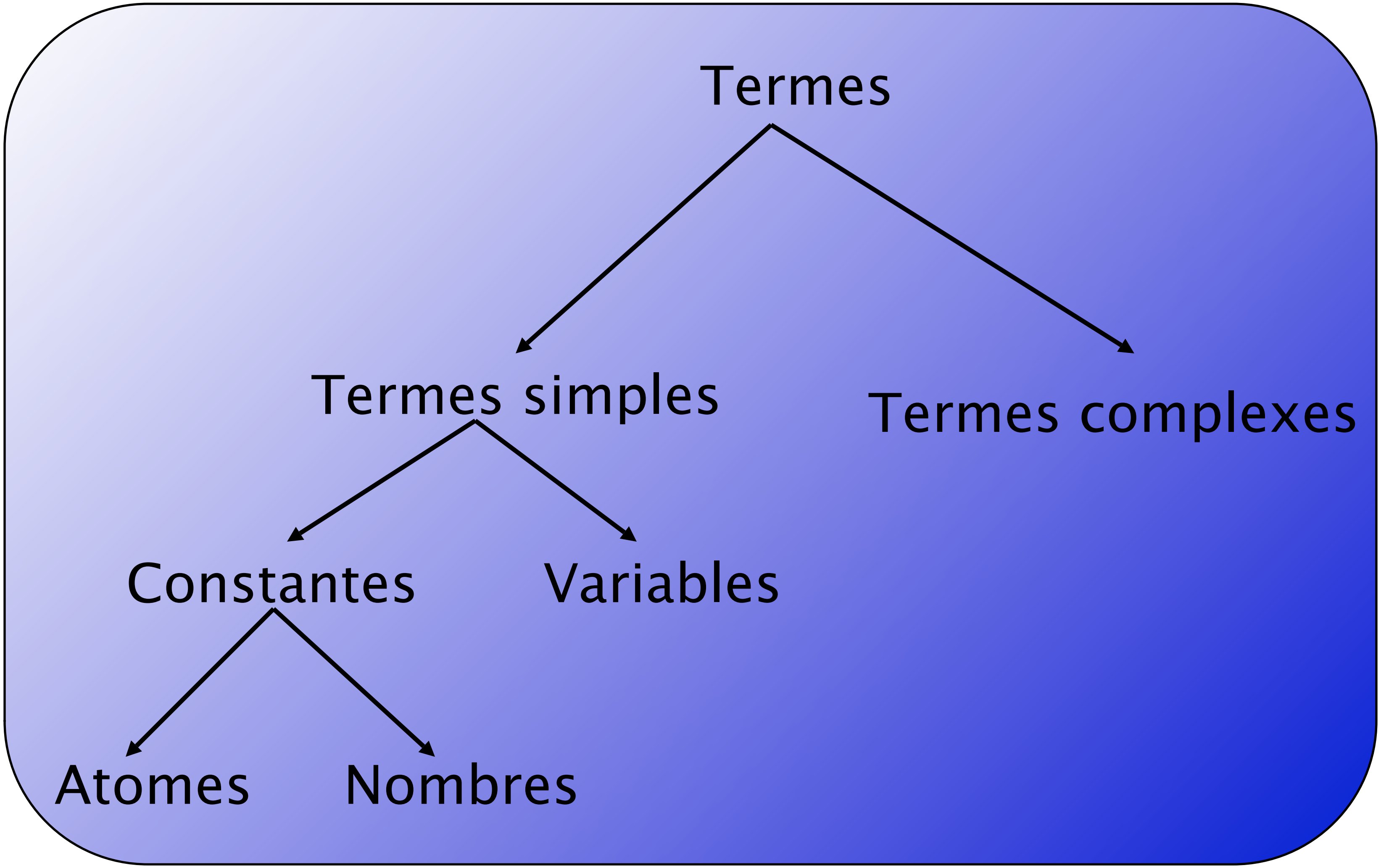
  femme(X)

  unifie avec

  femme(mia)

  en substituant la variable **X** par l’atome **mia**.

Les termes de Prolog



Définition des termes en forme Backus–Naur (BNF)

foncteur arguments

terme ::= atome(terme,...,terme) | variable | constante

constante ::= atome | nombre

Unification (définition préliminaire)

- Deux termes **unifient** si
  - ils sont le même terme , ou
  - s'ils contiennent des variables qui peuvent uniformément être substitués par des termes tel que les termes résultants soient égaux.

Unification: exemples 1

- Exemples:
  - **mia** et **mia** unifient
  - **42** et **42** unifient
  - **femme(mia)** et **femme(mia)** unifient

- Puis:
  - **vincent** et **mia** n'unifient pas
  - **femme(mia)** et **femme(jody)** n'unifient non plus

Unification: exemples 2

- Que faire pour ces termes:
  - **mia** et **X**
  - **femme(Z)** et **femme(mia)**

Unification: exemples 2

- Que faire pour ces termes:
  - **mia** et **X**
  - **femme(Z)** et **femme(mia)**
  - **aime(mia,X)** et **aime(X,vincent)**

Substitutions

- En unifiant deux termes Prolog fait toutes les substitutions nécessaires, tel qu'ils deviennent égaux.
- Ceci rend l'unification un mécanisme de programmation puissant

Définition unification 1/4

Si $terme_1$ et $terme_2$ sont des constantes, alors $terme_1$ et $terme_2$ unifient si et seulement si ils sont le même atome, ou le même nombre.

Définition unification 2/4

Si **terme$_1$** est une variable et **terme$_2$** est n'importe quel type de terme, alors **terme$_1$** et **terme$_2$** unifient, et **terme$_1$** est substitué par **terme$_2$**. De même, si terme$_2$ est une variable et terme$_1$ est n'importe quel type de terme, alors terme$_1$ et terme$_2$ unifient et terme$_2$ est substitué par terme$_1$. (Donc s'ils sont tous deux des variables, ils sont tous deux instantiés l'un à l'autre et on dit qu'il y a partage de valeur)

Définition unification 3/4

Si **terme**$_1$ et **terme**$_2$ sont des termes complexes, alors ils unifient si et seulement si:

(a) Ils ont le même foncteur et la même arité

(b) tous leurs arguments correspondants unifient

(c) les substitutions de variables sont compatibles (par exemple il n'est pas possible de substituer la variable **X** par **mia** quand on unifie une paire d'arguments, et de substituer en même temps **X** par **vincent** quand on unifie une autre paire d'arguments)

Définition unification 4/4

Deux termes unifient si et seulement si il découle des trois affirmations précédentes qu'ils unifient.

Définition revue 2/4

1. Si T_1 et T_2 sont des constantes, alors T_1 et T_2 unifient si ils sont le même atome, ou le même nombre.
2. Si T_1 est une variable et T_2 un terme quelconque, alors T_1 et T_2 unifient en substituant T_1 par T_2 (et l'inverse).

Définition revue 3/4

1. Si T_1 et T_2 sont des constantes, alors T_1 et T_2 unifient si ils sont le même atome, ou le même nombre.
2. Si T_1 est une variable et T_2 un terme quelconque, alors T_1 et T_2 unifient en substituant T_1 par T_2 (et l'inverse).
3. Si T_1 et T_2 sont des termes complexes alors ils unifient si:
   a) Ils coïncident en foncteur et arité, et que
   b) tous leurs arguments correspondants unifient et
   c) les substitutions de variables sont compatibles.

Définition revue 4/4

1. (comme avant)
2. (comme avant)
3. (comme avant)
4. T_1 et T_2 n'unifient en aucun autre cas

Unification en Prolog =/2

```
?- mia = mia.
yes
?-
```

Unification en Prolog =/2

```
?- mia = mia.
yes
?- mia = vincent.
no
?-
```

Unification en Prolog =/2

```
?- mia = X.
X=mia
yes
?-
```

Que répondra Prolog?

```
?- X=mia, X=vincent.
```

Que répondra Prolog?


```
?- X=mia, X=vincent.
no
?-
```

Pourquoi? Après avoir traité le premier but, Prolog a substitué X par **mia**, et ne peut plus ensuite l'unifier avec **vincent**. Donc, le deuxième but échoue.

Exemple avec termes complexes

```
?- k(s(g),Y) = k(X,t(k)).
```

Exemple avec termes complexes

```
?- k(s(g),Y) = k(X,t(k)).
X=s(g)
Y=t(k)
yes
?-
```

Exemple avec termes complexes

?- k(s(g),t(k)) = k(X,t(Y)).

Exemple avec termes complexes

```
?- k(s(g),t(k)) = k(X,t(Y)).
X=s(g)
Y=k
yes
?-
```

Dernier exemple

?- aime(X,X) = aime(marsellus,mia).

Exo 2.1 de LPN

1. pain = pain
2. ‘Pain’ = pain
3. ‘pain’ = pain
4. Pain = pain
5. pain = saucisse
6. nourriture(pain)=pain
7. nourriture(pain)=X
8. nourriture(X)=nourriture(pain)

Exo 2.1 de LPN

9. nourriture(pain,X) = nourriture(X,saucisse)
10. nourriture(pain,X,bière) =nourriture(Y,saucisse,X)
11. nourriture(pain,X,bière) =nourriture(Y,kahuna_burger)
12. nourriture(X) = X
13. repas(nourriture(pain),boisson(bière))=repas(X,Y)
14. repas(nourriture(pain),X)=repas(X,boisson(bière))

Solutions

1. `bread = bread` unifies.
2. `'Bread' = bread` doesn't unify.
3. `'bread' = bread` unifies.
4. `Bread = bread` unifies; the variable `Bread` gets instantiated with the atom `bread`.
5. `bread = sausage` doesn't unify.
6. `food(bread) = bread` doesn't unify.
7. `food(bread) = X` unifies; `X` gets instantiated with `food(bread)`.
8. `food(X) = food(bread)` unifies; `X` gets instantiated with `bread`.
9. `food(bread,X) = food(Y,sausage)` unifies; `X` gets instantiated with `sausage` and `Y` gets instantiated with `bread`.
10. `food(bread,X,beer) = food(Y,sausage,X)` doesn't unify; `X` cannot be instantiated with `sausage` as well as `beer`.

Solutions

11. `food(bread,X,beer) = food(Y,kahuna burger)` doesn't unify; the functors are of different arity.

12. `food(X) = X` is trickier. According to the basic definition of unification given in the text, these two terms do not unify, as no matter what (finite) term we instantiate `X` to, the two sides won't be identical. However (as we mentioned in the text) modern Prolog interpreters will detect that there is a problem here and will instantiate `X` with the 'infinite term' `food(food(food(...)))`, and report that unification succeeds. In short, there is no 'correct' answer to this question; it's essentially a matter of convention. The important point is to understand why such unifications need to be handled with care.

13. `meal(food(bread),drink(beer)) = meal(X,Y)` unifies; `X` gets instantiated with `food(bread)` and `Y` with `drink(beer)`.

14. `meal(food(bread),X) = meal(X,drink(beer))` doesn't unify; `X` cannot get instantiated twice with different things

Prolog et l'unification

- Prolog n'utilise *pas* un algorithme d'unification standard
- Considérez la requête suivante:

  ?- pere(X) = X.

- Ces termes peuvent-ils unifier ou non?

Termes infinis

?- pere(X) = X.

X=pere(pere(pere(pere(pere(pere(pere(p
ere(pere(pere(pere(pere(pere(pere(per
e(pere(pere(pere(pere(pere(pere(pere(
pere(pere(pere(pere(pere(pere(pere(pe
re(pere(pere(pere(pere(pere(pere(pere(
pere(pere(pere(pere(pere(pere(pere(pe
re(pere(pere(pere(

Termes infinis

```
?- pere(X) = X.
X=pere(pere(pere(…))))
yes
?-
```

Occurs check

- Les algorithmes d’unification courants font un test d'occurrence
- Quand on lui demande d’unifier une variable avec un autre terme Prolog contrôle si la variable apparaît dans le terme
- En Prolog:

```
?- unify_with_occurs_check(pere(X), X).
no
```

Programmation par unification

```
vertical( ligne(point(X,Y),
                point(X,Z))).

horizontal(ligne(point(X,Y),
                 point(Z,Y))).
```

Programmation par unification


```
vertical( ligne(point(X,Y),
                point(X,Z))).

horizontal( ligne(point(X,Y),
                  point(Z,Y))).
```

```
?-
```

Programmation par unification


```
vertical( ligne(point(X,Y),
                point(X,Z))).

horizontal( ligne(point(X,Y),
                  point(Z,Y))).
```

```
?- vertical(ligne(point(1,1),point(1,3))).
yes
?-
```

Programmation par unification


```
vertical( ligne(point(X,Y),
                point(X,Z))).

horizontal( ligne(point(X,Y),
                  point(Z,Y))).
```

```
?- vertical(ligne(point(1,1),point(1,3))).
yes
?- vertical(ligne(point(1,1),point(3,2))).
no
?-
```

Programmation par unification


```
vertical( ligne(point(X,Y),
                point(X,Z))).

horizontal( ligne(point(X,Y),
                  point(Z,Y))).
```

```
?- horizontal(ligne(point(1,1),point(1,Y))).
Y = 1;
no
?-
```

Programmation par unification

```
vertical( ligne(point(X,Y),
                point(X,Z))).

horizontal( ligne(point(X,Y),
                  point(Z,Y))).
```

```
?- horizontal(ligne(point(2,3),Point)).
Point = point(_554,3);
no
?-
```

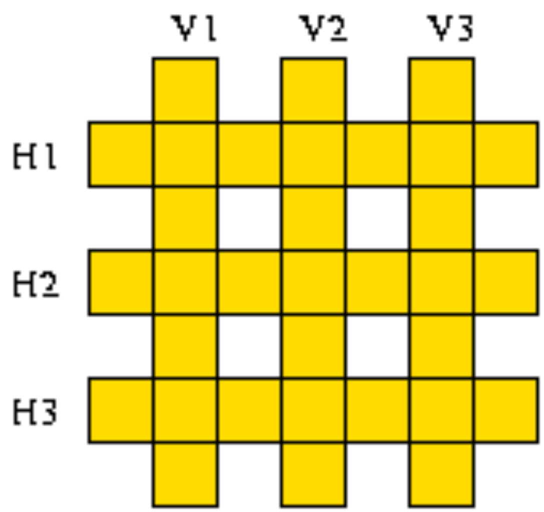
Exo: mots croisés

Exercise 2.4 (du livre LPN) Voilà six mots anglais:

abalone, *abandon*, *anagram*, *connect*, *elegant*, *enhance*.

Ils devront être arrangés sur une grille du style mots croisés comme:

Voilà nos mots comme base de connaissances:

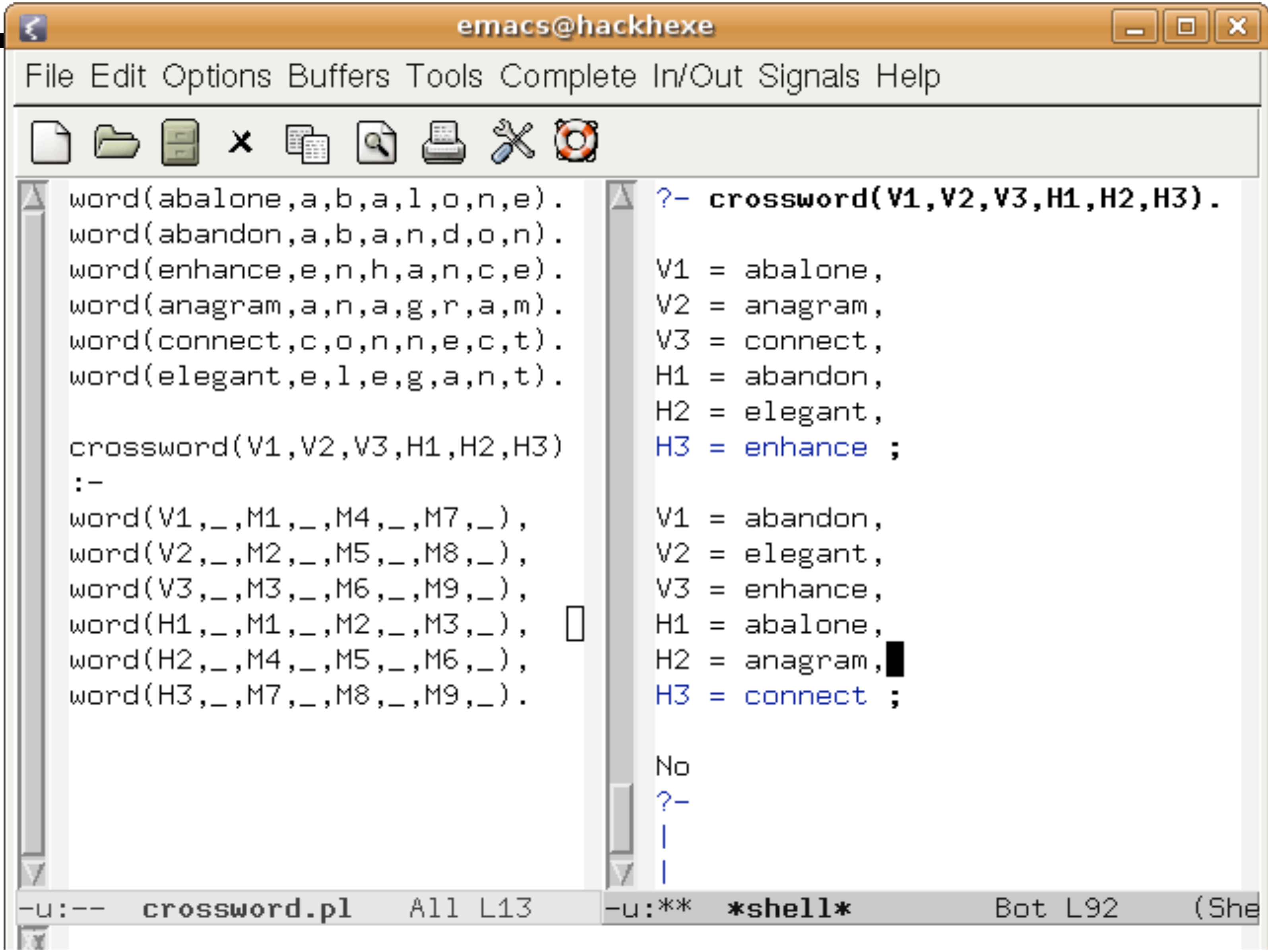



```
mot(abalone,a,b,a,l,o,n,e).
mot(abandon,a,b,a,n,d,o,n).
mot(enhance,e,n,h,a,n,c,e).
mot(anagram,a,n,a,g,r,a,m).
mot(connect,c,o,n,n,e,c,t).
mot(elegant,e,l,e,g,a,n,t).
```

Ecrivez un prédicat `crossword/6` qui nous indique

comment remplir la grille, c.a.d. les trois premiers arguments seront les mots verticaux de gauche à droite, les trois arguments suivants les mots horizontaux de haut en bas.

Programme mots croisés



Recherche de preuve

- Maintenant que nous connaissons l'unification, nous pouvons apprendre comment Prolog traverse une base de connaissances pour déterminer si une requête est satisfaite.
- Autrement dit: nous sommes prêts à étudier la recherche de preuve.

Exemple

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

```
?- k(Y).
```

Exemple: arbre de résolution

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

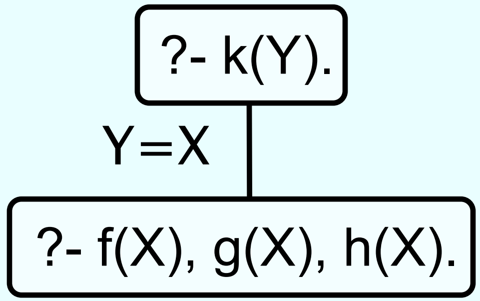
```
?- k(Y).
```

```
?- k(Y).
```

Exemple: arbre de résolution

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

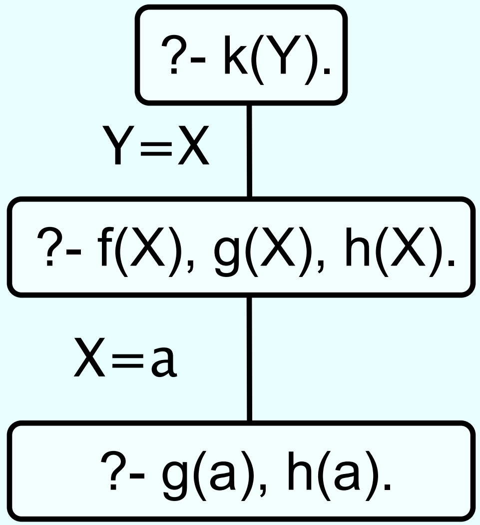
```
?- k(Y).
```



Exemple: arbre de résolution


```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

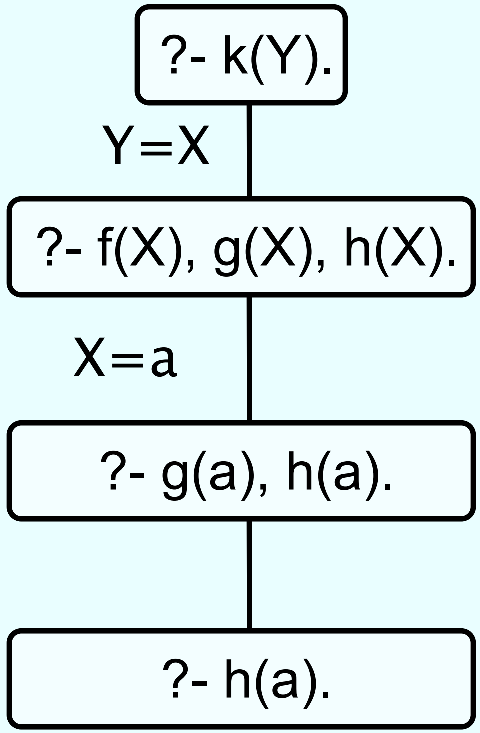
```
?- k(Y).
```



Exemple: arbre de résolution

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

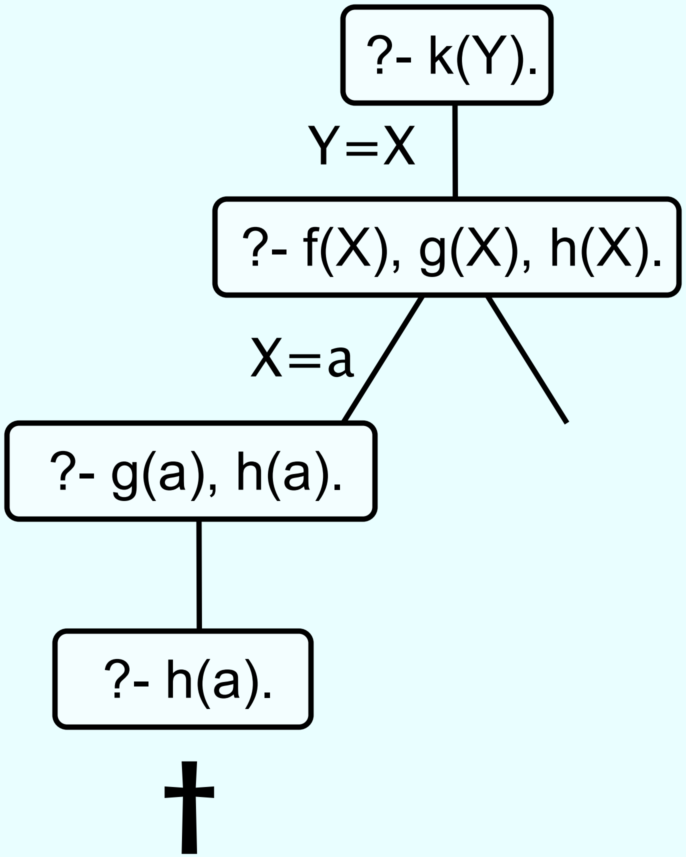
```
?- k(Y).
```



Exemple: arbre de résolution

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

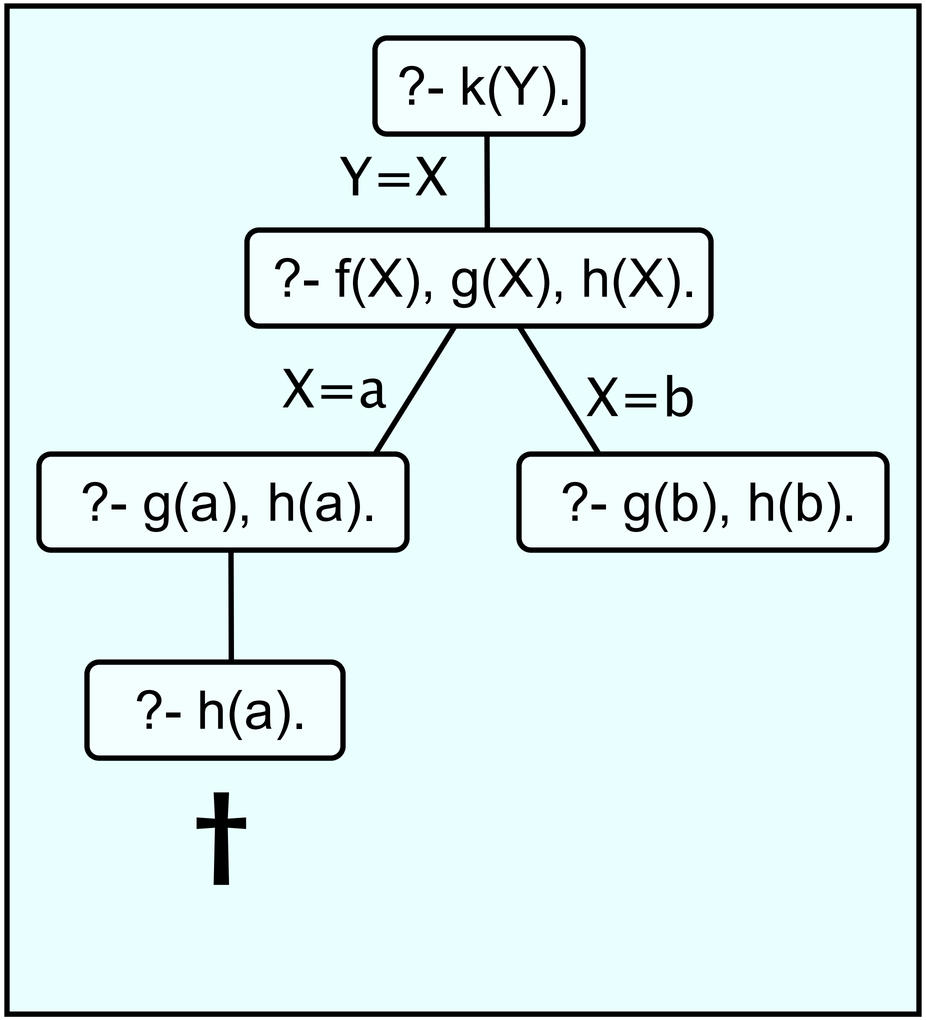
```
?- k(Y).
```



Exemple: arbre de résolution

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

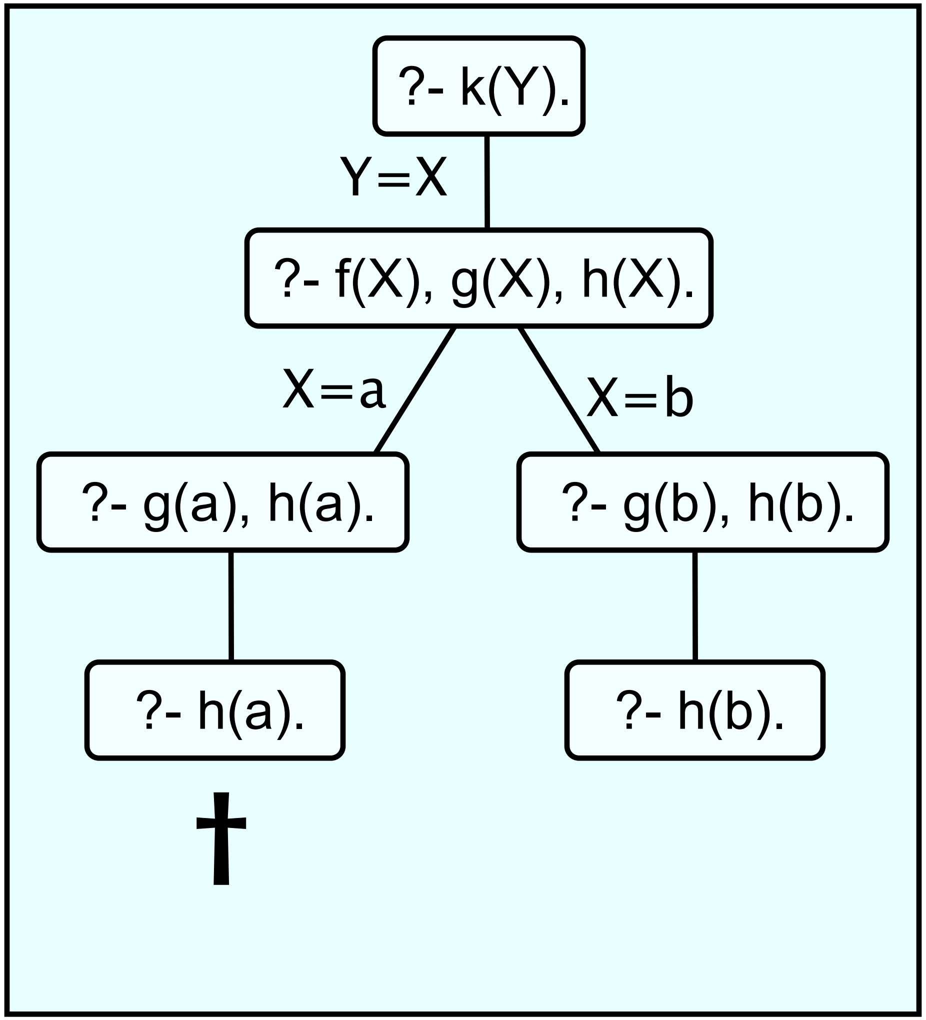
```
?- k(Y).
```



Exemple: arbre de résolution


```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

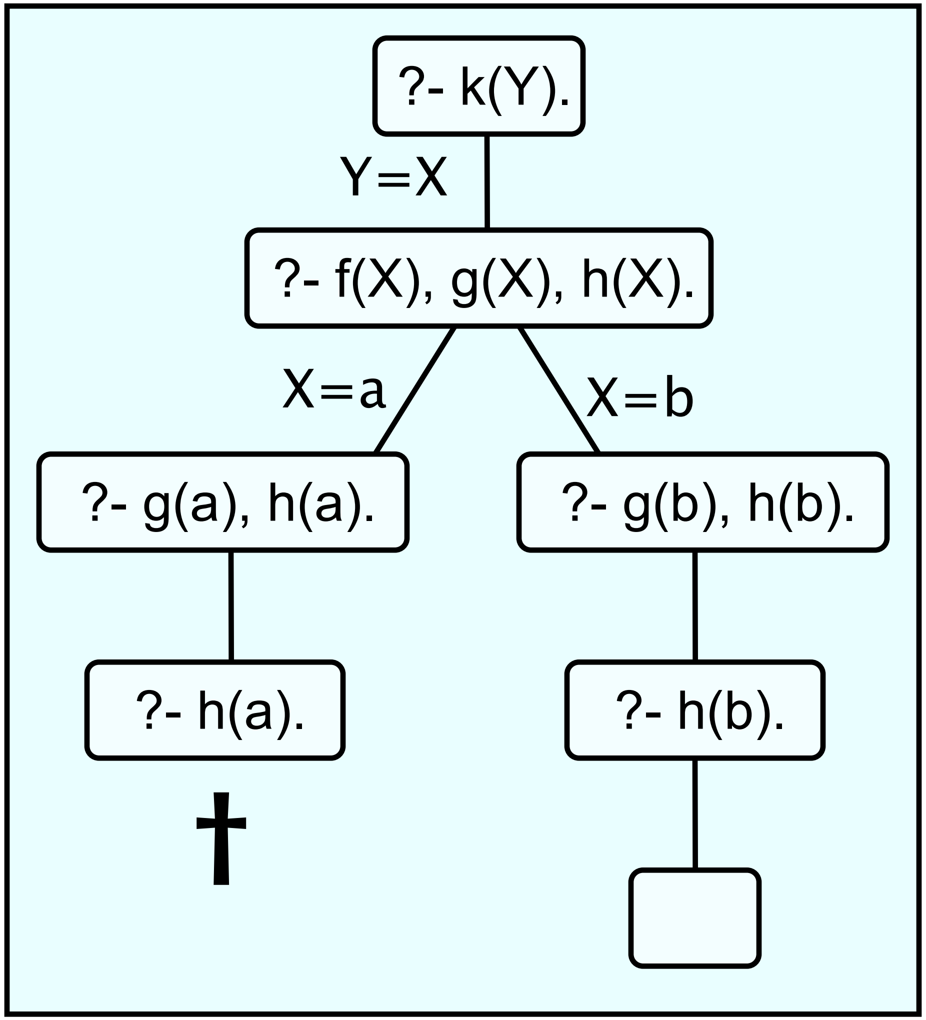
```
?- k(Y).
```



Exemple: arbre de résolution

f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).

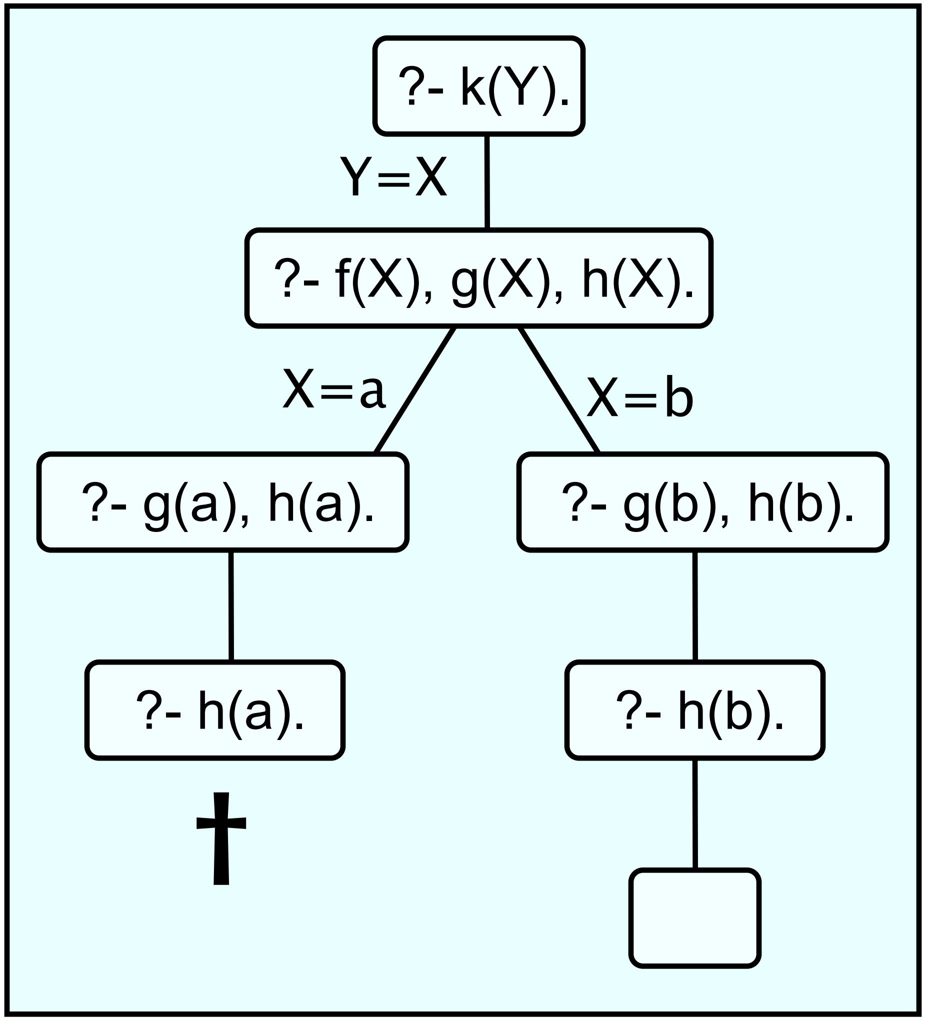
?- k(Y).
Y=b



Exemple: arbre de résolution


```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

```
?- k(Y).
Y=b;
no
?-
```



Autre exemple


```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
?- jaloux(X,Y).
```

Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
?- jaloux(X,Y).
```

```
?- jaloux(X,Y).
```

Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
?- jaloux(X,Y).
```

?- jaloux(X,Y).

X=A Y=B

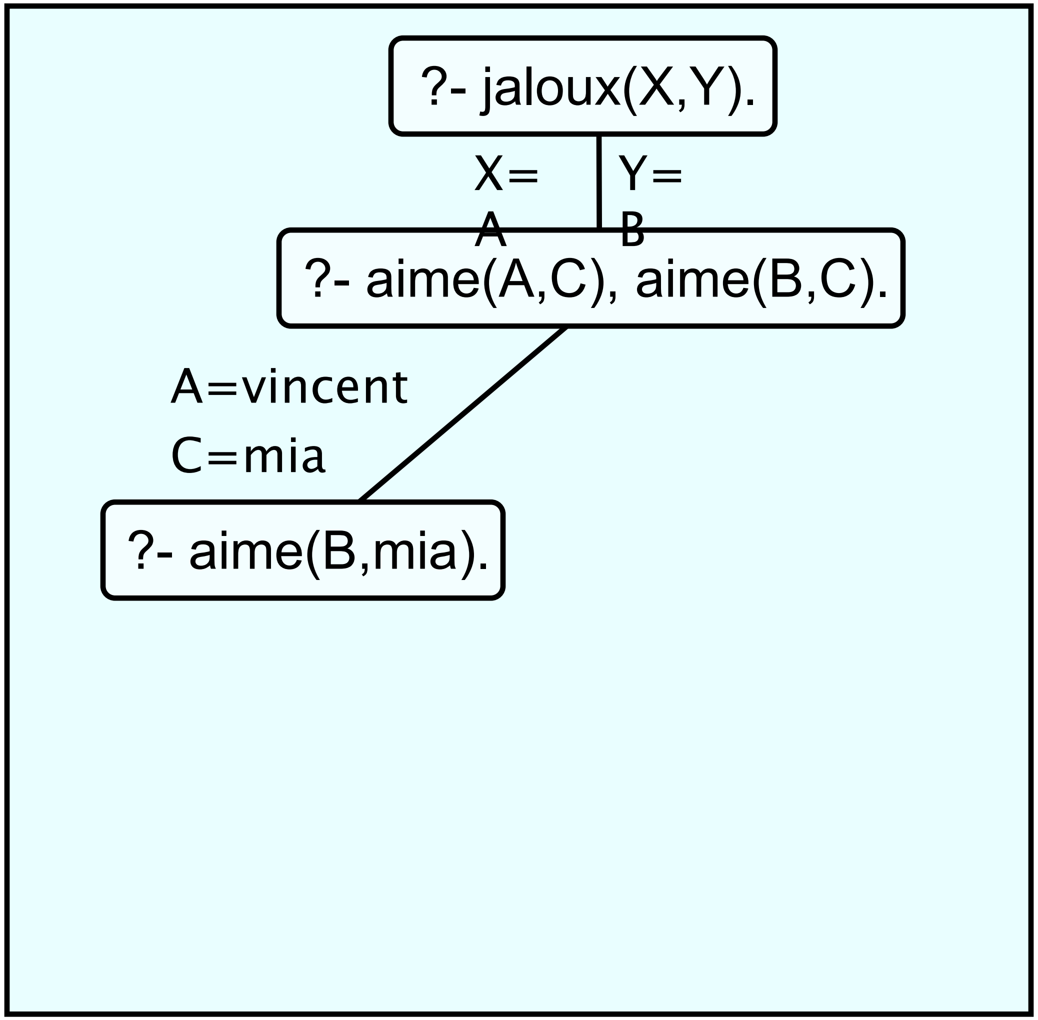
?- aime(A,C), aime(B,C).

Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
?- jaloux(X,Y).
```

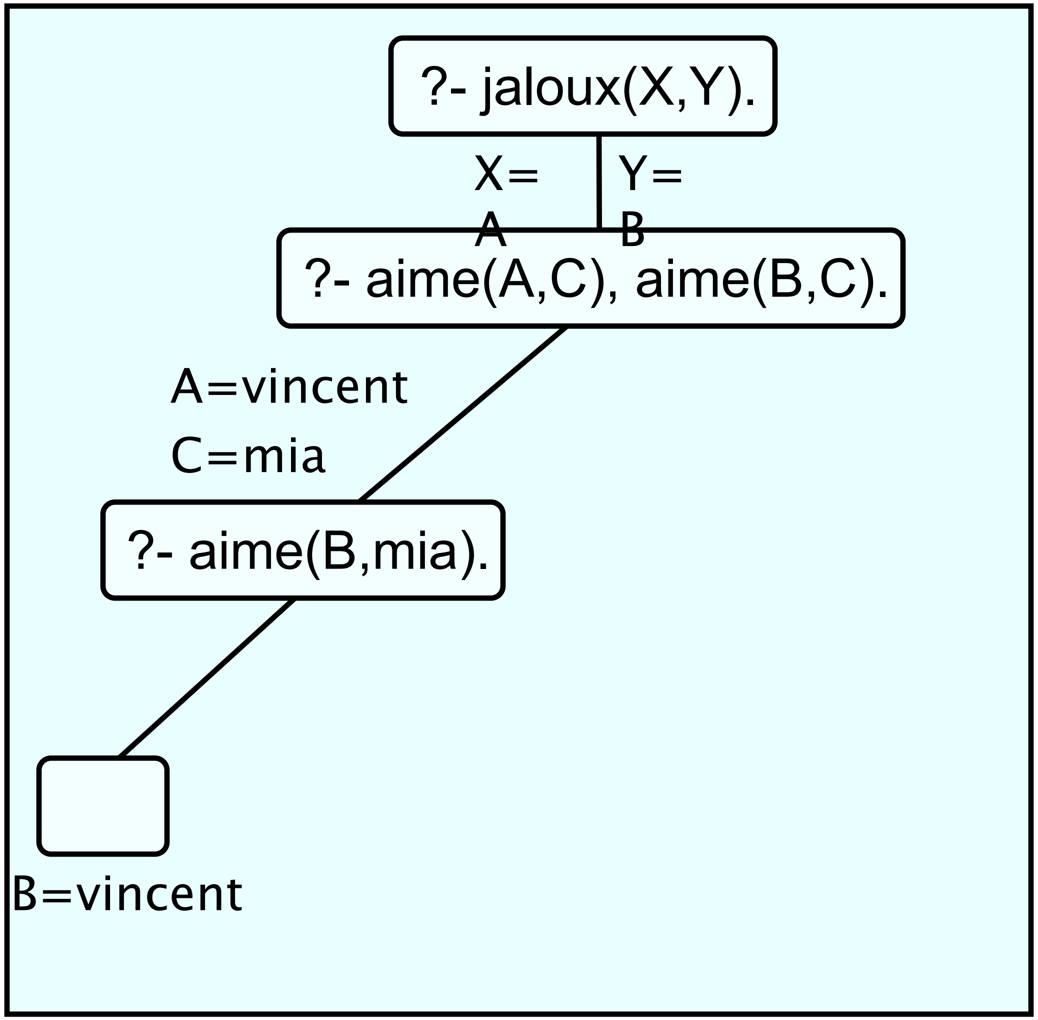


Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
?- jaloux(X,Y).
X=vincent
Y=vincent
```

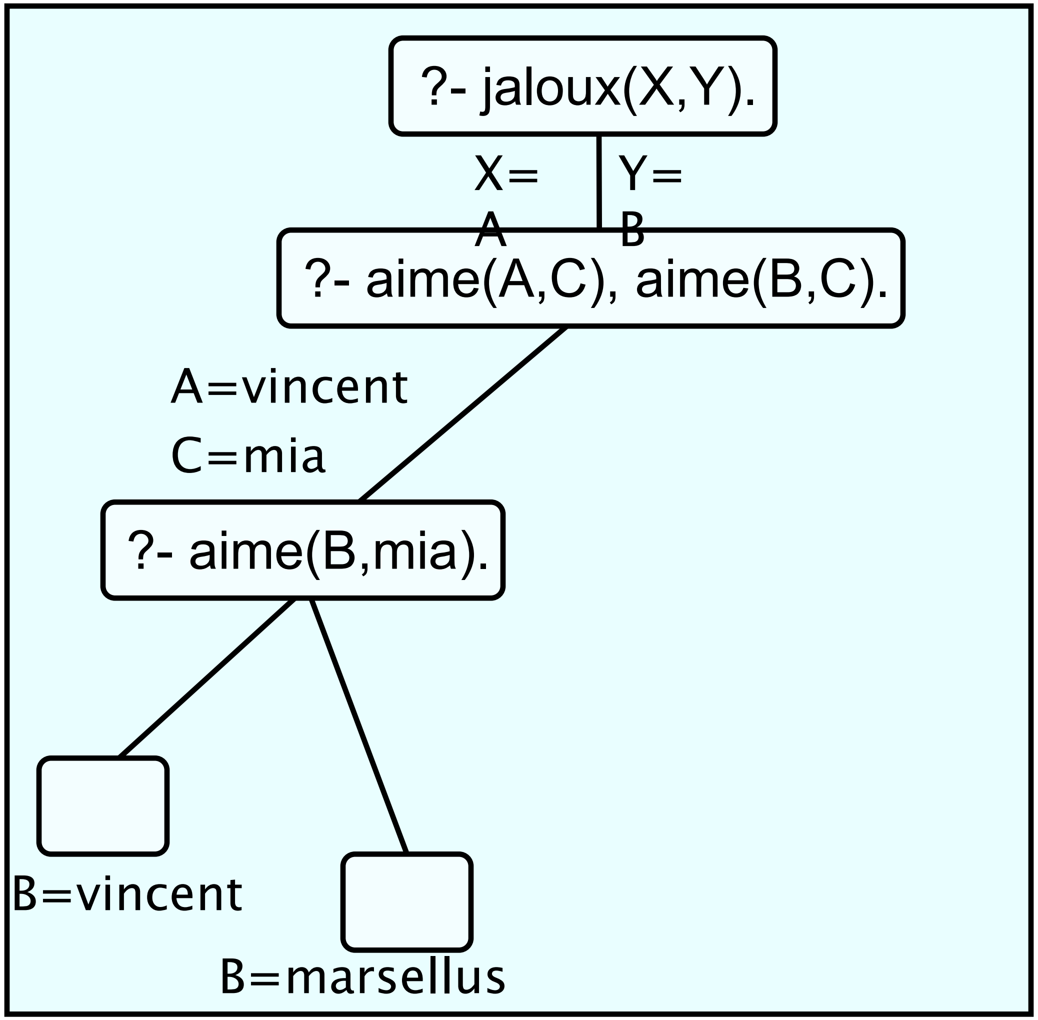


Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
?- jaloux(X,Y).
X=vincent
Y=vincent;
X=vincent
Y=marsellus
```

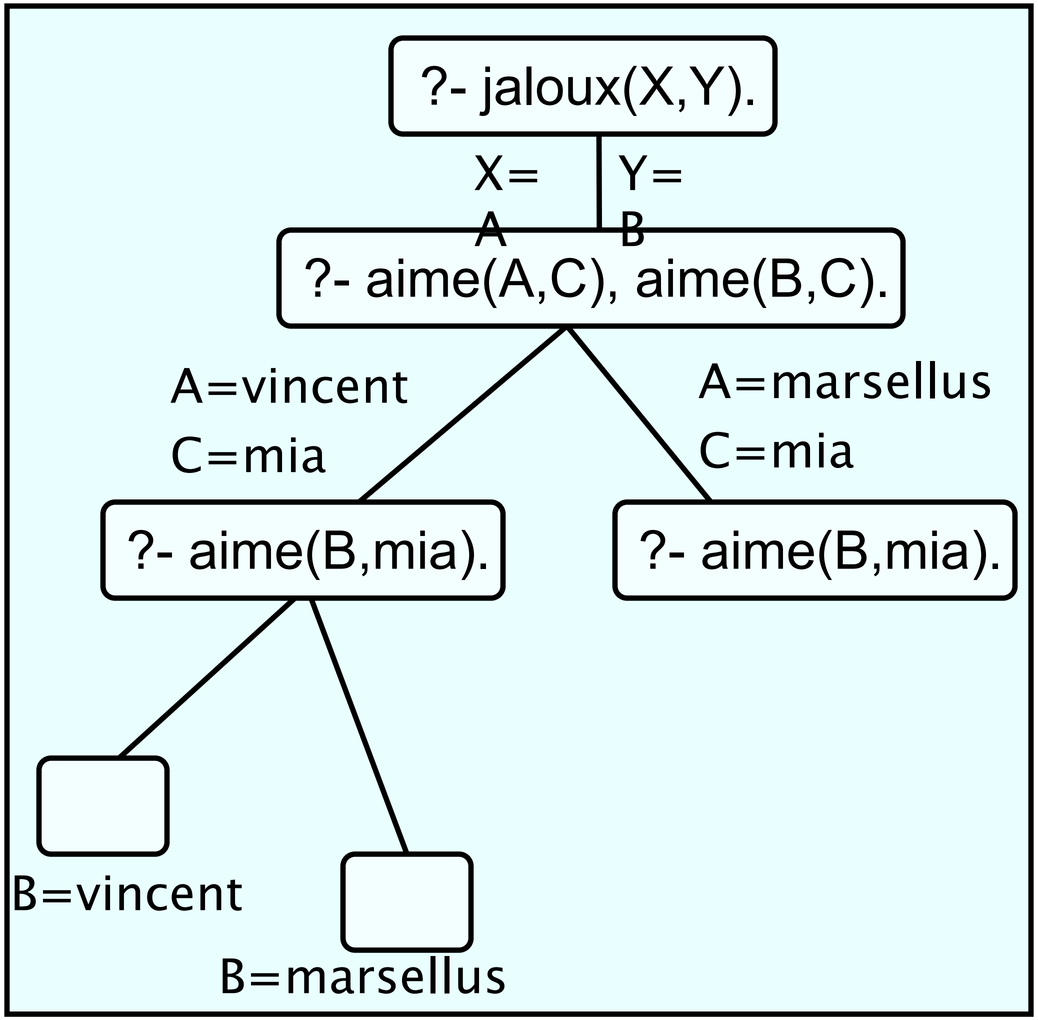


Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
?- jaloux(X,Y).
X=vincent
Y=vincent;
X=vincent
Y=marsellus;
```

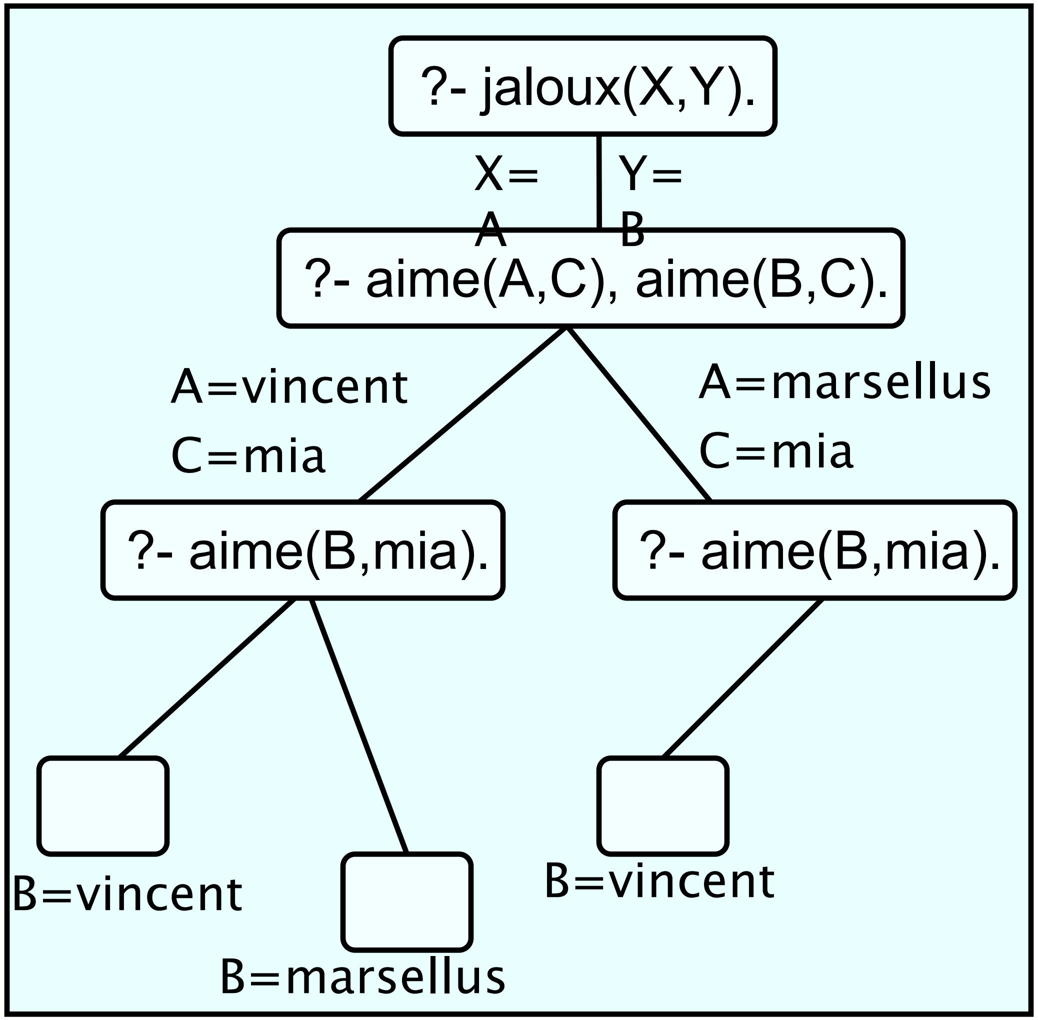


Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
....
X=vincent
Y=marsellus;
X=marsellus
Y=vincent
```

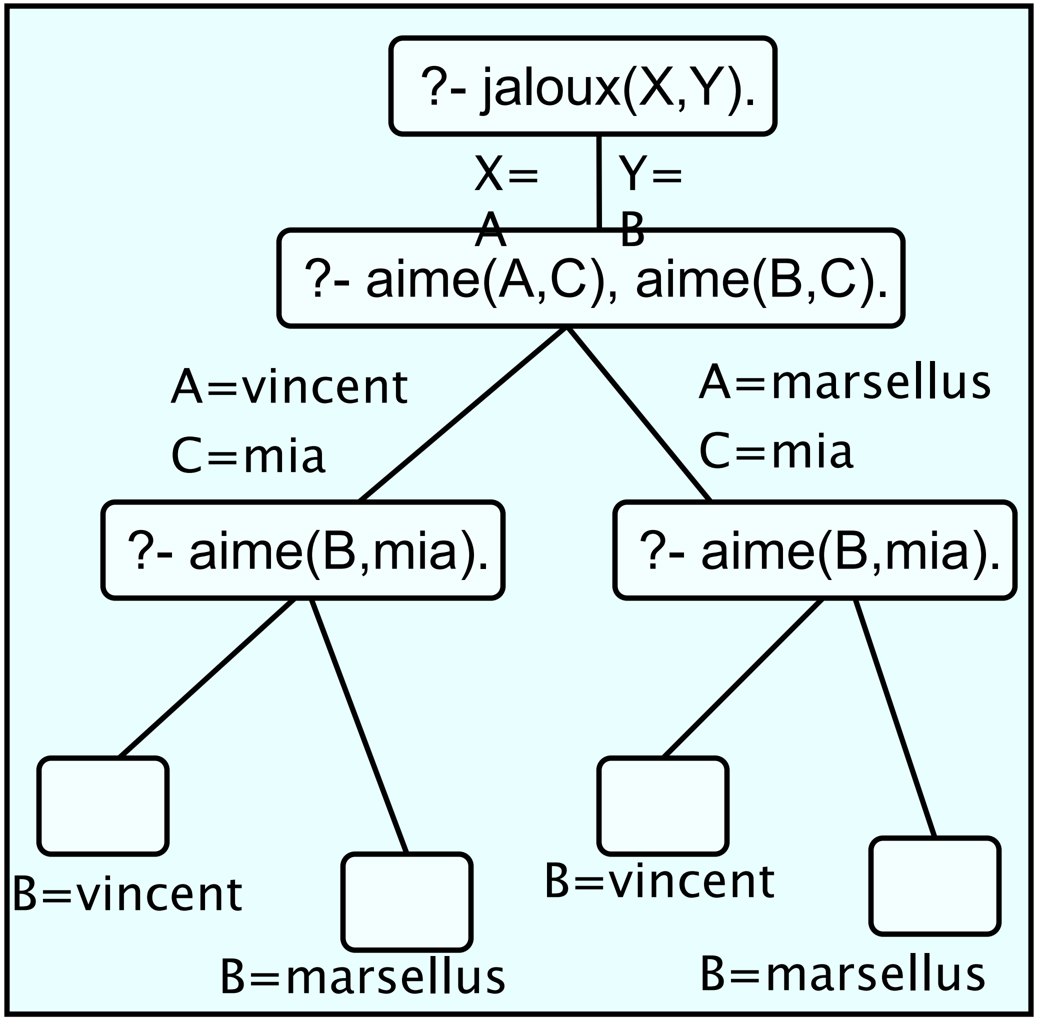


Autre exemple


```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
….
X=marsellus
Y=vincent;
X=marsellus
Y=marsellus
```

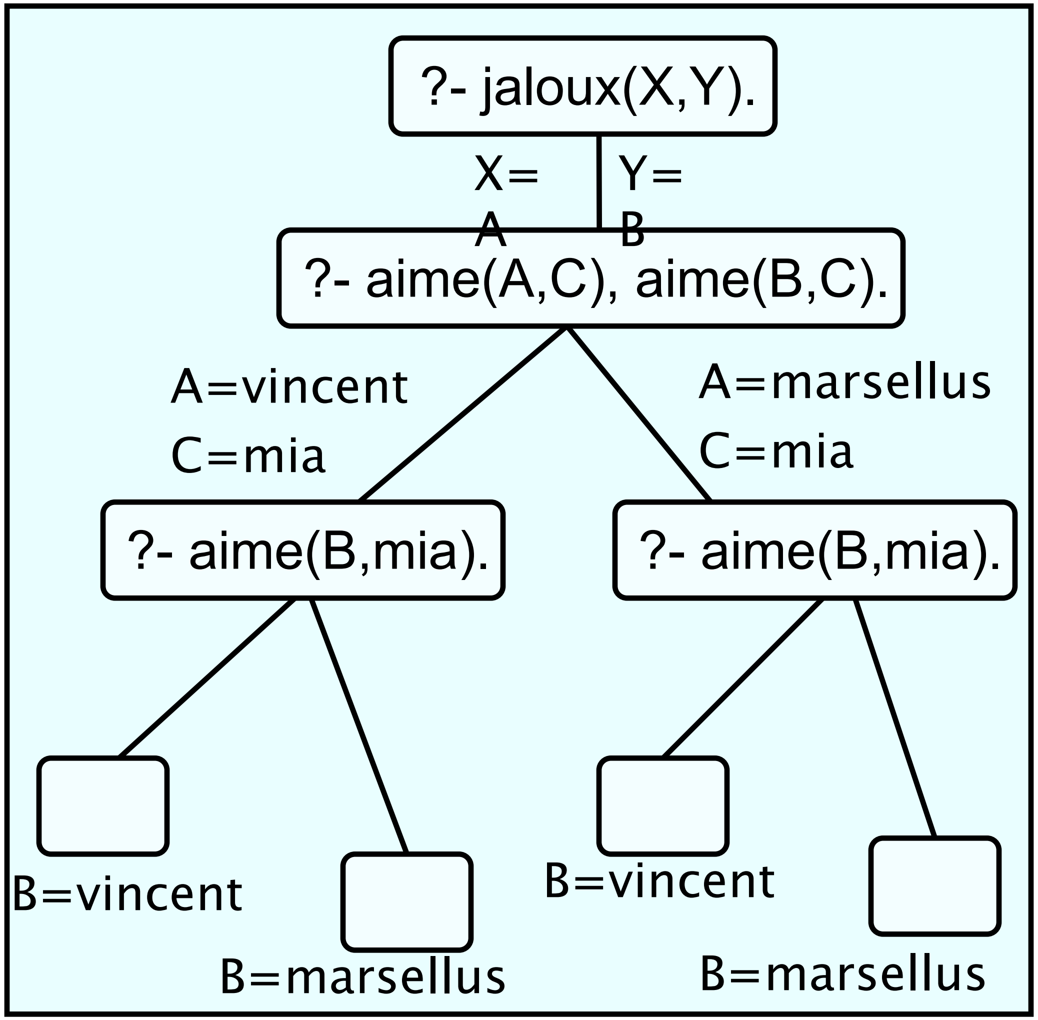


Autre exemple

```
aime(vincent,mia).
aime(marsellus,mia).

jaloux(A,B):-
    aime(A,C),
    aime(B,C).
```

```
….
X=marsellus
Y=vincent;
X=marsellus
Y=marsellus;
no
```



Exercices

ex 2.2

Résumé

- Nous avons aujourd'hui
  - défini l'unification
  - expliqué en quoi l'algorithme d'unification de Prolog est différent de l'unification pure en théorie
  - introduit les arbres de résolution

Prochain sujet

- La **récurrence** en Prolog
  - définitions s’utilisant elles-mêmes
- Montrer que le sens déclaratif d’un programme Prolog ne coïncide pas toujours avec son sens déclaratif, ni son sens procédural.