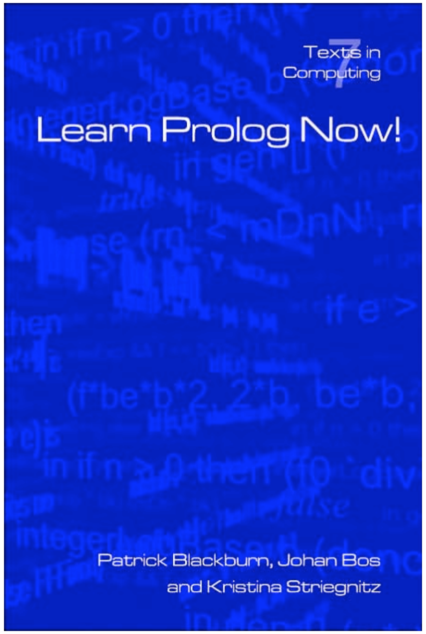
Lecture 2

- Theory
  - Unification
  - Unification in Prolog
  - Proof search

- Exercises
  - Exercises of LPN chapter 2
  - Practical work



Aim of this lecture

- Discuss **unification** in Prolog
  - Show how Prolog unification differs from standard unification
- Explain Prolog's **search strategy**
  - Prolog deduces new information from old, using modus ponens

Unification

- Recall the previous example, where we said that Prolog unifies

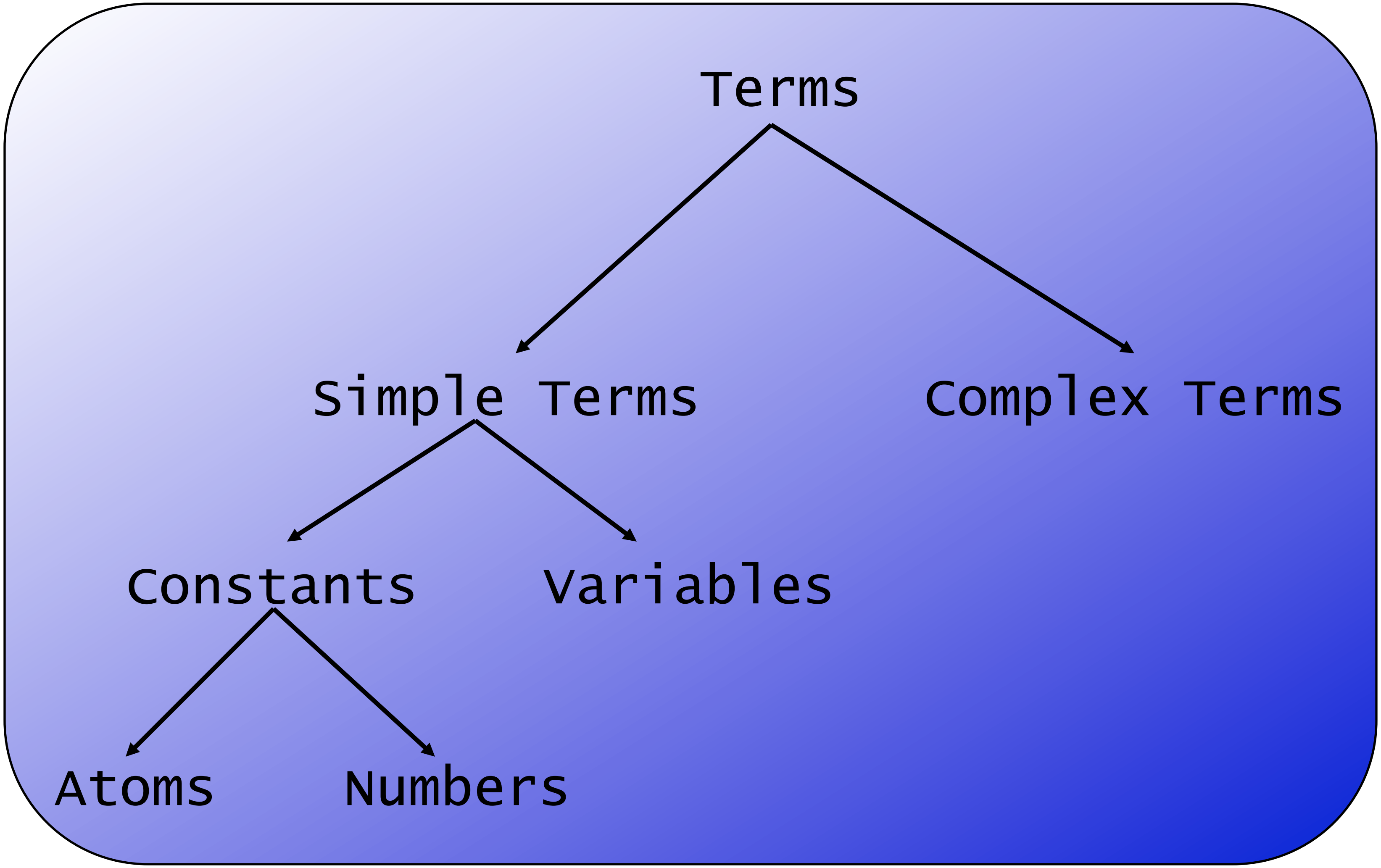
  woman(X)

  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

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 very 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:
   1. They have the same functor and arity, and
   2. all their corresponding arguments unify, and
   3. 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).

One last example

```
?- loves(X,X) = loves(marsellus,mia).
no
?-
```

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(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(………….

Infinite terms (SWI Prolog)

```
?- 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 this term
- In Prolog (ISO standard):

```
?- 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
?-
```

Exercises: unification

Exercise 2.1

Which of the following pairs of terms unify? Where relevant, give the variable instantiations that lead to successful unification.

1. bread = bread
2. 'Bread' = bread
3. 'bread' = bread
4. Bread = bread
5. bread = sausage
6. food(bread) = bread
7. food(bread) = X
8. food(X) = food(bread)
9. food(bread,X) = food(Y,sausage)
10. food(bread,X,beer) = food(Y,sausage,X)
11. food(bread,X,beer) = food(Y,kahuna_burger)
12. food(X) = X
13. meal(food(bread),drink(beer)) = meal(X,Y)
14. meal(food(bread),X) = meal(X,drink(beer))

Exercise 2.2a

We are working with the following knowledge base:

```
house_elf(dobby).
witch(hermione).
witch('McGonagall').
witch(rita_skeeter).
magic(X):- house_elf(X).
magic(X):- wizard(X).
magic(X):- witch(X).
```

Which of the following queries are satisfied? Where relevant, give all the variable instantiations that lead to success.

1. ?- magic(house_elf).
2. ?- wizard(harry).
3. ?- magic(wizard).
4. ?- magic('McGonagall').
5. ?- magic(Hermione).

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 <u>proof search</u> and <u>search trees</u>

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).
```

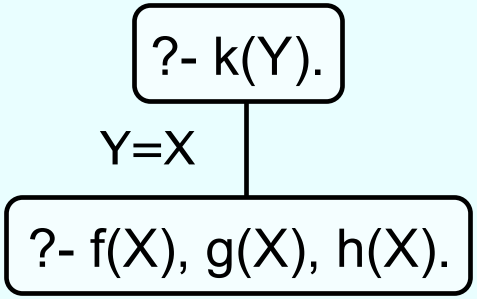
```
?- k(Y).
```

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

Example: search tree

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

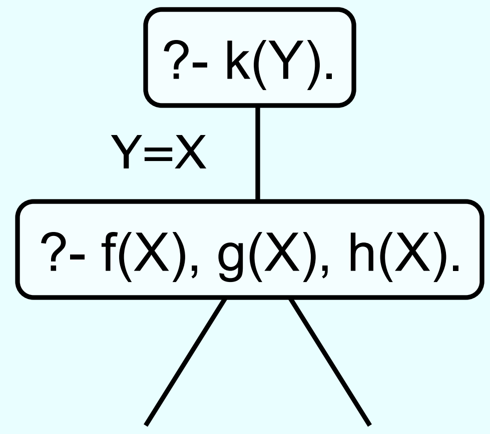
```
?- k(Y).
```



Example: search tree

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

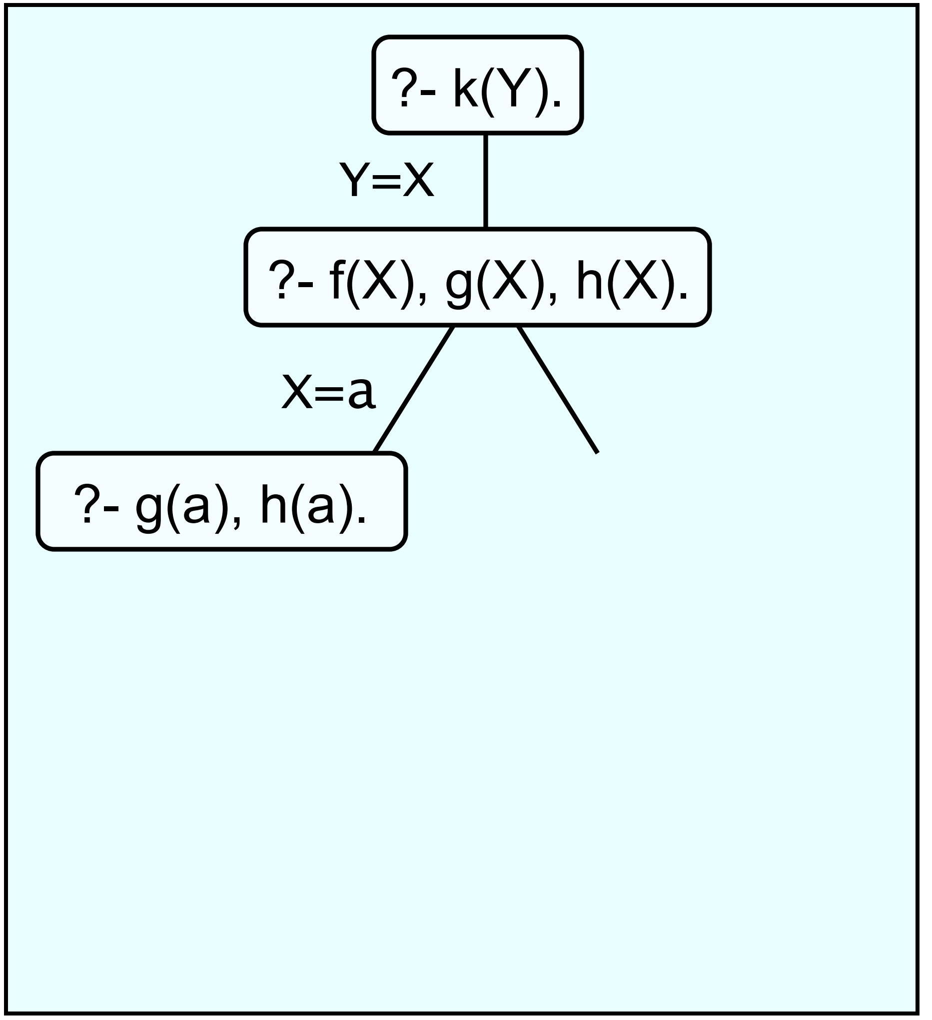
```
?- k(Y).
```



Example: search tree

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

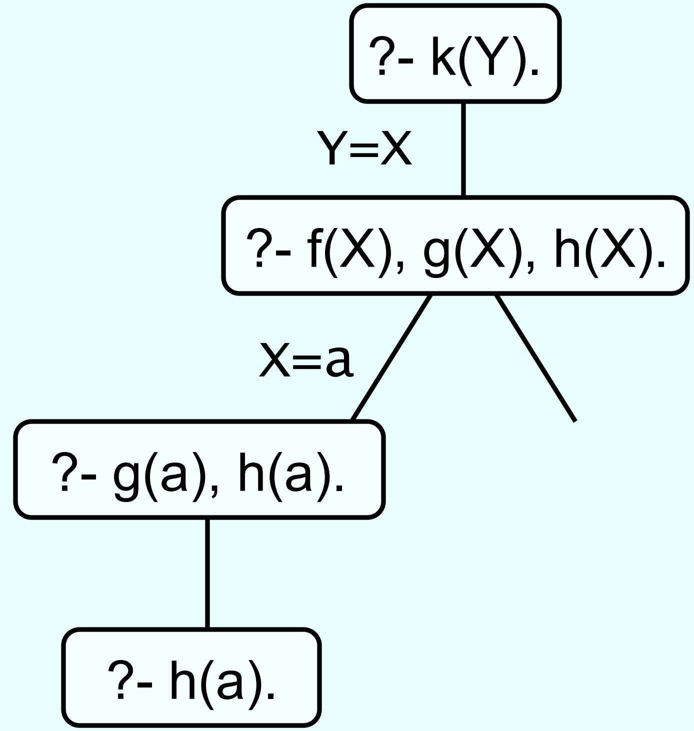
?- k(Y).



Example: search tree

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

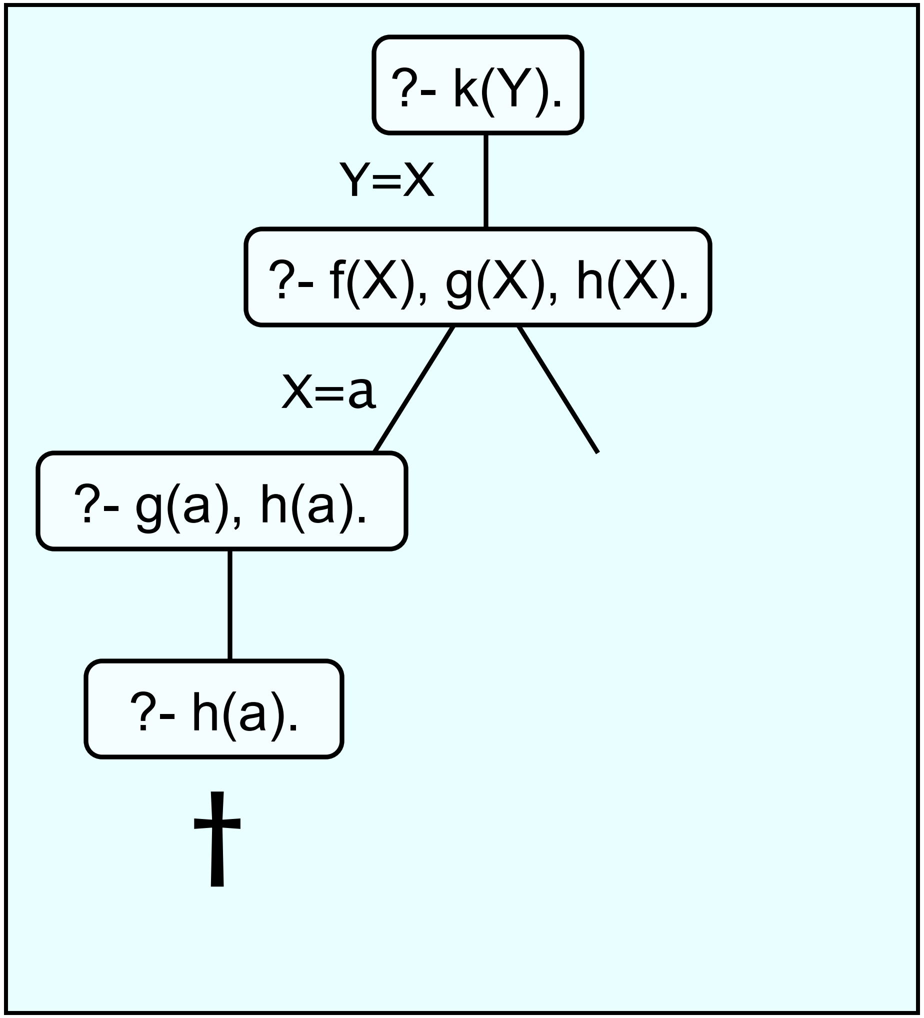
?- k(Y).



Example: search tree

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

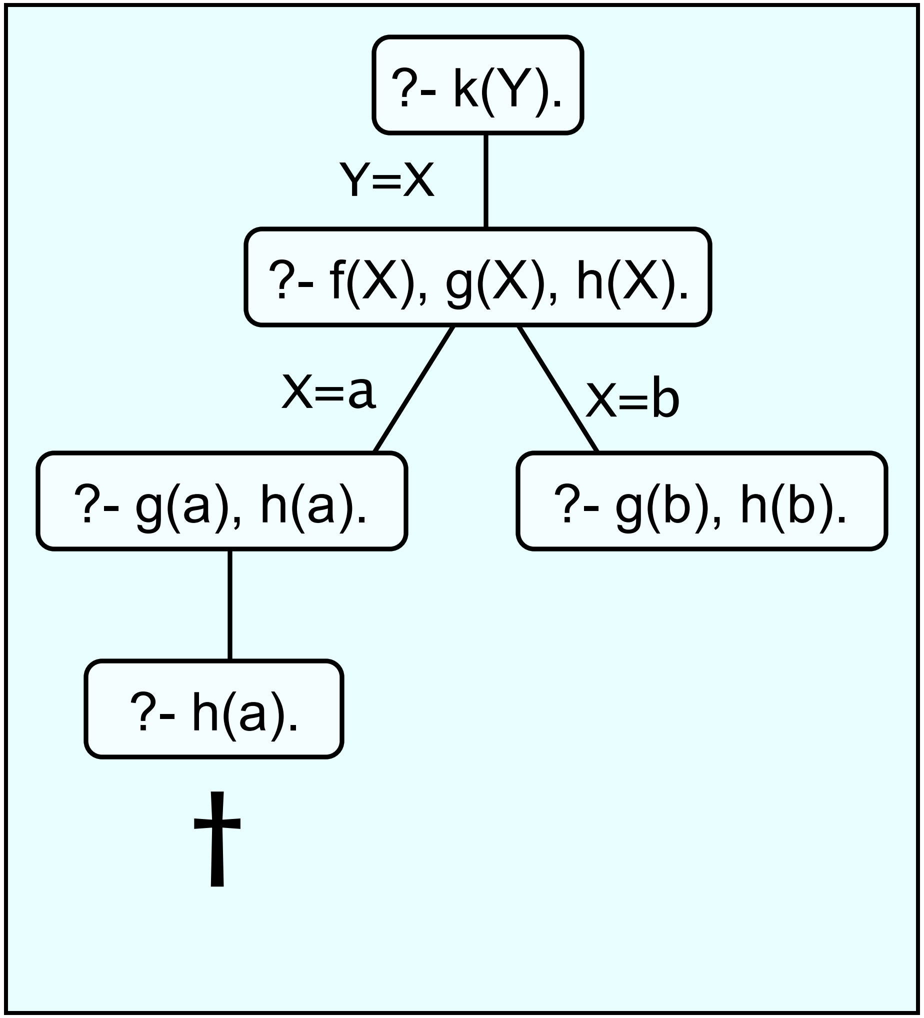
```
?- k(Y).
```



Example: search tree

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

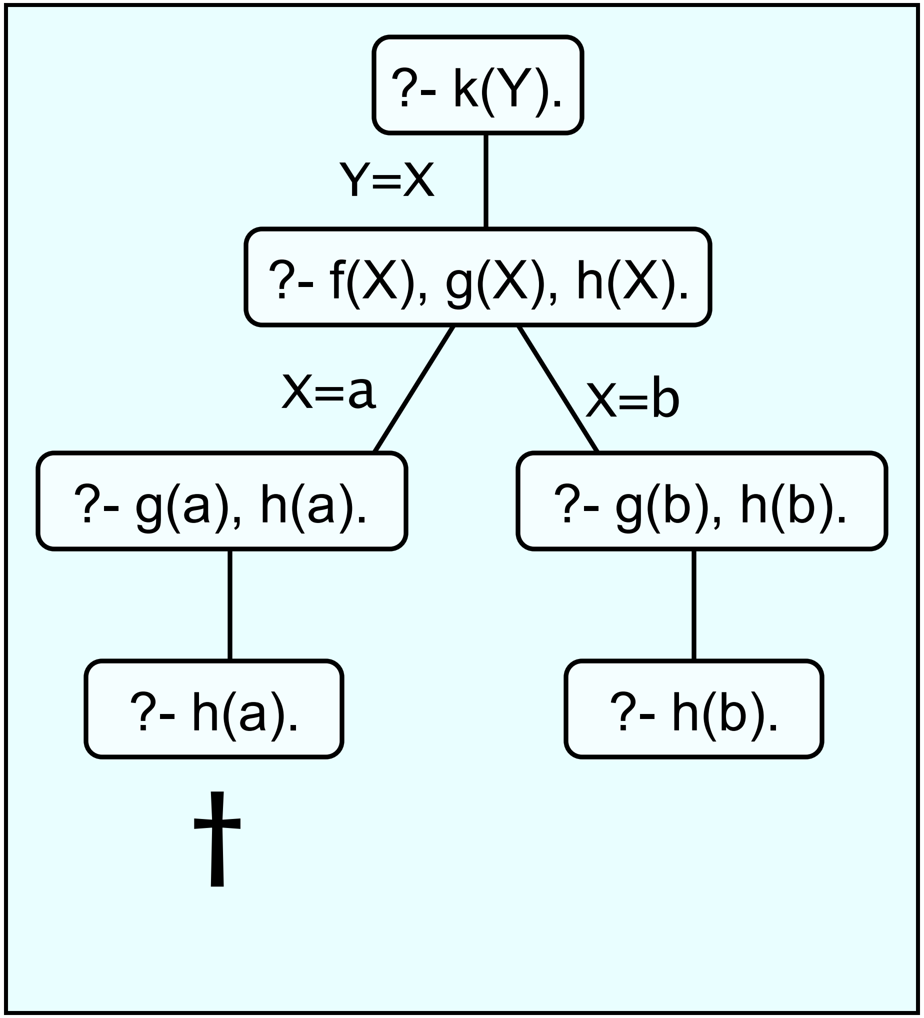
```
?- k(Y).
```



Example: search tree

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

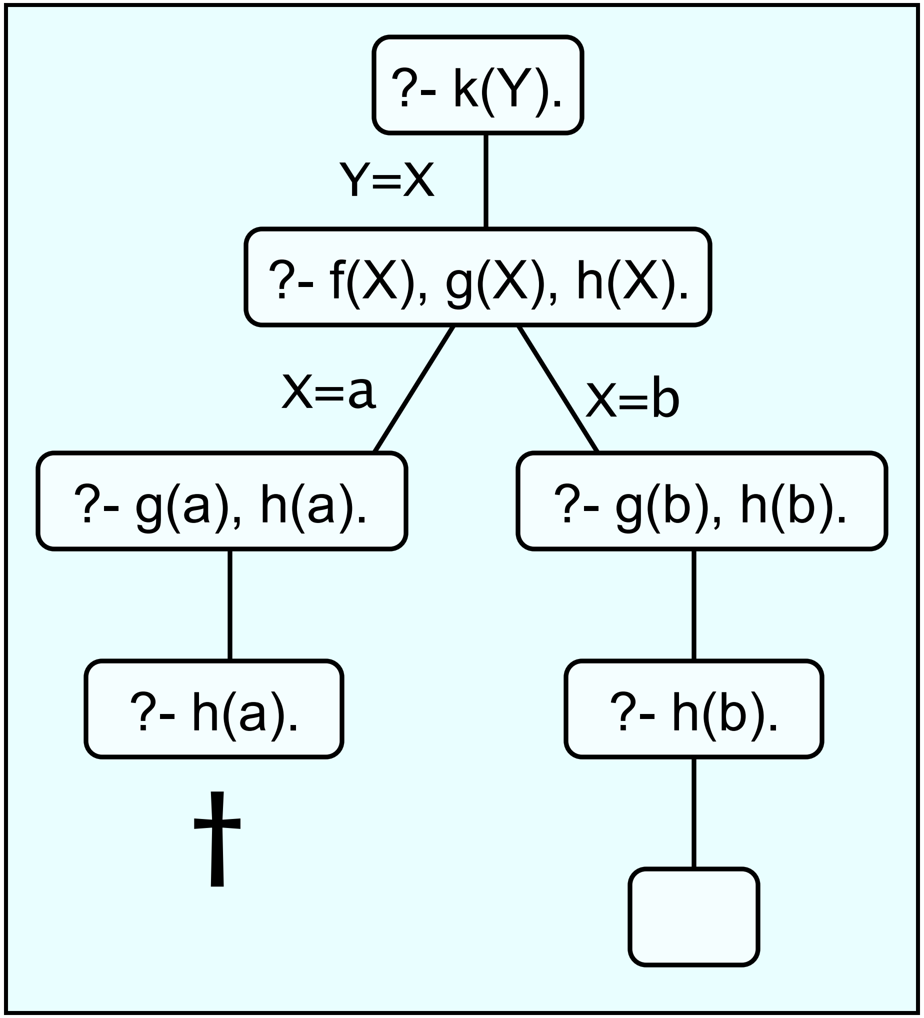
```
?- k(Y).
```



Example: search tree

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

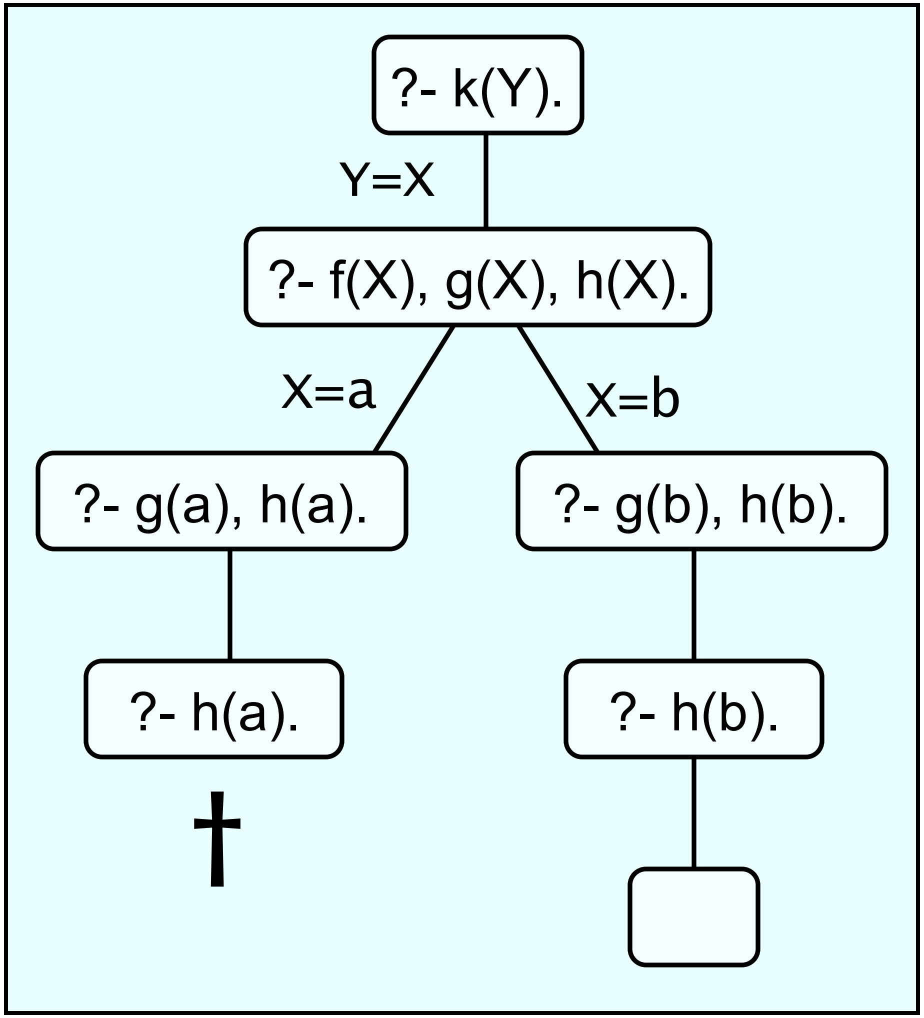
?- k(Y).
Y=b



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).
```

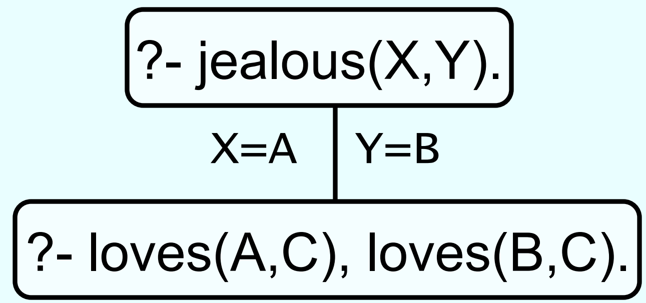
```
?- jealous(X,Y).
```

Another example

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

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

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

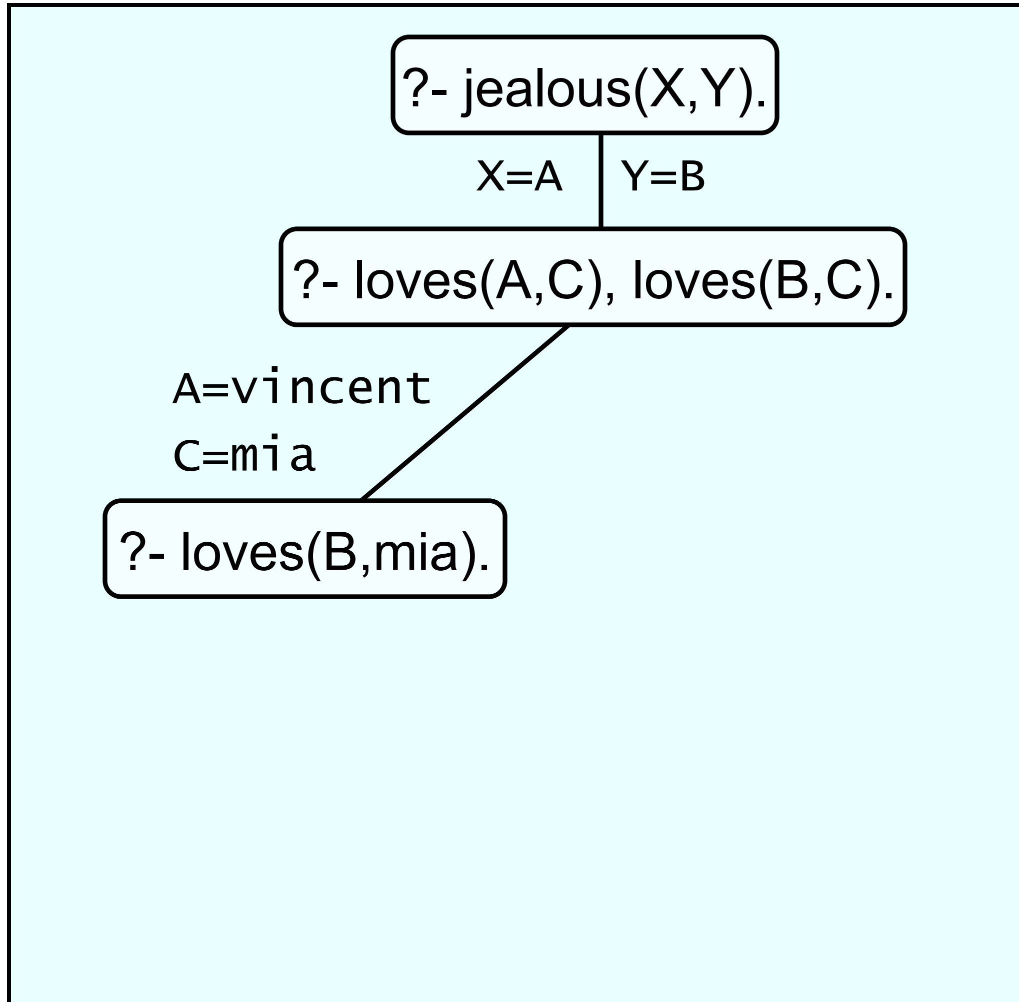


Another example

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

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

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

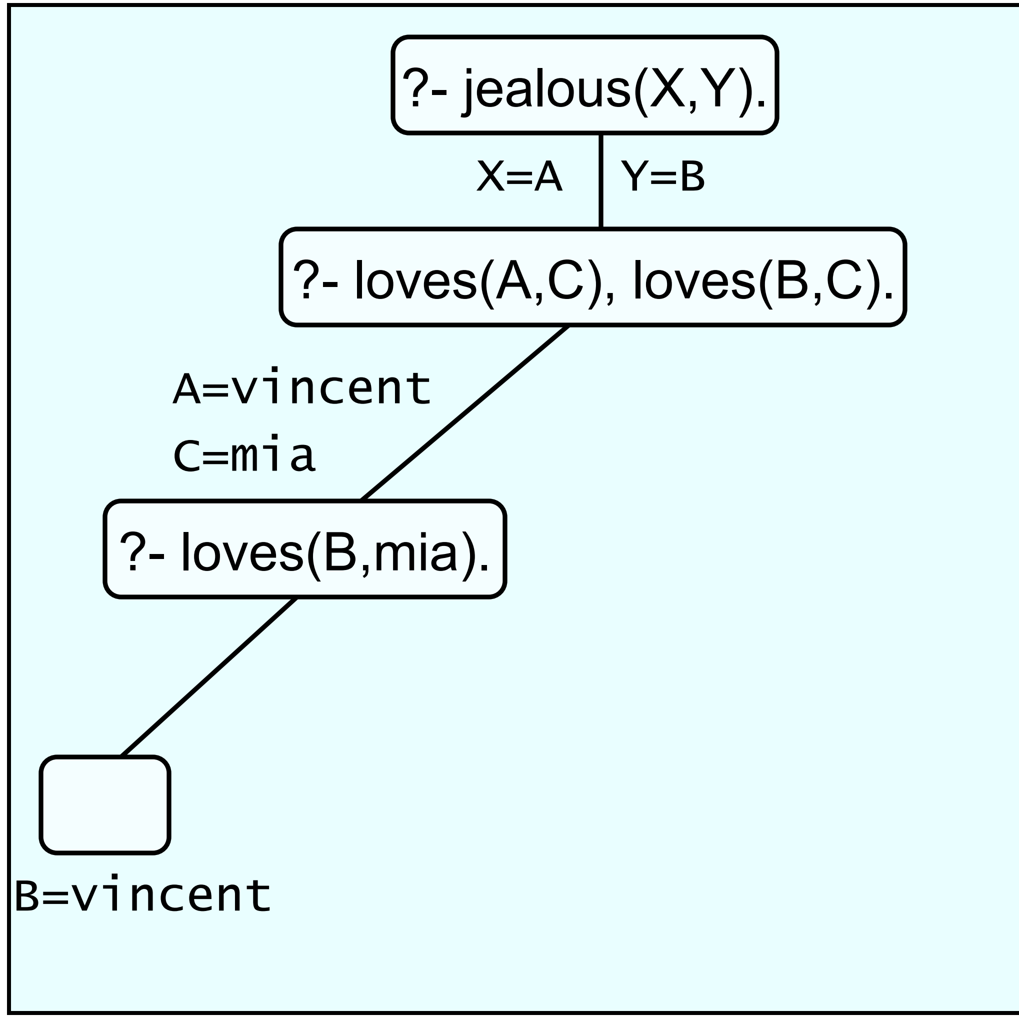


Another example

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

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

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

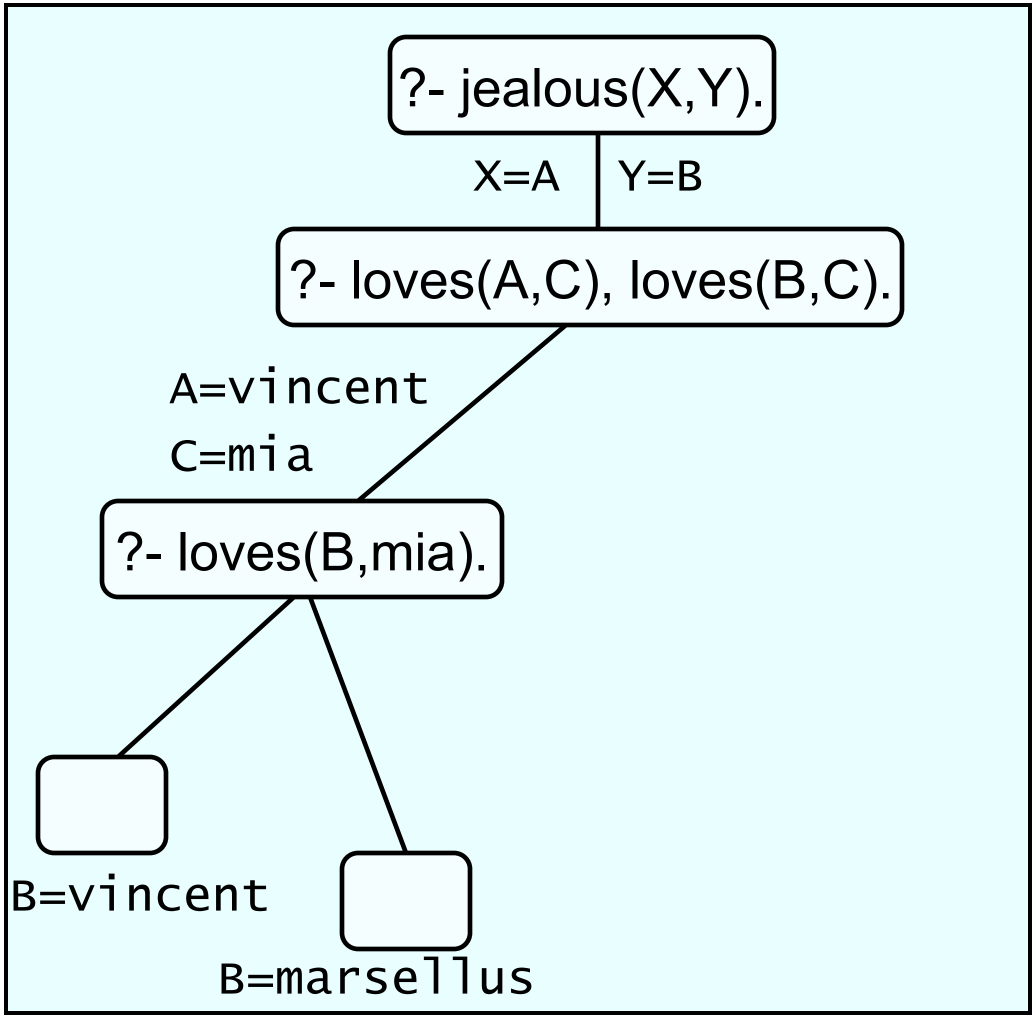


Another example

```
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
```

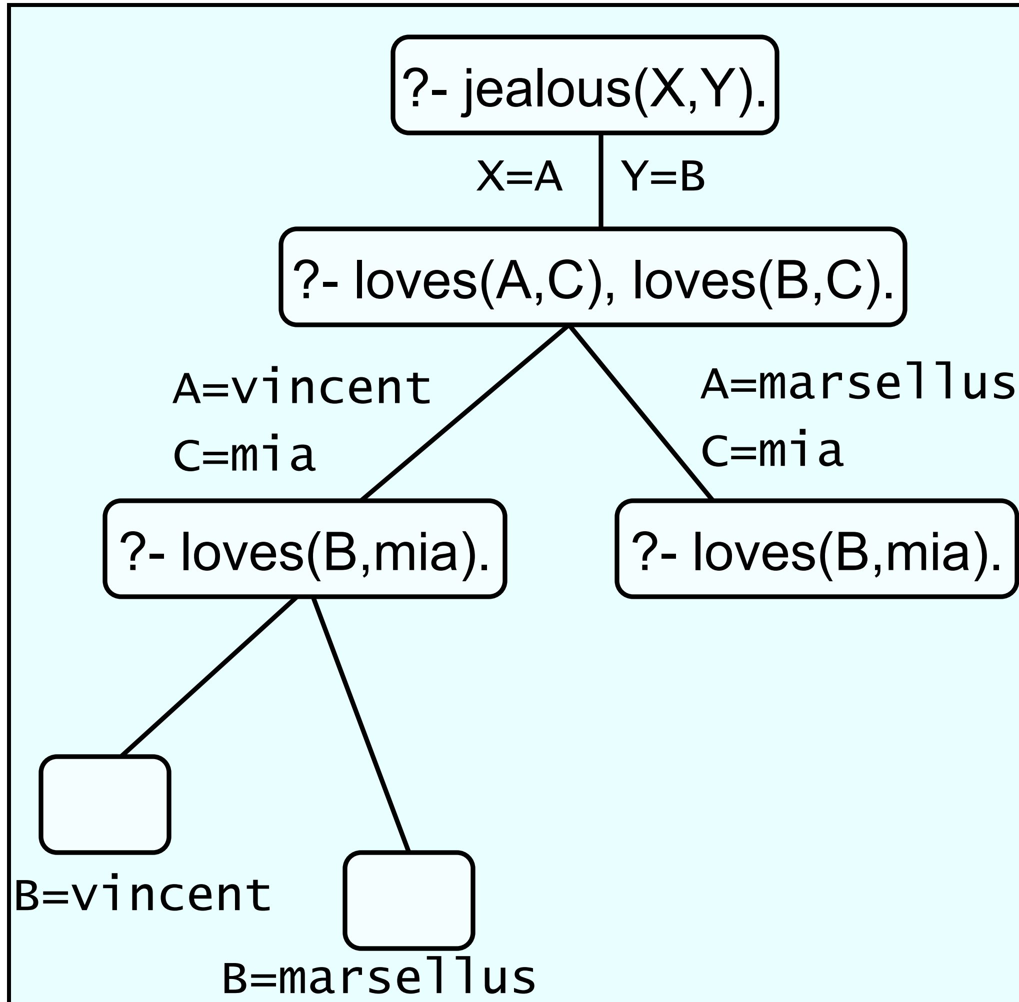


Another example

```
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;
```

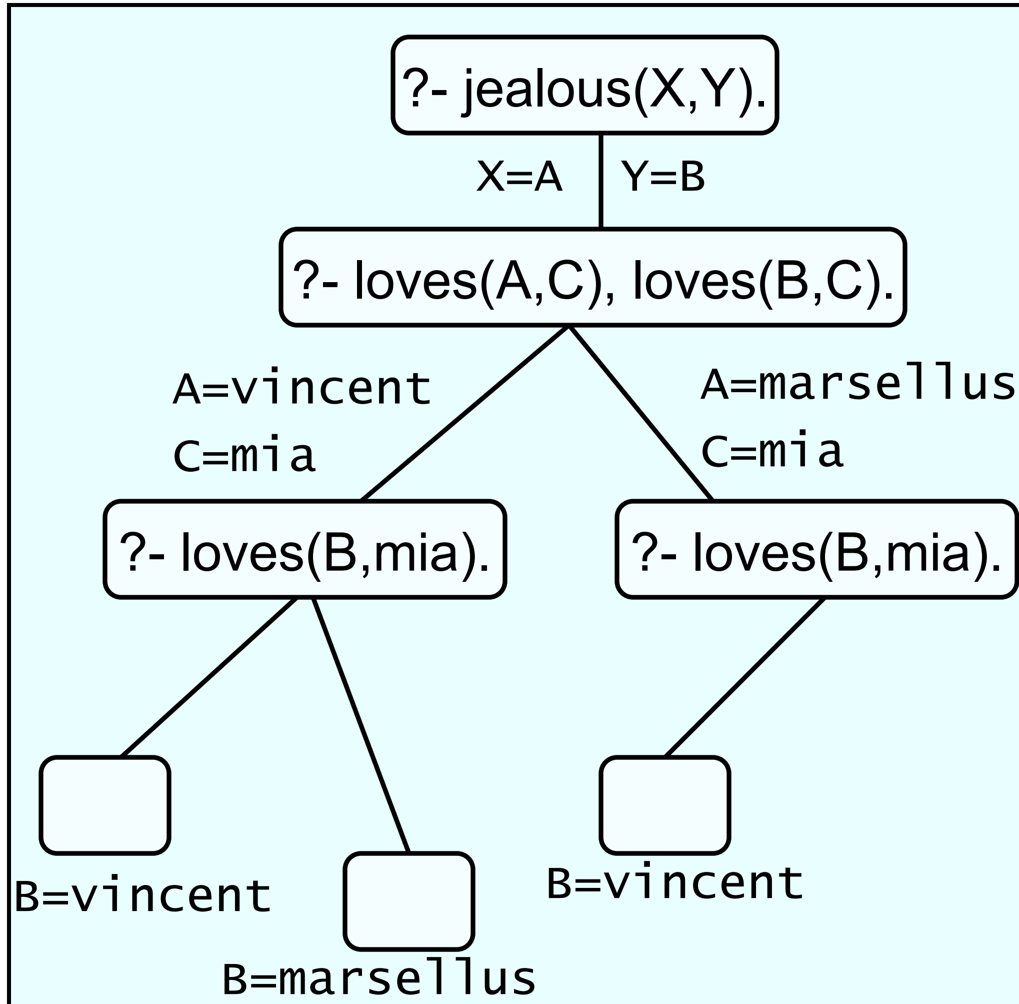


Another example

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

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

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

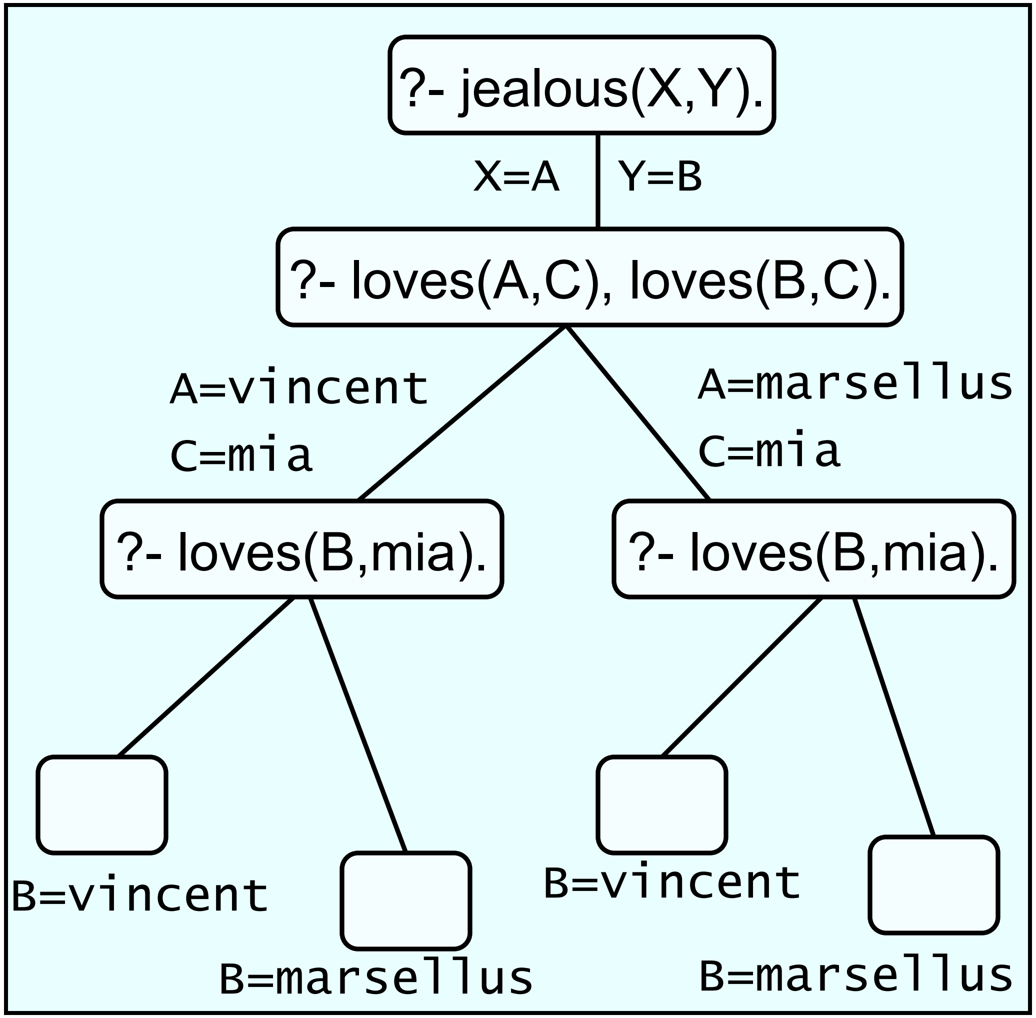


Another example

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

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

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

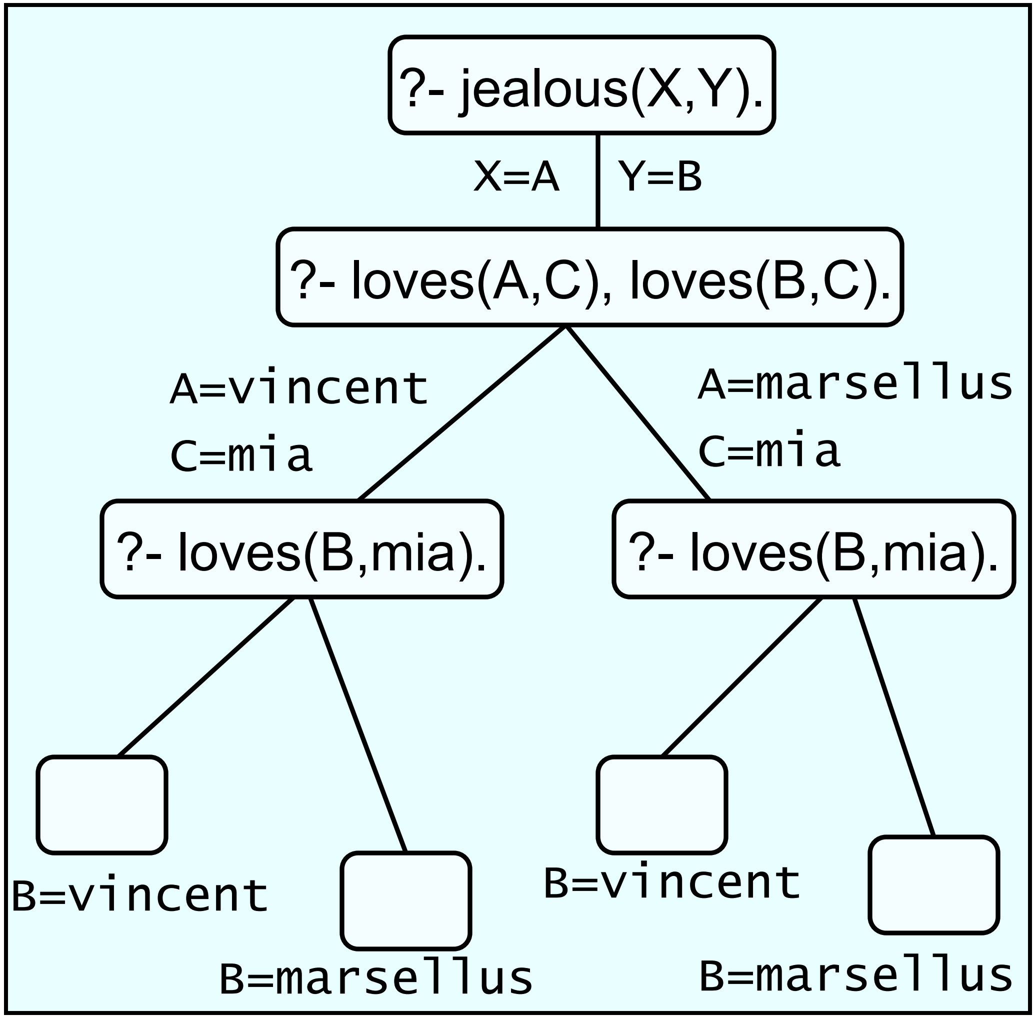


Another example

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

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

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



Exercises

Exercise 2.2b

We are working with the following knowledge base:

```
house_elf(dobby).
witch(hermione).
witch('McGonagall').
witch(rita_skeeter).
magic(X):- house_elf(X).
magic(X):- wizard(X).
magic(X):- witch(X).
```

Draw the search tree for:

```
?- magic(Hermione).
```

Next lecture

- Chapter 3 of LPN:
  Introducing **recursive definitions**

- Show that there can be mismatches between the <u>declarative</u> and <u>procedural</u> meaning in Prolog programs