

# Lecture 2

---

- Theory
  - Unification
  - Unification in Prolog
  - Proof search
- Exercises
  - Correction exercises of LPN chapter 1
  - Exercises of LPN chapter 2

# Aim of this lecture

---

- Discuss **unification** in Prolog
  - Show how Prolog unification differs from standard unification
- Explain the search strategy that Prolog uses when it tries to deduce new information from old, using modus ponens

# Unification

- Recall previous example, where we said that Prolog unifies

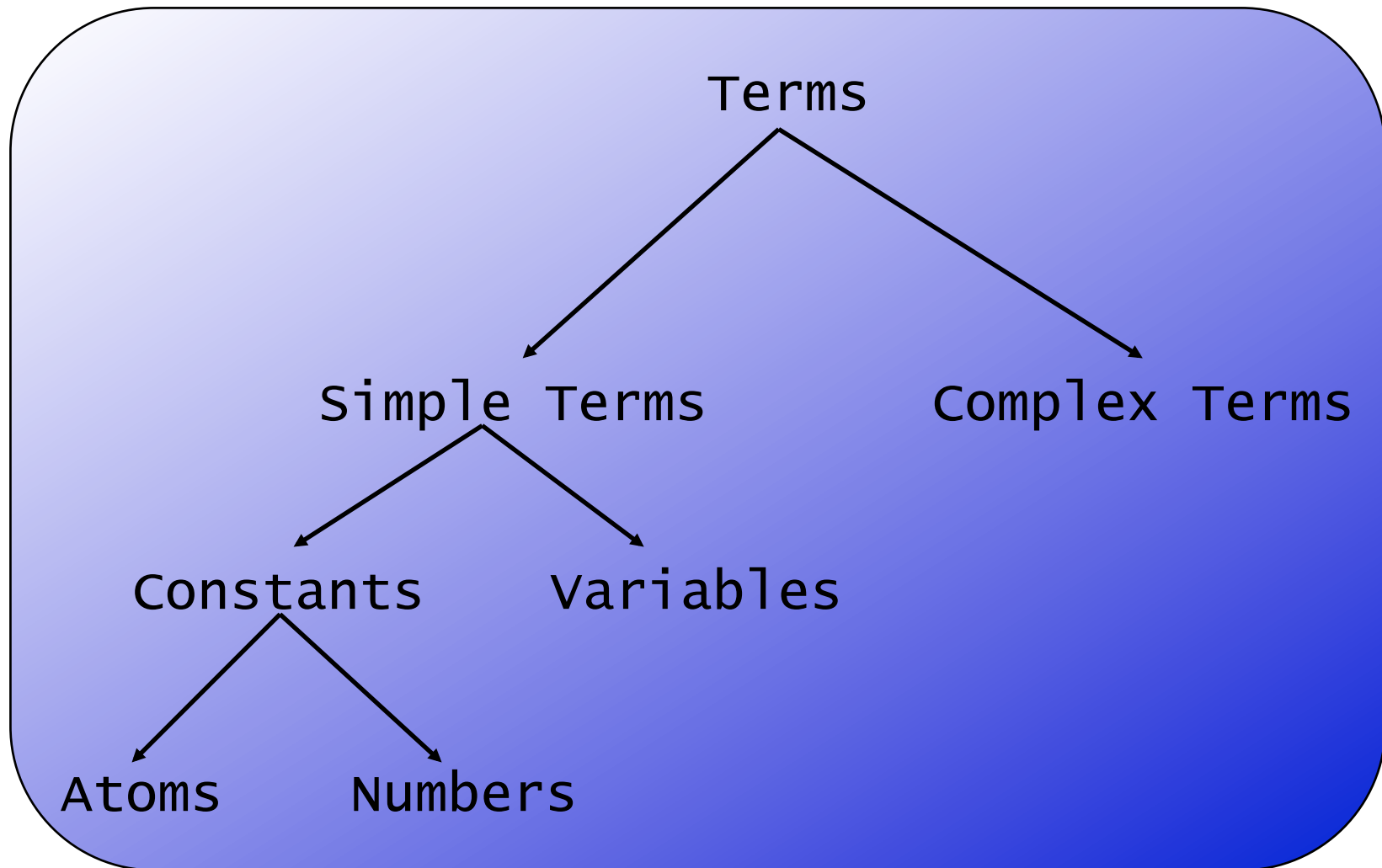
**woman(X)**

with

**woman(mia)**

thereby instantiating the variable **X** with the atom **mia**.

# Recall Prolog Terms



# Unification

---

- Working definition:
  - Two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal

# Unification

- This means that:
  - **mia** and **mia** unify
  - **42** and **42** unify
  - **woman(mia)** and **woman(mia)** unify
- This also means that:
  - **vincent** and **mia** do not unify
  - **woman(mia)** and **woman(jody)** do not unify

# Unification

- What about the terms:
  - **mia** and **X**

# Unification

- What about the terms:
  - **mia** and **X**
  - **woman(Z)** and **woman(mia)**



# Unification

- What about the terms:
  - **mia** and **X**
  - **woman(Z)** and **woman(mia)**
  - **loves(mia,X)** and **loves(X,vincent)**

# Instantiations

---

- When Prolog unifies two terms it performs all the necessary instantiations, so that the terms are equal afterwards
- This makes unification a powerful programming mechanism

# Revised Definition 1/3

---

1. If  $T_1$  and  $T_2$  are constants, then  $T_1$  and  $T_2$  unify if they are the same atom, or the same number.

# Revised Definition 2/3

1. If  $T_1$  and  $T_2$  are constants, then  $T_1$  and  $T_2$  unify if they are the same atom, or the same number.
2. If  $T_1$  is a variable and  $T_2$  is any type of term, then  $T_1$  and  $T_2$  unify, and  $T_1$  is instantiated to  $T_2$ . (and vice versa)

# Revised Definition 3/3

1. If  $T_1$  and  $T_2$  are constants, then  $T_1$  and  $T_2$  unify if they are the same atom, or the same number.
2. If  $T_1$  is a variable and  $T_2$  is any type of term, then  $T_1$  and  $T_2$  unify, and  $T_1$  is instantiated to  $T_2$ . (and vice versa)
3. If  $T_1$  and  $T_2$  are complex terms then they unify if:
  - a) They have the same functor and arity, and
  - b) all their corresponding arguments unify, and
  - c) the variable instantiations are compatible.

# Prolog unification: =/2

?- mia = mia.

yes

?-

# Prolog unification: =/2

?- mia = mia.

yes

?- mia = vincent.

no

?-

# Prolog unification: =/2

?- mia = X.

X=mia

yes

?-



# How will Prolog respond?

?- X=mia, X=vincent.

# How will Prolog respond?

?- X=mia, X=vincent.

no

?-

Why? After working through the first goal, Prolog has instantiated X with **mia**, so that it cannot unify it with **vincent** anymore. Hence the second goal fails.

# Example with complex terms

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

# Example with complex terms

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

$X = s(g)$

$Y = t(k)$

yes

?-

# Example with complex terms

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

# Example with complex terms

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

$X = s(g)$

$Y = k$

yes

?-

# One last example

---

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

# Prolog and unification

- Prolog does not use a standard unification algorithm
- Consider the following query:  
  
?- father(X) = X.
- Do these terms unify or not?





# Infinite terms

?- father(X) = X.

X=father(father(father(...))))

yes

?-

# Occurs Check

- A standard unification algorithm carries out an occurs check
- If it is asked to unify a variable with another term it checks whether the variable occurs in the term
- In Prolog:

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

# Programming with Unification

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

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

# Programming with Unification

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

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

?-

# Programming with Unification

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

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

```
?- vertical(line(point(1,1),point(1,3))).
```

yes

```
?-
```

# Programming with Unification

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

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

```
?- vertical(line(point(1,1),point(1,3))).
```

yes

```
?- vertical(line(point(1,1),point(3,2))).
```

no

```
?-
```

# Programming with Unification

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

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

```
?- horizontal(line(point(1,1),point(1,Y))).
```

```
Y = 1;
```

```
no
```

```
?-
```



# Programming with Unification

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

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

```
?- horizontal(line(point(2,3),Point)).
```

```
Point = point(_554,3);
```

```
no
```

```
?-
```

# Proof Search

---

- Now that we know about unification, we are in a position to learn how Prolog searches a knowledge base to see if a query is satisfied.
- In other words: we are ready to learn about proof search

# Example

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

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

# Example: search tree

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

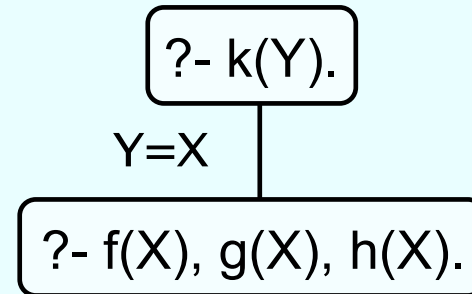
?- k(Y).

?- k(Y).

# Example: search tree

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

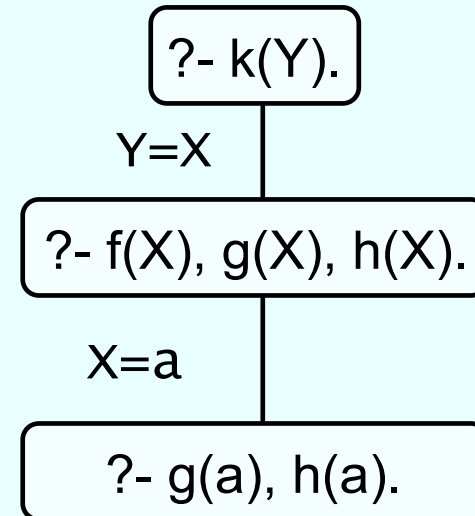
?- k(Y).



# Example: search tree

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

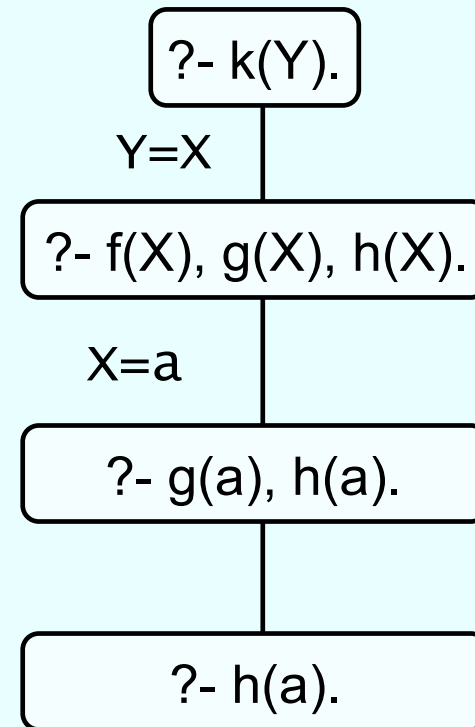
?- k(Y).



# Example: search tree

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

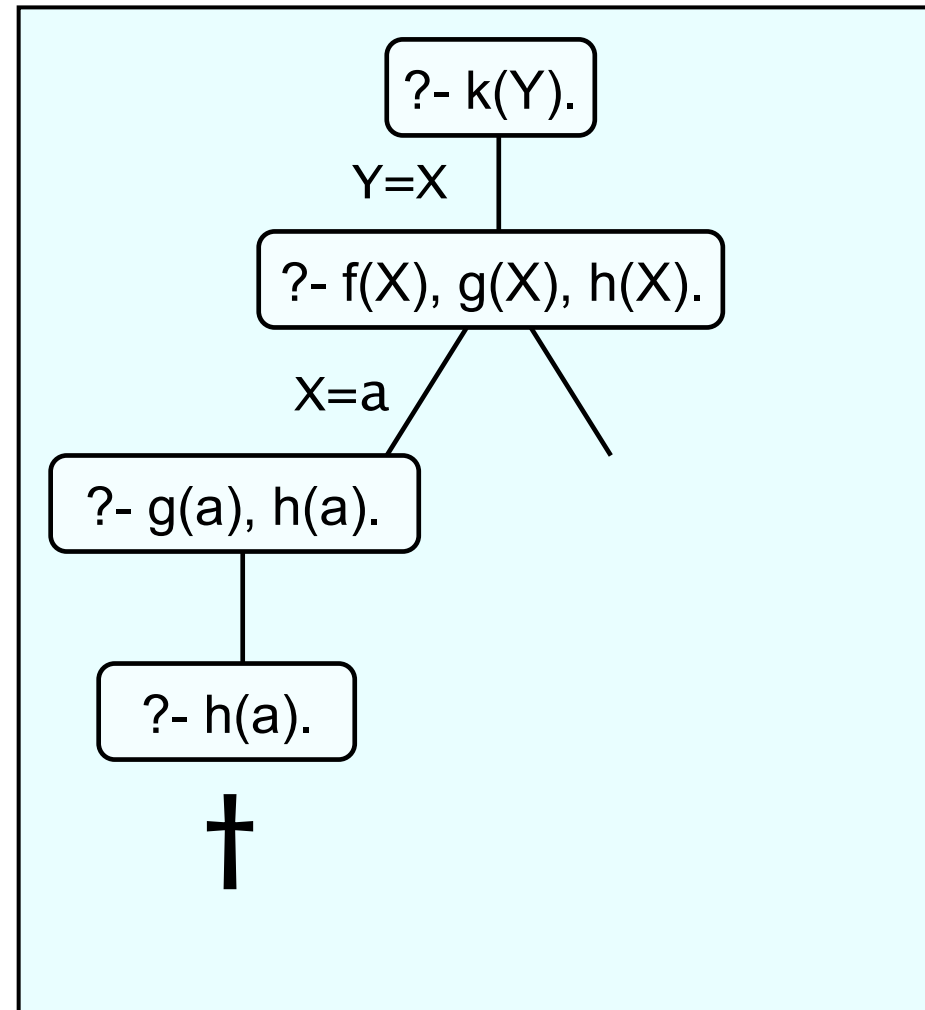
?- k(Y).



# Example: search tree

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

?- k(Y).

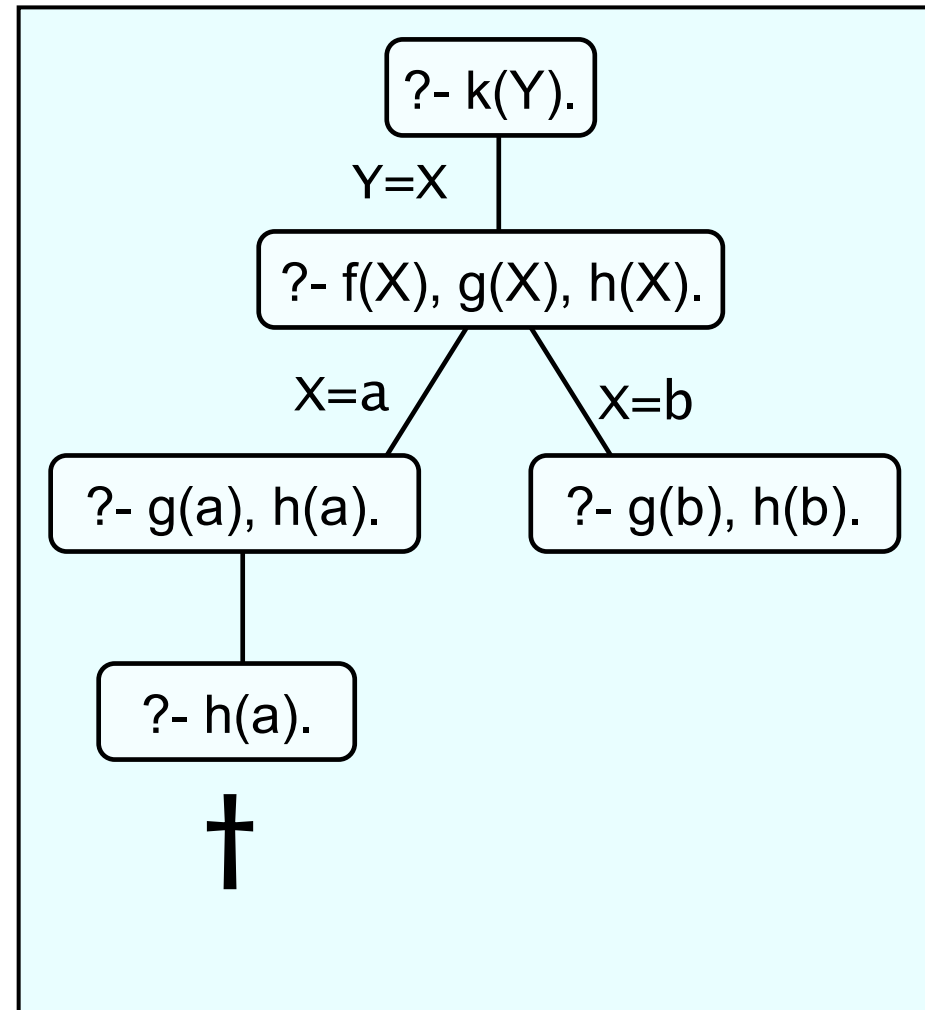




# Example: search tree

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

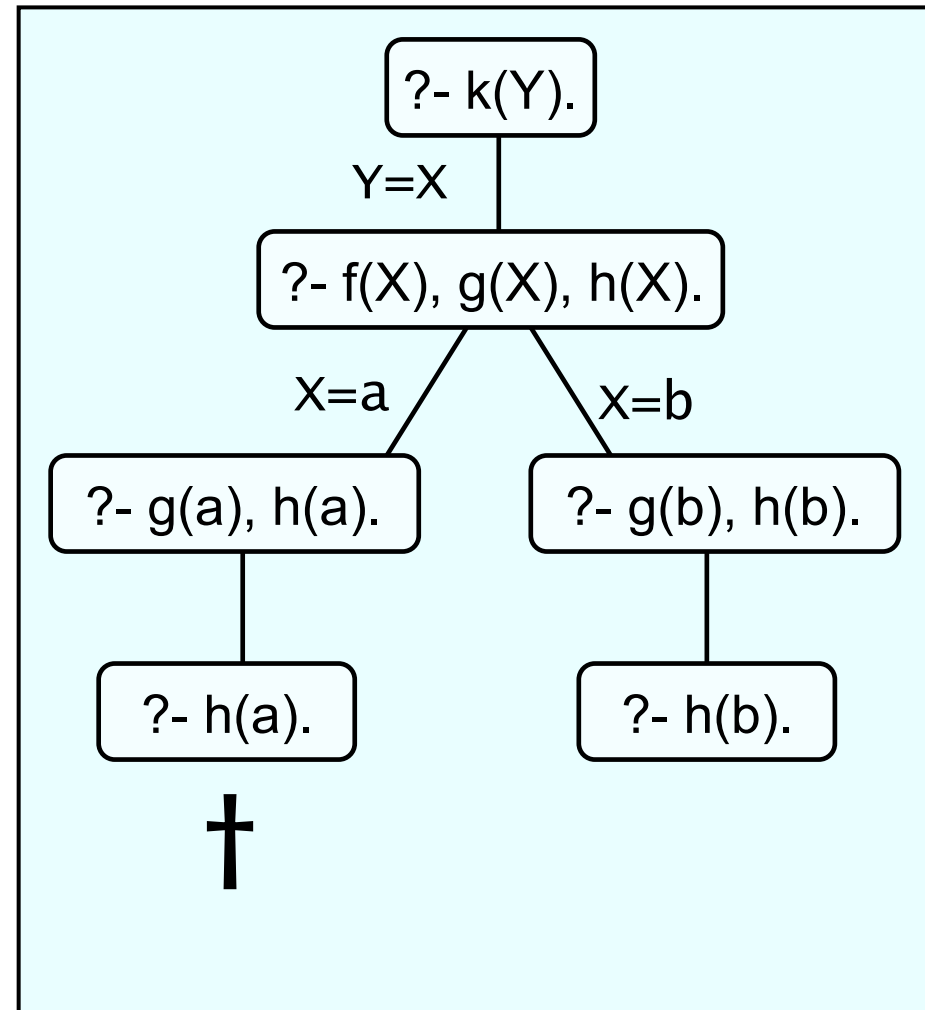
?- k(Y).



# Example: search tree

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

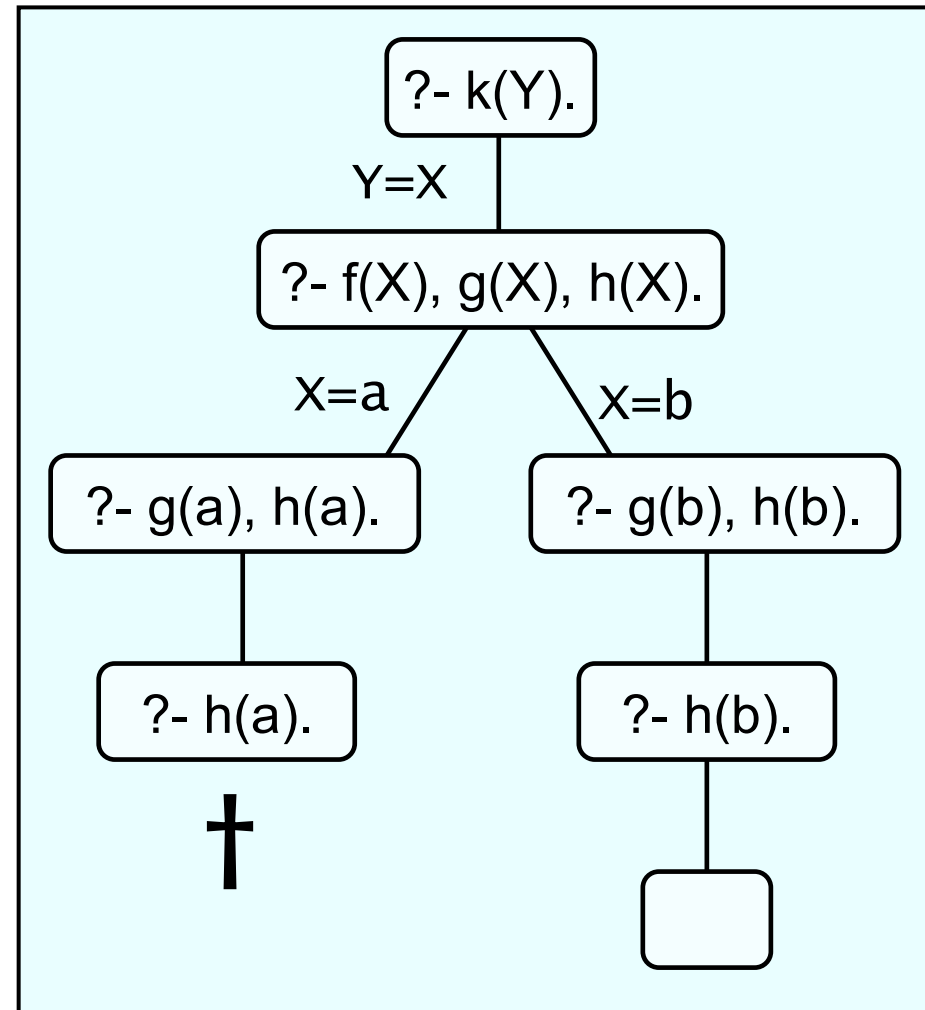
?- k(Y).



# Example: search tree

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

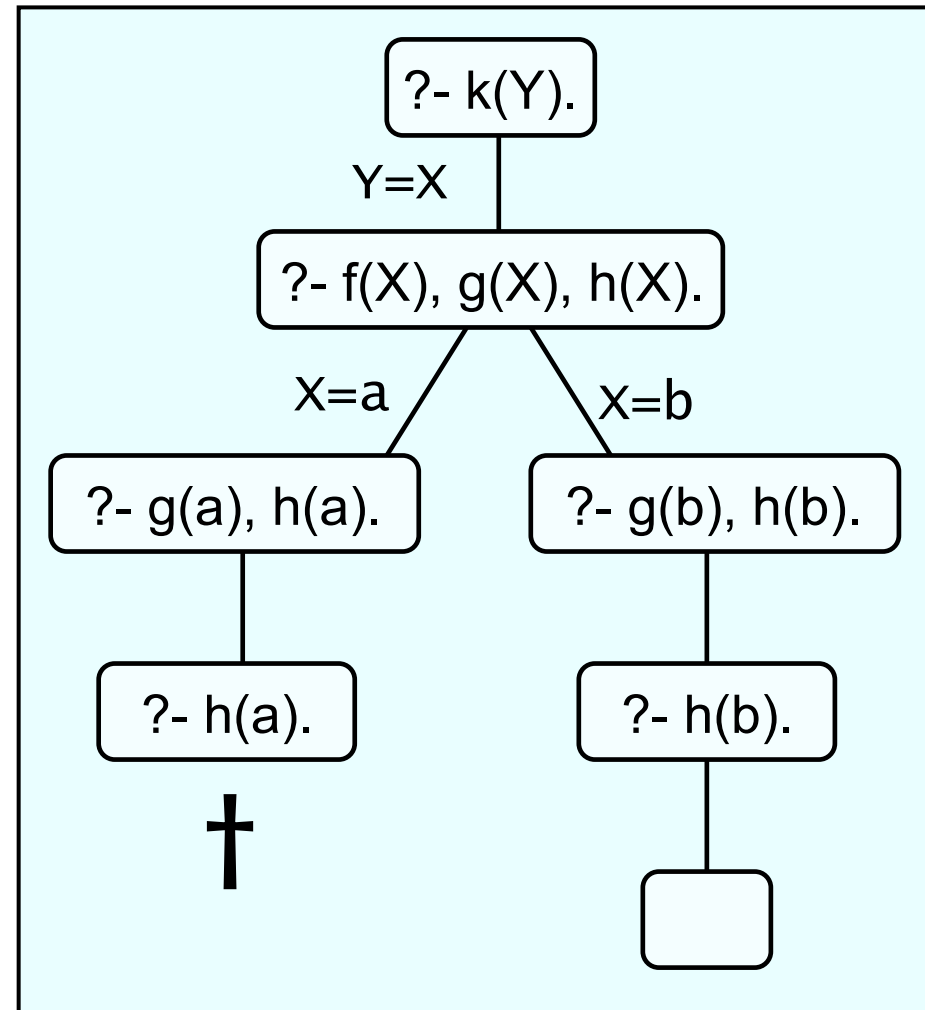
?- k(Y).  
Y=b



# Example: search tree

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

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



# Another example

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

```
jealous(A,B):-  
    loves(A,C),  
    loves(B,C).
```

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

# Another example

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

```
jealous(A,B):-  
    loves(A,C),  
    loves(B,C).
```

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

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

# Another example

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

```
jealous(A,B):-  
    loves(A,C),  
    loves(B,C).
```

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

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

```
X=A | Y=B
```

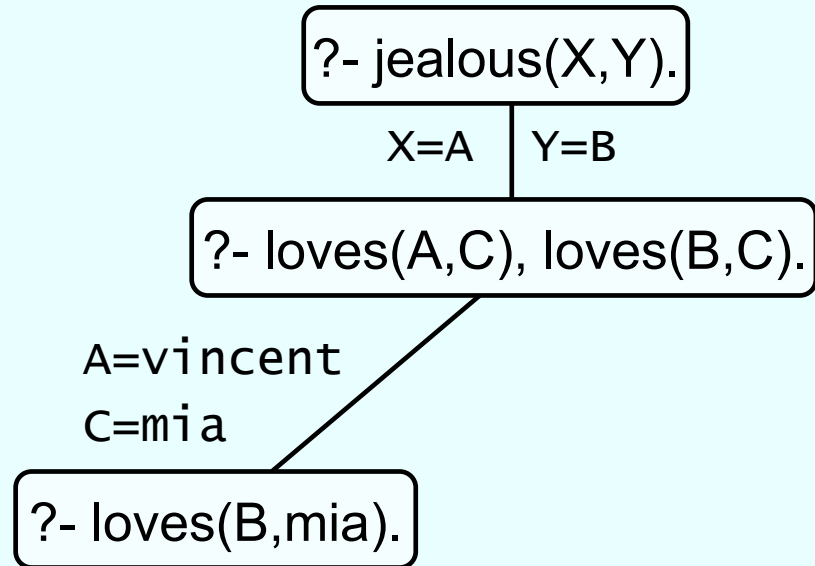
```
?- loves(A,C), loves(B,C).
```

# Another example

loves(vincent,mia).  
loves(marsellus,mia).

jealous(A,B):-  
  loves(A,C),  
  loves(B,C).

?- jealous(X,Y).



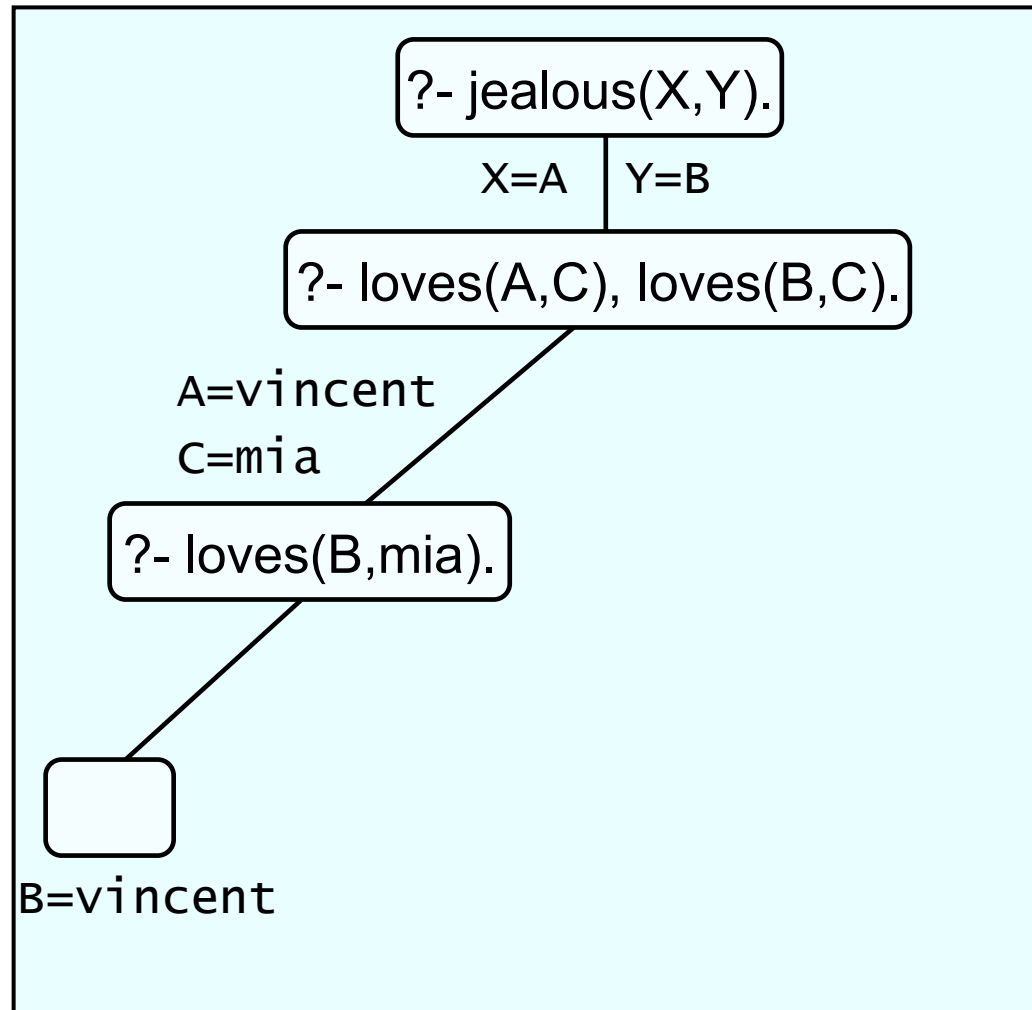


# Another example

loves(vincent,mia).  
loves(marsellus,mia).

jealous(A,B):-  
  loves(A,C),  
  loves(B,C).

?- jealous(X,Y).  
X=vincent  
Y=vincent

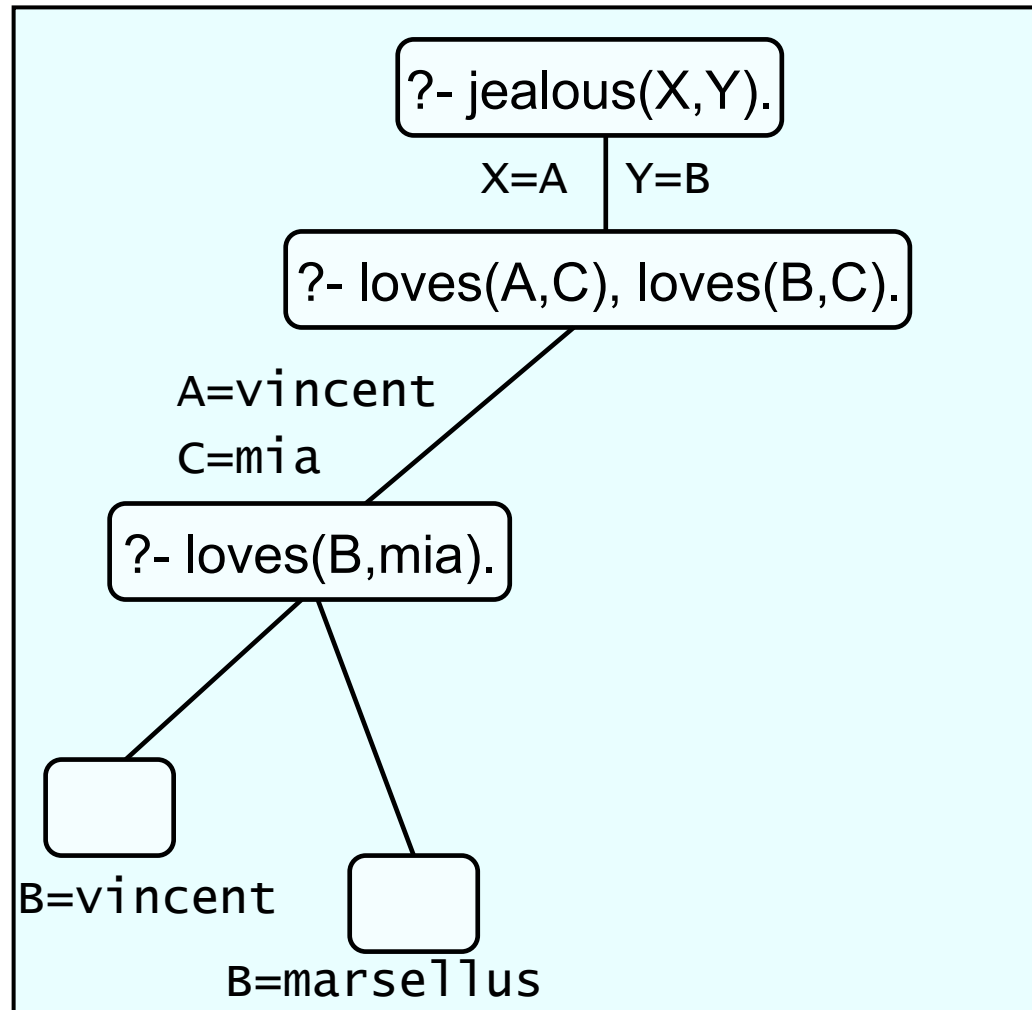


# Another example

loves(vincent,mia).  
loves(marsellus,mia).

jealous(A,B):-  
  loves(A,C),  
  loves(B,C).

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

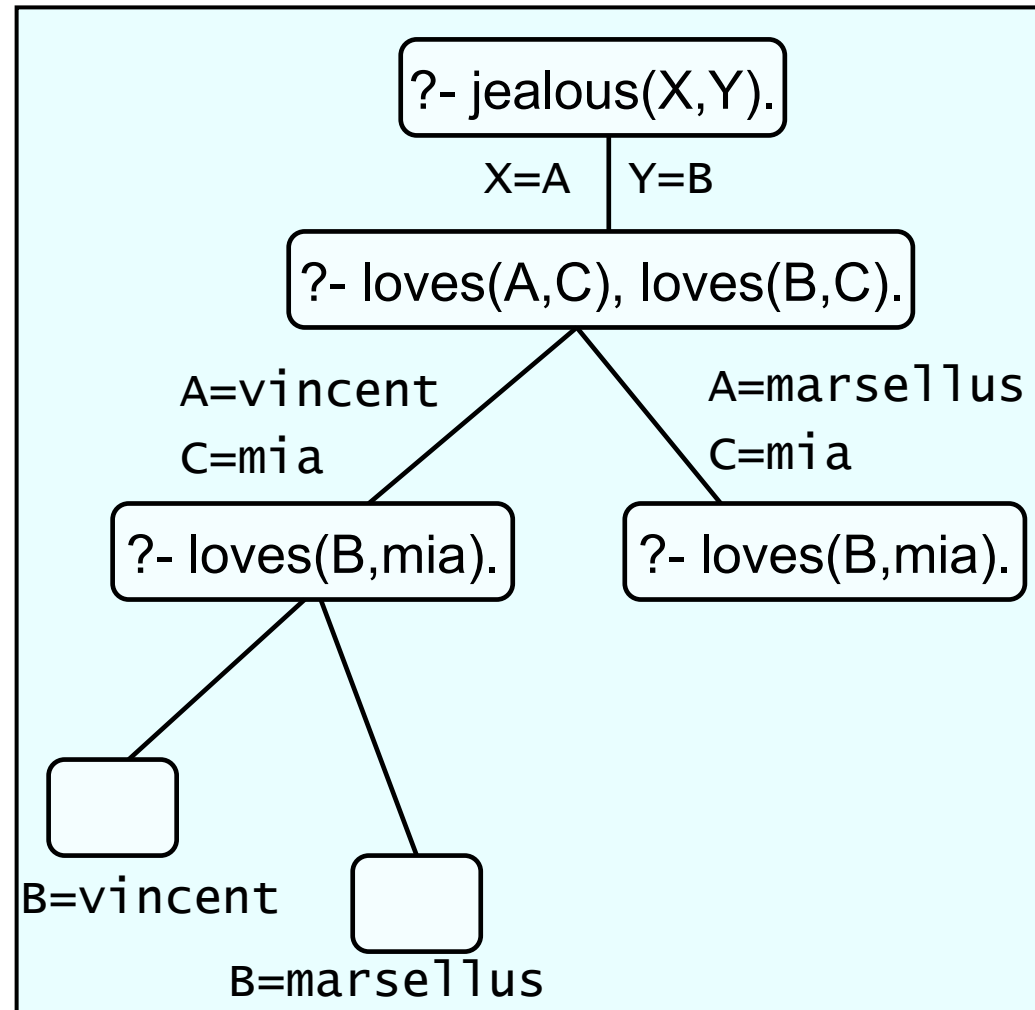


# Another example

loves(vincent,mia).  
loves(marsellus,mia).

jealous(A,B):-  
  loves(A,C),  
  loves(B,C).

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

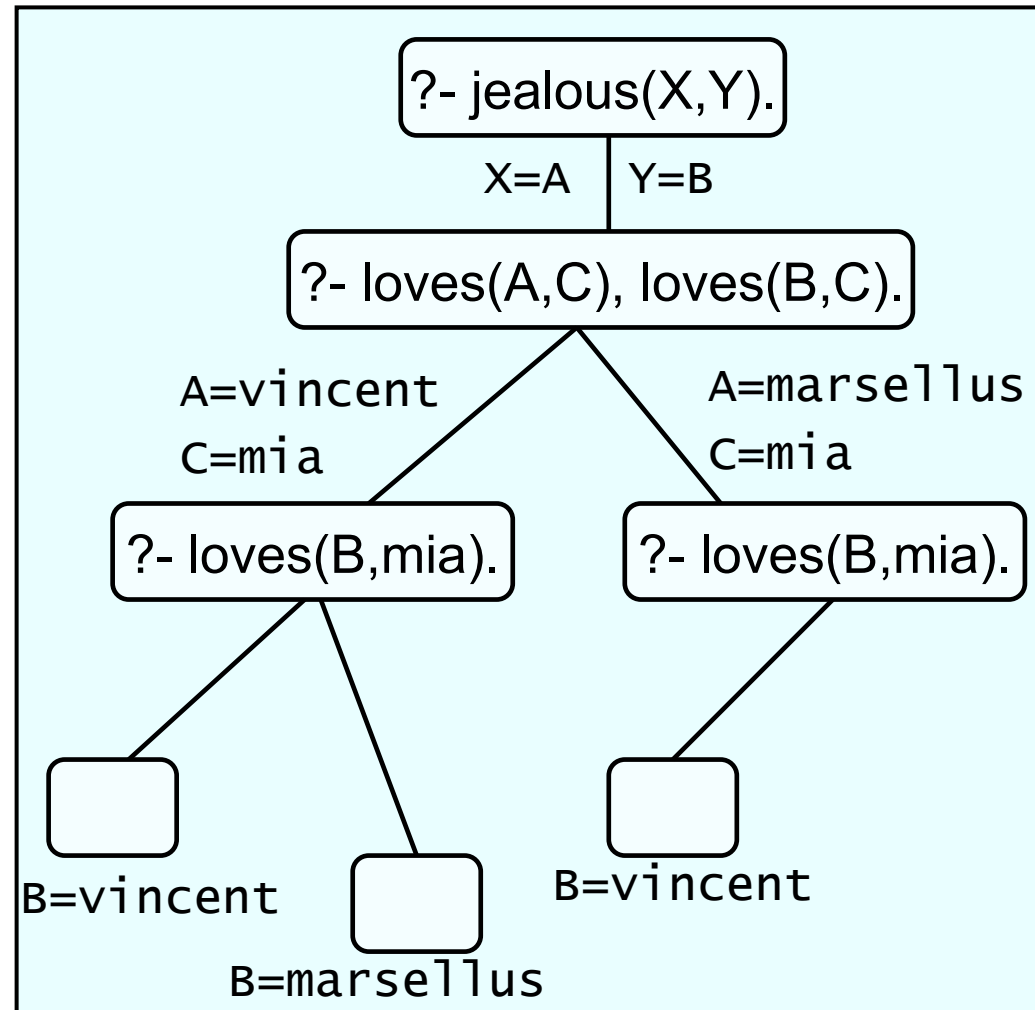


# Another example

loves(vincent,mia).  
loves(marsellus,mia).

jealous(A,B):-  
  loves(A,C),  
  loves(B,C).

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

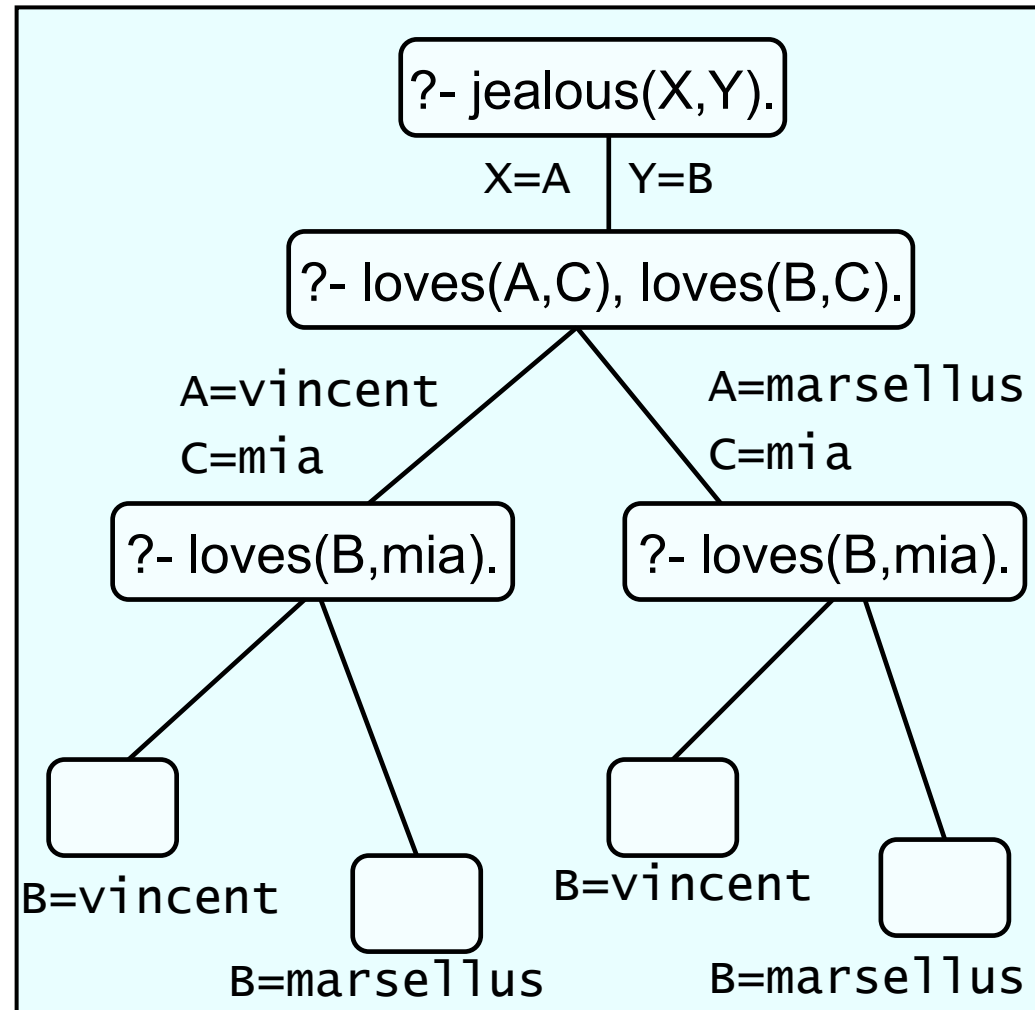


# Another example

loves(vincent,mia).  
loves(marsellus,mia).

jealous(A,B):-  
  loves(A,C),  
  loves(B,C).

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

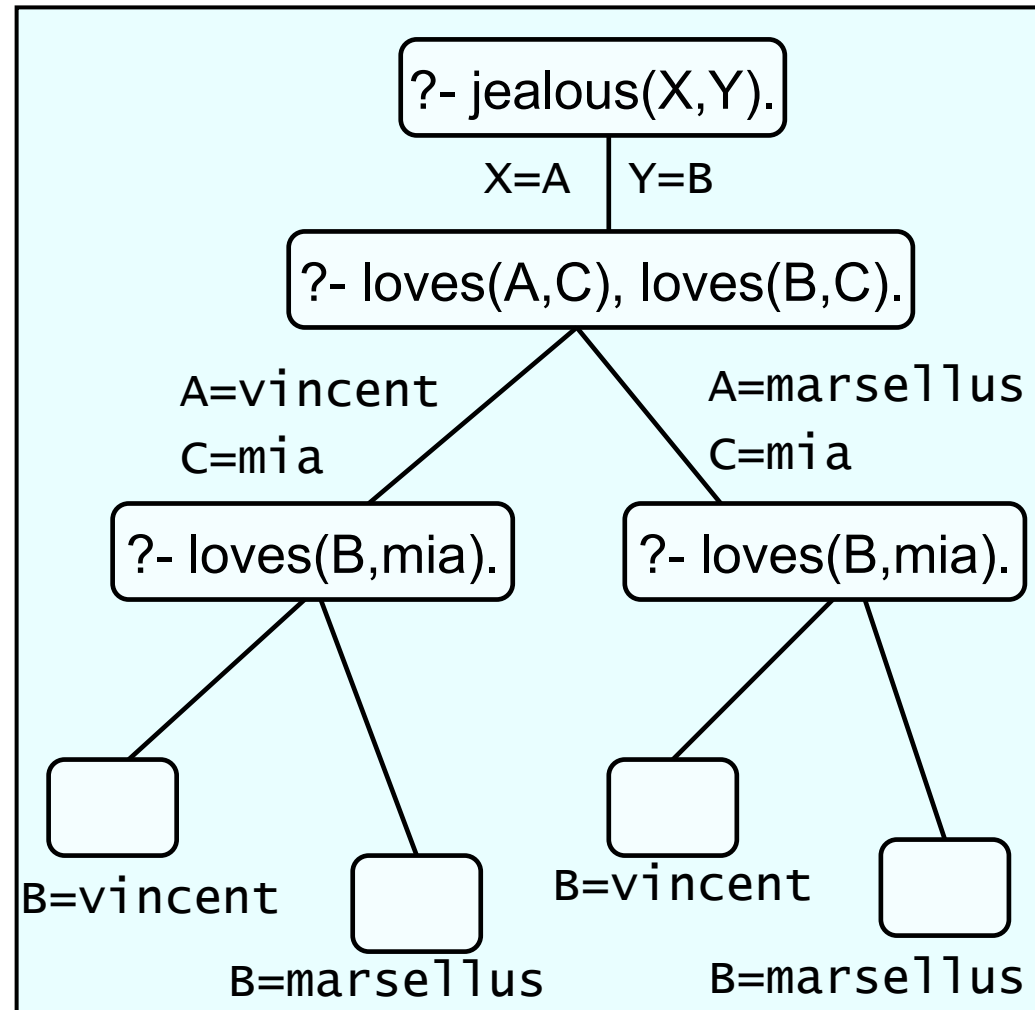


# Another example

loves(vincent,mia).  
loves(marsellus,mia).

jealous(A,B):-  
  loves(A,C),  
  loves(B,C).

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



# Solution Exercise 1.1

Which of the following sequences of characters are atoms, which are variables, and which are neither?

1.vINCENT	Atom
2.Footmassage	Variable
3.variable23	Atom
4.variable2000	Variable
5.big_kahuna_burger	Atom
6.'big kahuna burger'	Atom
7.big kahuna burger	neither: spaces without single quotes
8.'Jules'	Atom
9._Jules	Variable: starts with an underscore
10.'_Jules'	Atom: in single quotes

# Solution Exercise 1.2

Which of the following sequences of characters are atoms, which are variables, which are complex terms, and which are not terms at all? Give the functor and arity of each complex term.

1. <code>loves(Vincent, mia)</code>	complex: loves/2
2. <code>'loves(Vincent, mia)'</code>	atom (in single quotes)
3. <code>Butch(boxer)</code>	not a term (the functor must be an atom, and it is here a variable)
4. <code>boxer(Butch)</code>	complex: boxer/1
5. <code>and(big(burger), kahuna(burger))</code>	complex: and/2, big/1, kahuna/1
6. <code>and(big(X), kahuna(X))</code>	complex: and/2, big/1, kahuna/1
7. <code>_and(big(X), kahuna(X))</code>	not a term (the functor must be an atom, and it is here a variable)
8. <code>(Butch kills Vincent)</code>	not a term (the character sequence '(Butch' does not start with a lower-case letter or with a single quote, so it is not an atom; it does not start with an upper-case letter or with an underscore, so it is not a variable; and it is obviously not a number or a complex term.)
9. <code>kills(Butch Vincent)</code>	not a term (an operator is missing for the variables 'Butch' and 'Vincent', or a comma is missing for them to be arguments of the functor 'kills')
10. <code>kills(Butch, Vincent</code>	not a term (closing parenthesis for it to be a predicate; and the opening parenthesis is not allowed in atoms and is not an operator)



# Solution Exercise 1.3

How many facts, rules, clauses, and predicates are there in the following knowledge base? What are the heads of the rules, and what are the goals they contain?

woman(vincent).

woman(mia).

man(jules).

person(X) :- man(X); woman(X).

loves(X,Y) :- knows(Y,X).

father(Y,Z) :- man(Y), son(Z,Y).

father(Y,Z) :- man(Y), daughter(Z,Y).

goal

Total 3 facts (clauses 1-3), 4 rules (clauses 4-7), 7 clauses

head

and 8 predicates (woman/1, man/1, person/1, loves/2, knows/2, father/2, son/2, daughter/2)

# Exercises Chapter 2

- 2.1, 2.2, 2.4

# Summary of this lecture

---

- In this lecture we have
  - defined unification
  - looked at the difference between standard unification and Prolog unification
  - introduced search trees

# Next lecture

---

- Discuss **recursion** in Prolog
  - Introduce recursive definitions in Prolog
  - Show that there can be mismatches between the declarative meaning of a Prolog program, and its procedural meaning.