



مجموعة جامعة
الجميع الذكية
للتعليم القابضة



Master of Science in Artificial intelligent

Lecture 9

AI511

Foundations of Artificial Intelligence

Prepared by
Prof. Lamiaa Fattouh Ibrahim

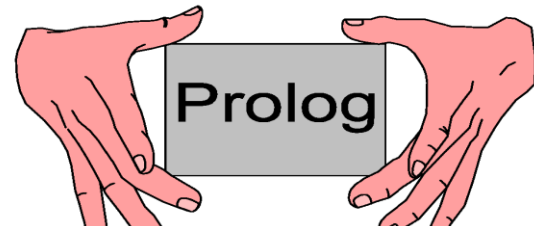
www.midocean.ae

OUTLINES

- Basics of PROLOG
- Facts
- Questions
- Variables
- Conjunction
- Rules

Recursion

- to be able to write recursive rules in Prolog;
- to be able to evaluate recursive rules.
- to be able to recognize (and avoid) infinite recursion;
- to appreciate the difference between declarative and procedural meanings for Prolog programs.



Textbook

- ◆ Title
 - PROLOG programming for artificial intelligence
- ◆ Author
 - Ivan Bratko



ENGLISH	PREDICATE CALCULUS	PROLOG
and	$\wedge$	,
or	$\vee$	;
Only if	$\leftarrow$	$:-$
not	$\neg$	not

Introduction

- ◆ What is Prolog?
 - A programming language for symbolic non-numeric computation
- ◆ What is programming in Prolog like?
 - Defining relations and querying about relations

“Prolog” = Programming in Logic

Imperative vs. declarative programming languages

- **Imperative:** describe the solution to a problem
- **Declarative:** describe the problem, let the computer find the solution

Prolog's view of the world:

- *relations* between *symbols* (which stand for entities in the real world)

Structure of programs

Facts: which relationships do we know to hold?

Rules: how can we derive new information about relationships from the one that we know?

Queries: given the facts and rules,
which *X*'s satisfy a certain relationship?

Relationships

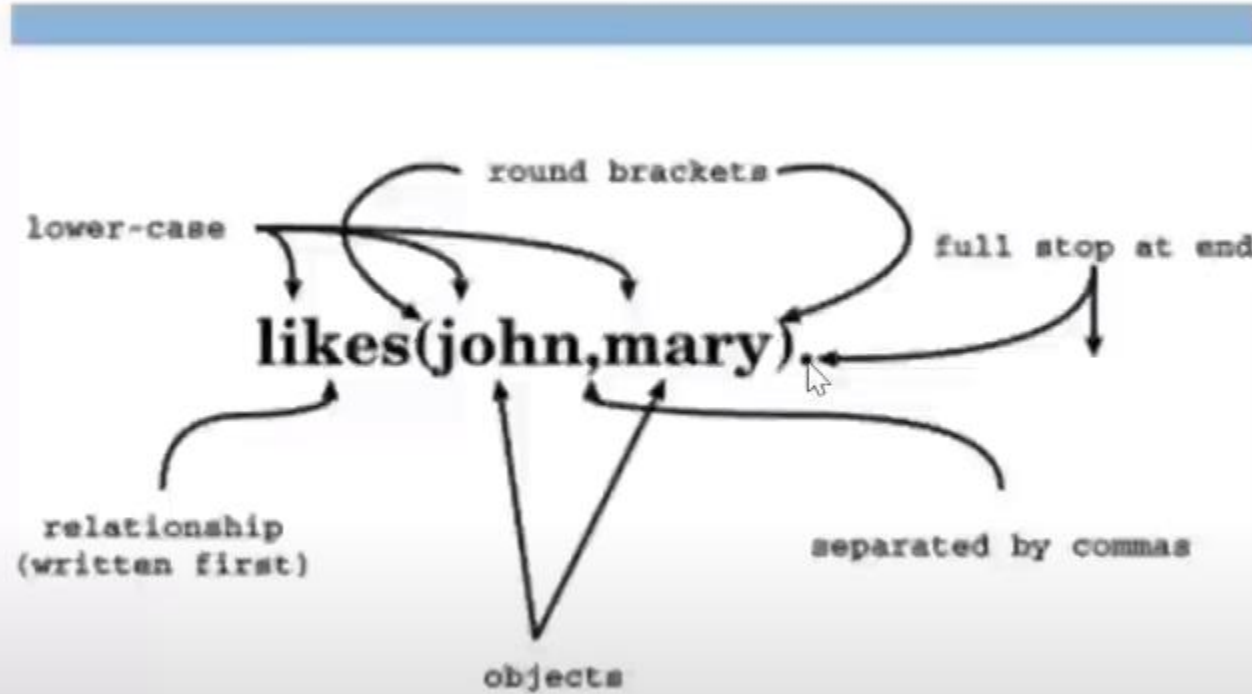
Example

John owns the book

- The relationship: *ownership*
- The objects: *book, John*

Directional:

- John owns the book
- **Not:** The book owns John



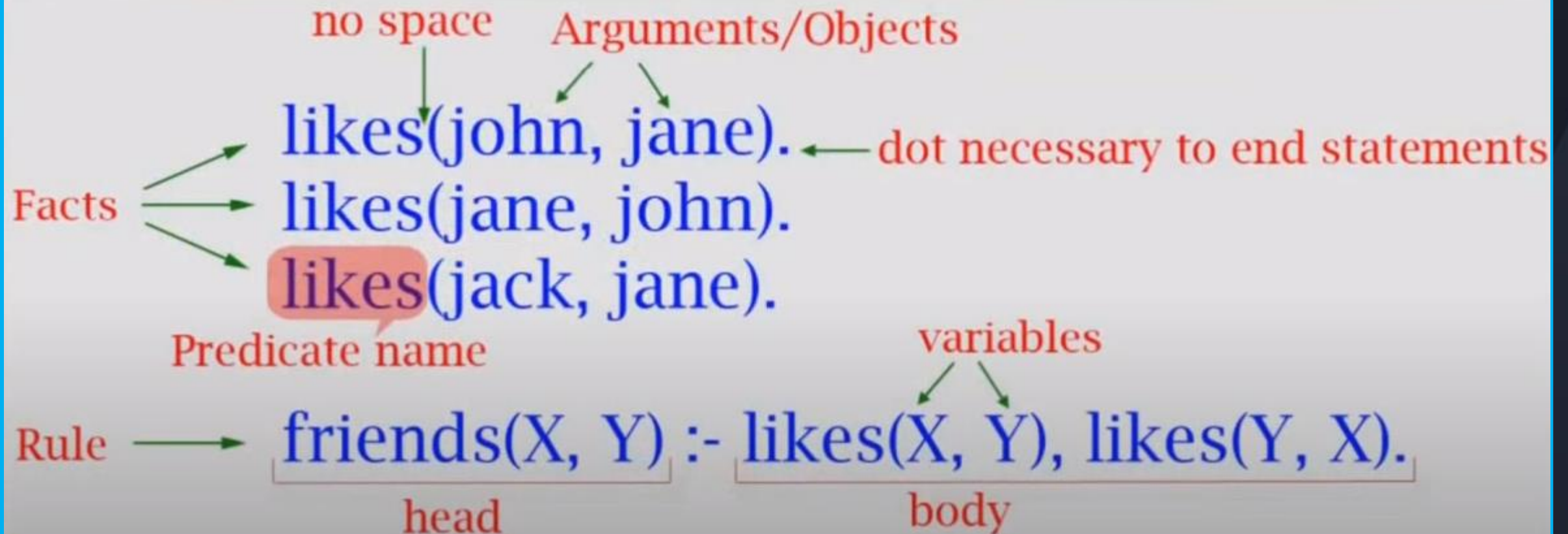
`likes(mary, john) .`

order defined by programmer

mary $\xrightarrow{\text{likes}}$ john

The fact says nothing
about how john likes mary

`john ...no info...> mary`



Example: Family Relations

Individuals (represented by symbols):

pam, tom, bob, liz, ann, pat, jim

The **relation** parent

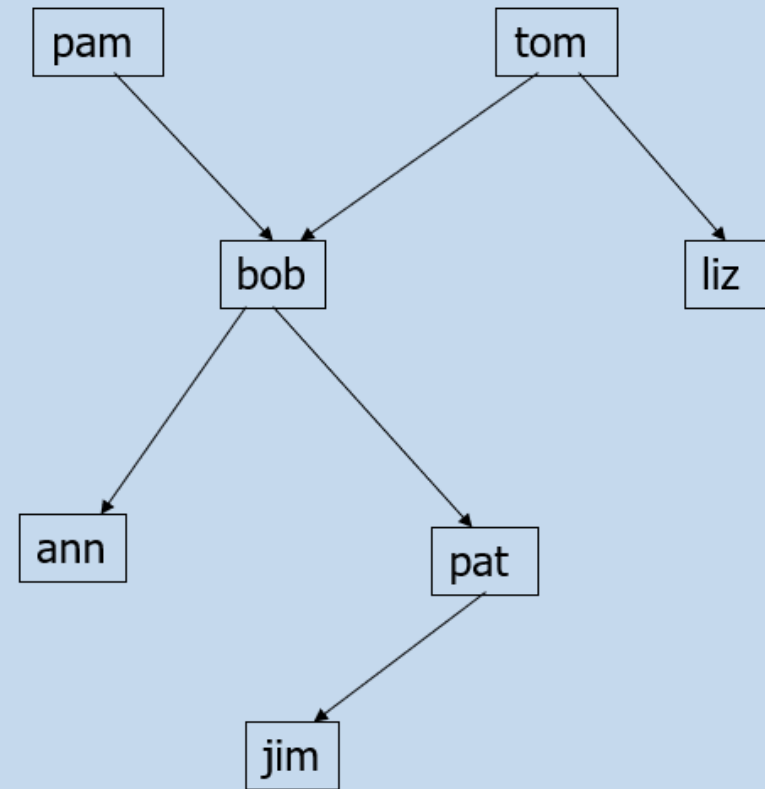
"parent(tom, bob)" means "Tom is a parent of Bob"

More *facts* about the parent relationships:

```
parent(pam, bob).    parent(tom, bob).  
parent(tom, liz).    parent(bob, ann).  
parent(bob, pat).    parent(pat, jim).
```

This is already a simple Prolog *program*. It consists of six *clauses*.

The parent facts can be visualized as a family tree



Family tree

Querying the Family Tree

We start the Prolog system in a shell, load our program into the system, and ask queries.

"Is Bob a parent of Pat?":

```
?- parent(bob, pat).
```

```
yes
```

Another query:

```
?- parent(liz, pat).
```

```
no
```

Also:

```
?- parent(liz, ben).
```

```
no
```

Queries with a Variable

"Who are Liz's parents?"

```
?- parent(X, liz).
```

```
X = tom
```

"Who are Bob's children?"

```
?- parent(bob, X).
```

```
X = ann
```

More answers! Type " ; "!

```
X = pat
```

Still more answers? Once more, type " ; "!

```
no
```

Queries with Several Variables

“Who is a parent of whom?”, or

Find X and Y such that X is a parent of Y !

In Prolog:

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

```
X = pam
```

```
Y = bob;
```

More answers!

```
X = tom
```

```
Y = bob;
```

More answers!

```
X = tom
```

```
Y = liz;
```

```
...
```

We can stop the stream of solutions by typing a return instead of a semicolon.

Composed Queries

“Who is a grandparent of Jim?”, or

1. Who is a parent of Jim? Assume it is some Y .
2. Who is a parent of Y ? Assume that this is some X .

In Prolog:

```
?- parent(Y, jim), parent(X, Y).
```

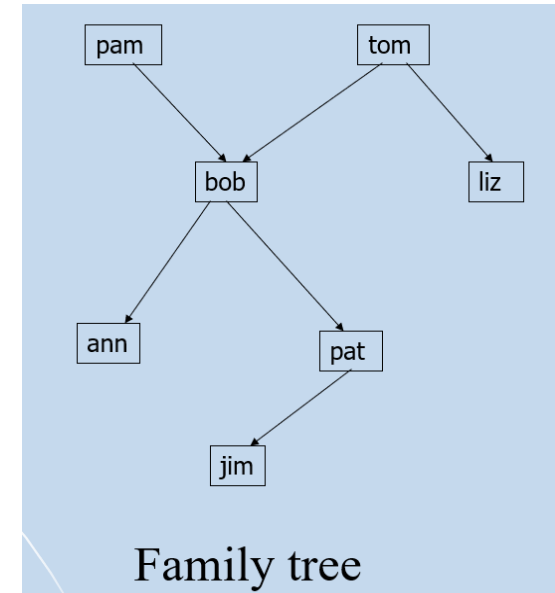
$X = \text{bob}$

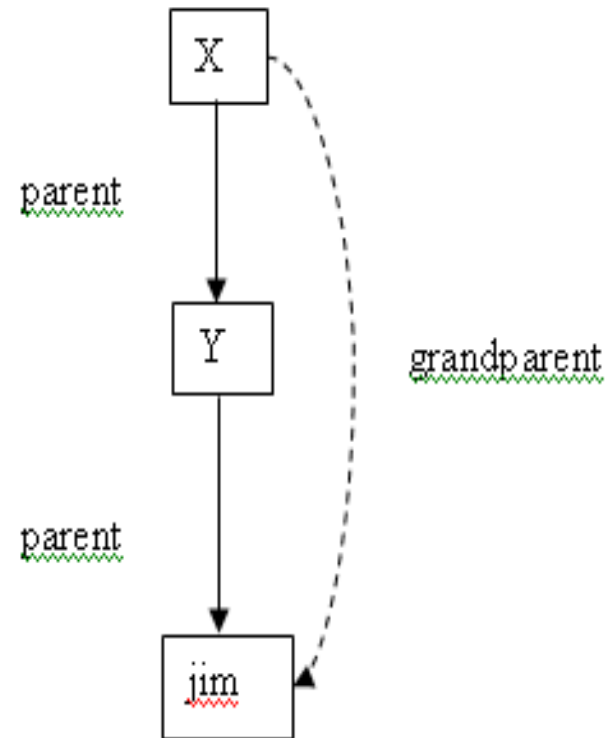
$Y = \text{pat}$

Imperative reading: Find such X and Y that satisfy the requirements

$\text{parent}(Y, \text{jim})$ and $\text{parent}(X, Y)!$

Logic $\leadsto$ The order can be changed without affecting the result!





In a similar way we can ask: Who are Tom's grandchildren?

?- parent(tom, X), parent(X, Y).

Prolog's answer are:

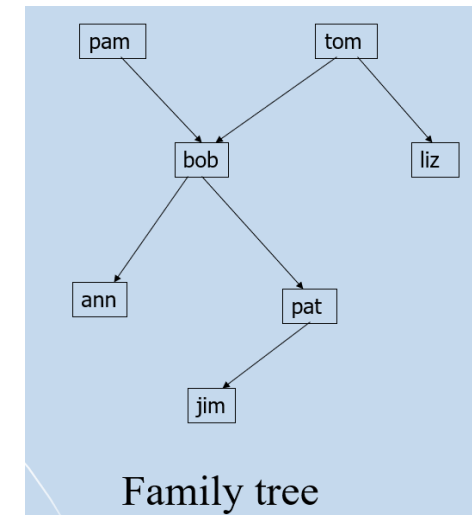
X = bob

Y = ann;

X = bob

Y = pat;

no



Yet another question could be:

Do Ann and Pat have a common parent? This can be expressed again in two steps:

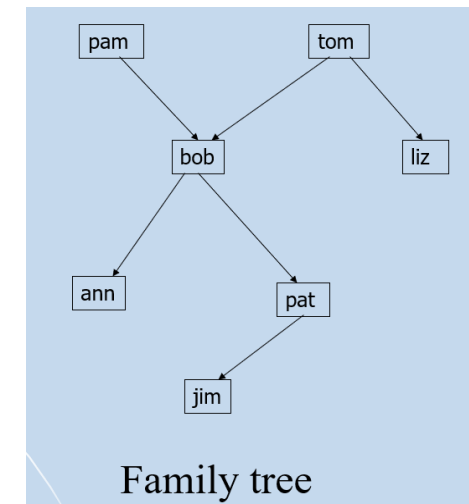
1. Who is a parent, X, of Ann?
2. Is (this same) X a parent of Pat?

The corresponding question to Prolog is then:

?- parent(X, ann), parent(X, pat).

The answer is:

X = bob



Prolog: First Insights

- A relation can be defined by stating the n -tuples that satisfy the relation.
There are also other ways!
- Prolog programs consist of *clauses*.
A clause terminates with a full stop.
- Possible arguments of a relation are: atoms (like tom and ann)
variables (like X and Y)
- Queries consist of one or more *goals*.
A *sequence* of goals means the *conjunction* of the goals.
- Answers can be positive or negative.
Positive answer: the goal was *satisfiable* and *succeeded*
Negative answer: the goal was *unsatisfiable* and *failed*
- If there are several answers, the user can let Prolog enumerate them.

Exercises

1. We assume that the relation `parent` is defined as in the example. What will be Prolog's answer to the following questions?

1. `:- parent(jim, X).`
2. `:- parent(X, jim).`
3. `:- parent(pam, X), parent(X, pat).`
4. `:- parent(pam, X), parent(X, Y), parent(Y, jim).`

2. Formulate in Prolog the following queries about the `parent` relation:

1. Who is Pat's parent?
2. Does Liz have a child?
3. Who is Pat's grandparent?

Defining New Relations Using Rules

We want to introduce the new relation `offspring`.

We could define `offspring` through new facts:

```
offspring(liz, pam).    ...
```

Or we can use the fact that `offspring` is the inverse of `parent`:

For all X and Y ,

Y is an offspring of X if

X is a parent of Y .

In Prolog, this can be expressed through the rule:

$$\underbrace{\text{offspring}(Y, X)}_{\text{head}} \quad :- \quad \underbrace{\text{parent}(X, Y)}_{\text{body}}.$$

Rules consist of a head and a body

- Rules have:
 - 1- A condition part (body) in the right-hand side of the rule.
 - 2- A conclusion part (head) in the left-hand side of the rule.
- A comma between two conditions indicates the conjunction of the conditions, meaning that both conditions have to be true.

Using Rules to Answer Queries

"Is Liz an offspring of Tom?"

```
?- offspring(liz, tom).
```

Instantiating the rule

```
offspring(Y, X) :- parent(X, Y).
```

by `X = tom` and `Y = liz` yields:

```
offspring(liz, tom) :- parent(tom, liz).
```

Applying the rule to the query goal yields a new goal:

```
?- parent(tom, liz).
```

Prolog answers

yes

More Rules

We add facts about the predicates `female` and `male` to our program:

```
female(pam).  male(tom).  male(bob).  
female(pam).  female(pat).  female(ann).  
male(jim).
```

We want to define the relation `mother`:

For all X and Y ,
 X is the mother of Y if
 X is a parent of Y and
 X is a female.

In Prolog:

```
mother(X, Y) :- parent(X, Y), female(X).
```

Rules with Existential Variables

We want to define the relation grandparent:

For all X and Y ,
 X is a grandparent of Y if
 there exists some Z such that
 X is a parent of Z and
 Z is a parent of Y .

In Prolog:

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Alternative reading:

For all X , Y , and Z
 if X is a parent of Z and
 Z is a parent of Y
 then X is a grandparent of Y .

Rules: Summary

- Prolog programs can be *extended* by adding *new clauses*.
- Prolog clauses are of *three* types: **facts**, **rules**, and **queries**.
 - *Facts* declare things that are always, unconditionally true.
 - *Rules* declare things that are true depending on a given condition.
 - By *queries* one can ask a program which things are true.
- Prolog clauses consist of a *head* and a *body*.
 - The body is a list of *goals* separated by commas.
 - Commas are understood as conjunctions.
- Facts are clauses with an *empty* body.
Queries *only* have a body.
Rules have *both*, a head and a non-empty body.

- Prolog gives us almost full freedom in choosing the layout of the program. So we can insert spaces and new lines as it best suits our taste.
- Relations such as female(pam) are unary (or one-place) relations.
- Binary relation like parent(tom , liz) defines a relation between pairs of objects.
- Unary relations can be used to declare simple yes/no properties of objects.
- Queries are answered by instantiating variables, creating new subgoals from rules and matching with facts.

Exercises

3. Translate the following statements into Prolog rules:
 1. Everybody who has a child is happy.
 2. For all X , if X has a child who has sister, then X has two children (introduce a new relation `hasTwoChildren`).
4. Define the relation `grandchild` using the `parent` relation.
5. Define the relation `aunt(X,Y)` in terms of the relations `parent` and `sister`.

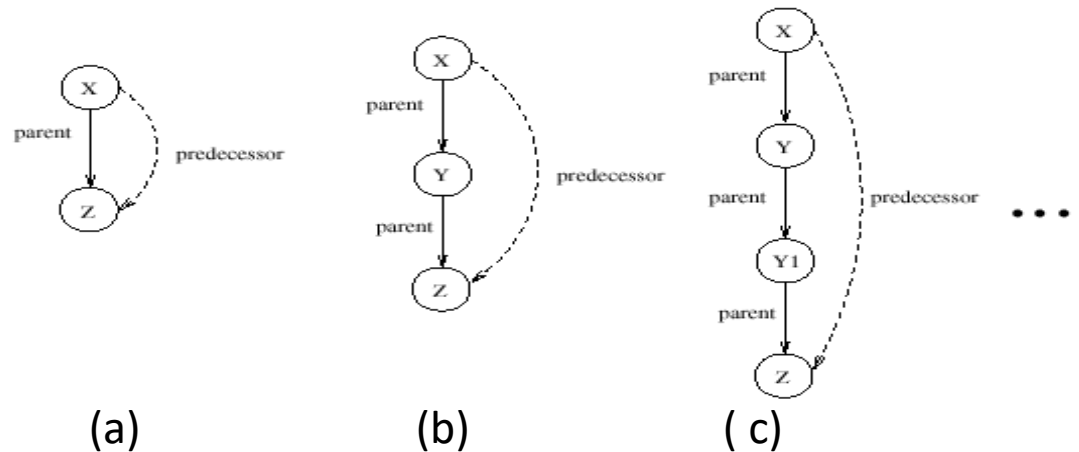
5. Define the relation $\text{aunt}(X,Y)$ in terms of the relations parent and sister .

The Predecessor Relation

Consider the family tree. We'd like to be able to define predecessor:

- a parent, or
- a parent of a parent, or
- a parent of a parent of a parent, or

...



Examples of the predecessor relation: (a) X is a direct predecessor of Z; (b) and (c) X is an indirect predecessor of Z

First Attempt

predecessor(X,Z) :- parent(X,Z).

predecessor(X,Z) :- parent(X,Y), parent(Y,Z).

predecessor(X,Z) :- parent(X,Y1), parent(Y1,Y2),
parent(Y2,Z).

...

Problems:

- Lengthy (not important)
- If we take the first two of these (or three or four etc) we get a finite depth on our search for predecessors.

Let's think...

Offspring's point of view:

You are my predecessor if either

- you are my parent, or
- you are a predecessor of my parent

Predecessor's point of view:

I am a predecessor of you if

- I am your parent, or
- I am a parent of your predecessor

Recursion

The predecessor's point of view can be expressed as

predecessor(X,Z):-

parent(X,Z).

predecessor(X,Z):-

parent(X,Y),

predecessor(Y,Z).

This type of definition is called a *recursive definition*. They are very important in Prolog.

A set of clauses referring to the same relation is known as a *procedure*.

Queries with Recursively Defined Relations

"Who are the descendants of Pam?"

```
?- predecessor(pam, X).
```

```
X = bob;
```

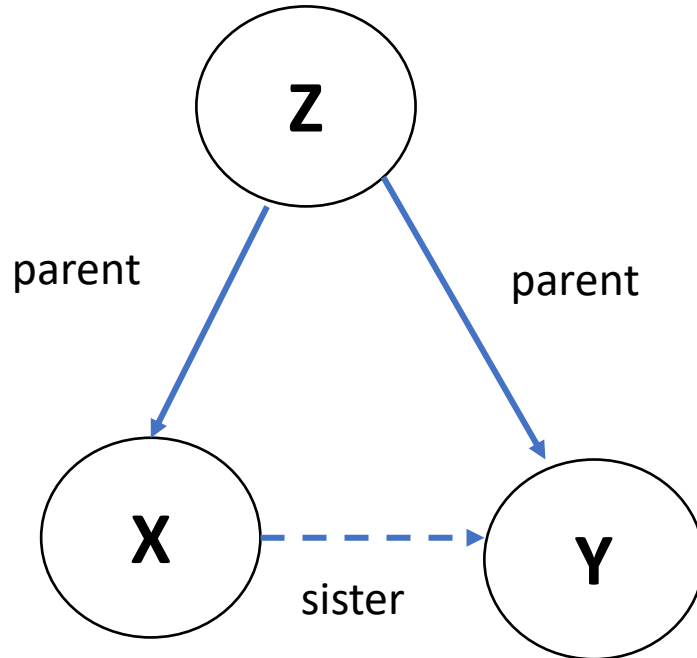
```
X = ann;
```

```
X = pat;
```

```
X = jim
```

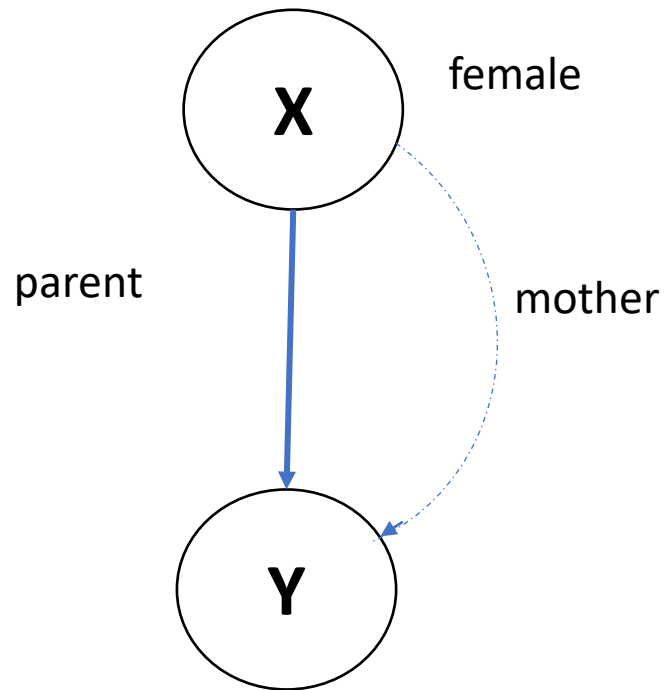
How did Prolog use the program to find these answers?

Defining the sister Relation



$\text{sister}(X, Y) \text{ :- } \text{parent}(Z, X), \text{parent}(Z, Y), \text{female}(X), \text{different}(X, Y).$

Defining the mother Relation



```
mother(X, Y) :- parent(X, Y), female(X).
```

The Family Program

```
parent(pam, bob).      female(pam).      male(tom).
parent(tom, bob).      female(pam).      male(bob).
parent(tom, liz).      female(pat).      male(jim).
parent(bob, ann).      female(ann).
parent(bob, pat).
parent(pat, jim).

offspring(Y, X) :- parent(X, Y).

mother(X, Y) :- parent(X, Y), female(X).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), different(X,Y).

predecessor(X, Y) :- parent(X, Y).
predecessor(X, Y) :- parent(X, Z), predecessor(Z, Y).
```

6. We give an alternative definition of predecessors with the relation predecessorMod:

```
predecessorMod(X, Z) :-  
    parent(X, Z).
```

```
predecessorMod(X, Z) :-  
    parent(Y, Z),  
    predecessorMod(X, Y).
```

Is this also a proper definition of predecessors?

Do you expect the same answers as for the previous one, if you use it in queries instead of predecessor?

Programming with matching

```
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 matching

```
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 matching

```
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 matching

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

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

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

Point = point(Z,3); or Point = point(_554,3);

no

?-

- Now that we know about matching, 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).

?- k(Y).

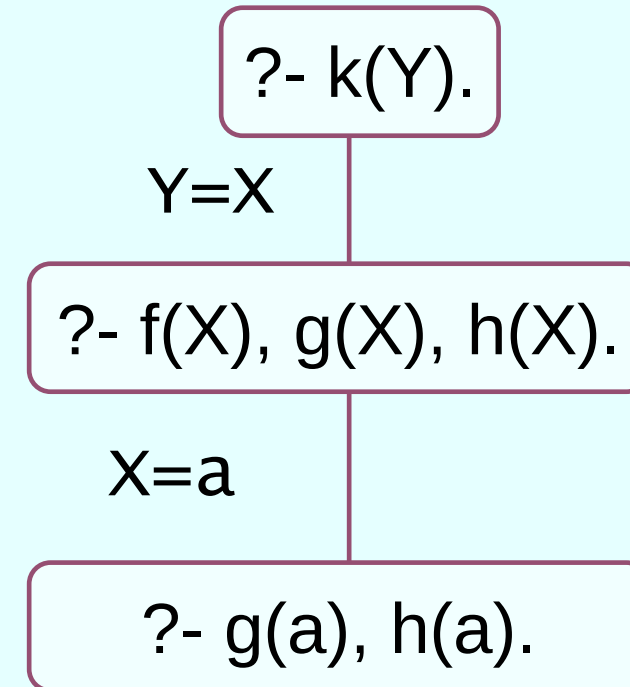
Y=X

?- f(X), g(X), h(X).

Example: Search Tree

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

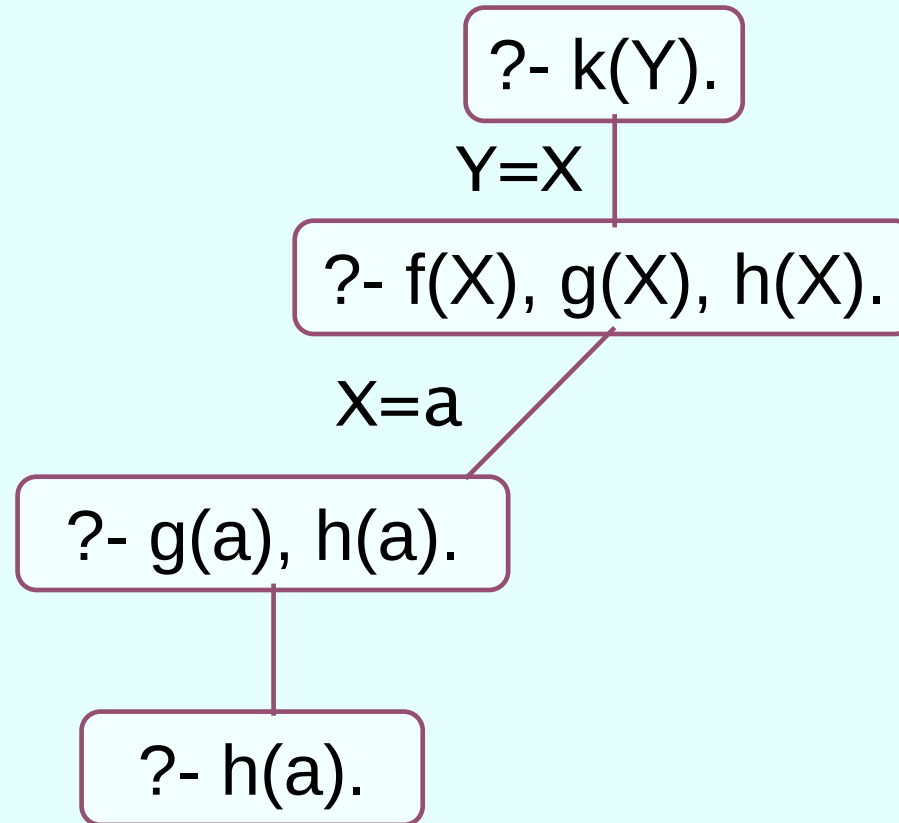
?- k(Y).



Example: Search Tree

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

?- k(Y).

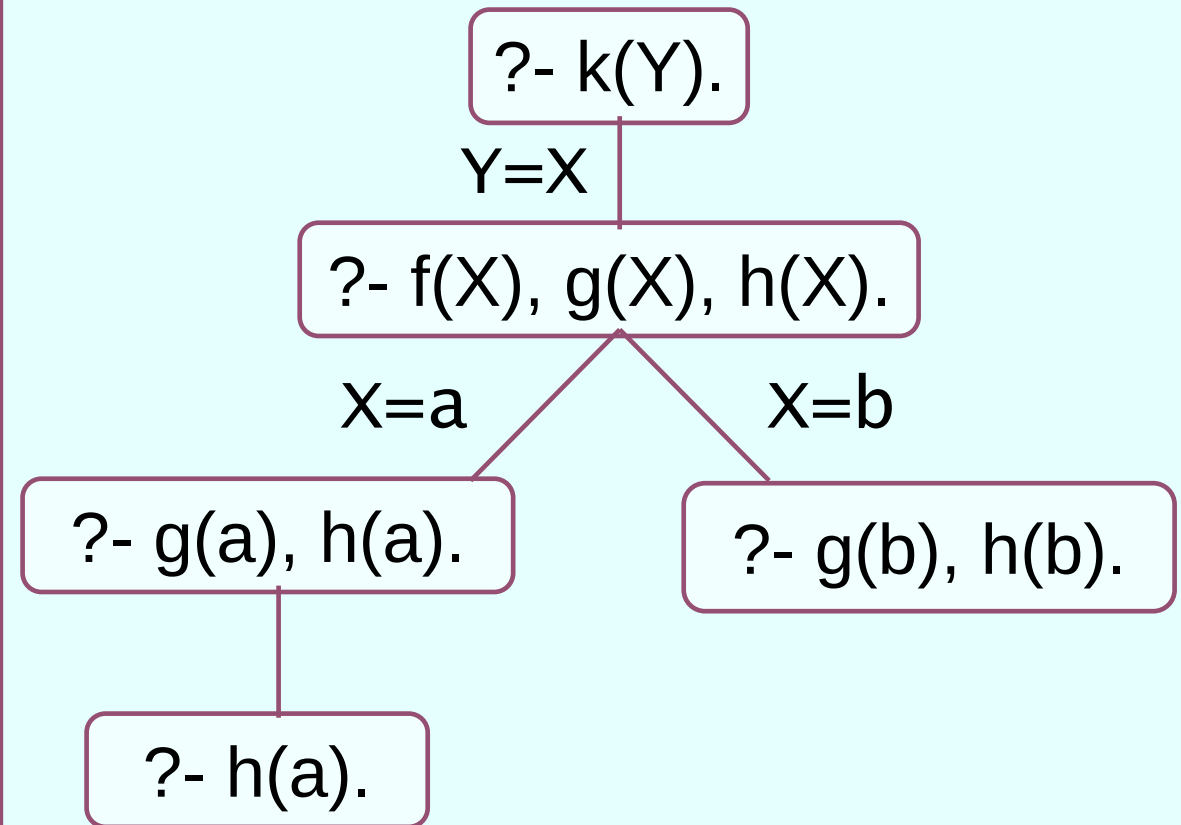


Fail

Example: Search Tree

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

?- k(Y).

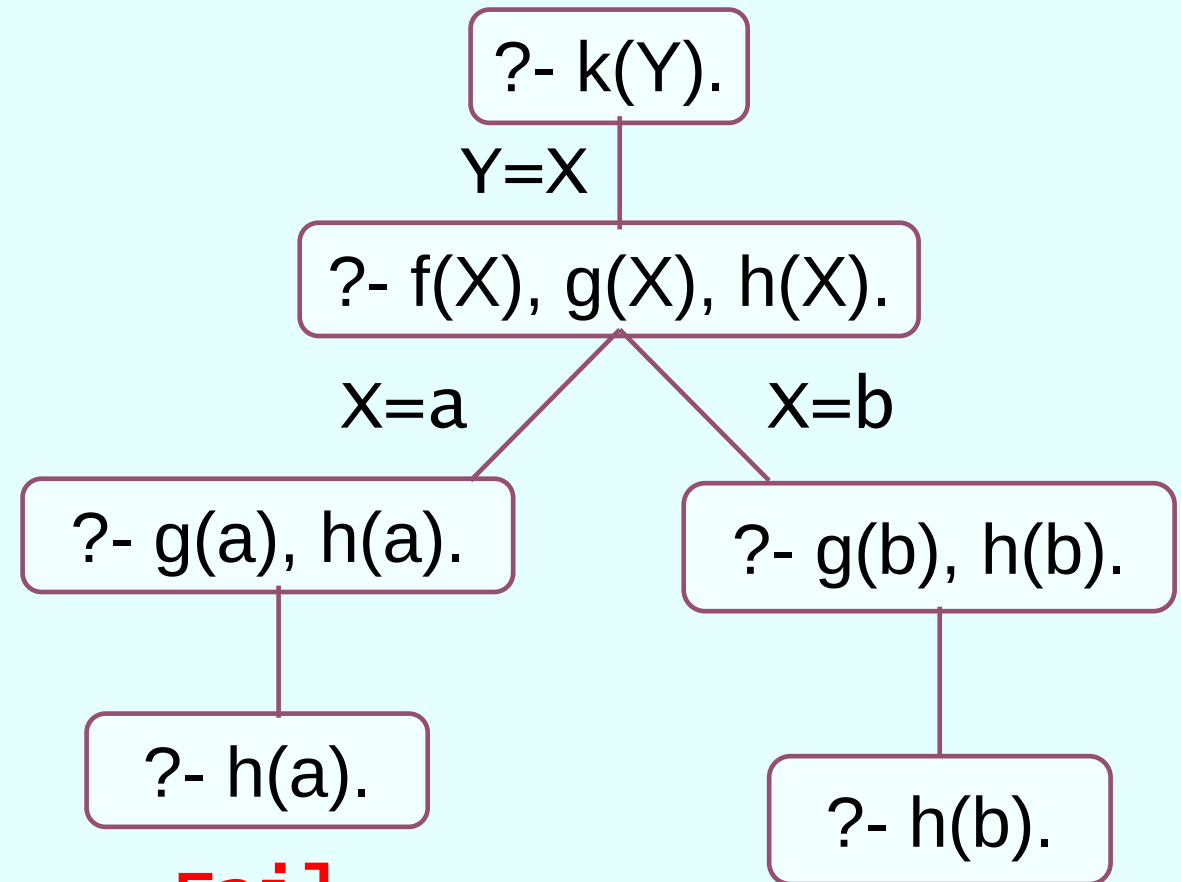


Fail

Example: Search Tree

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

?- k(Y).

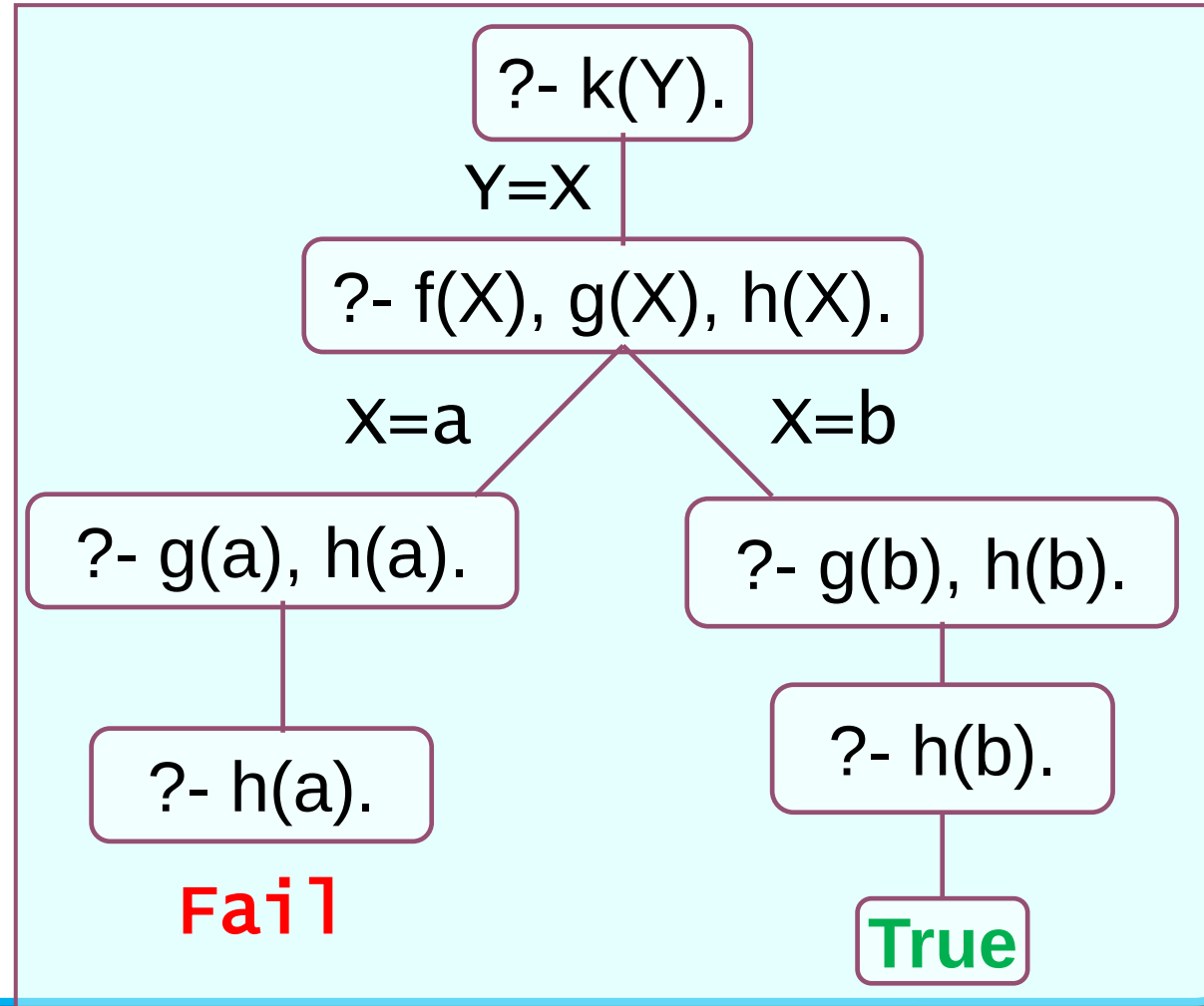


Fail

Example: Search Tree

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

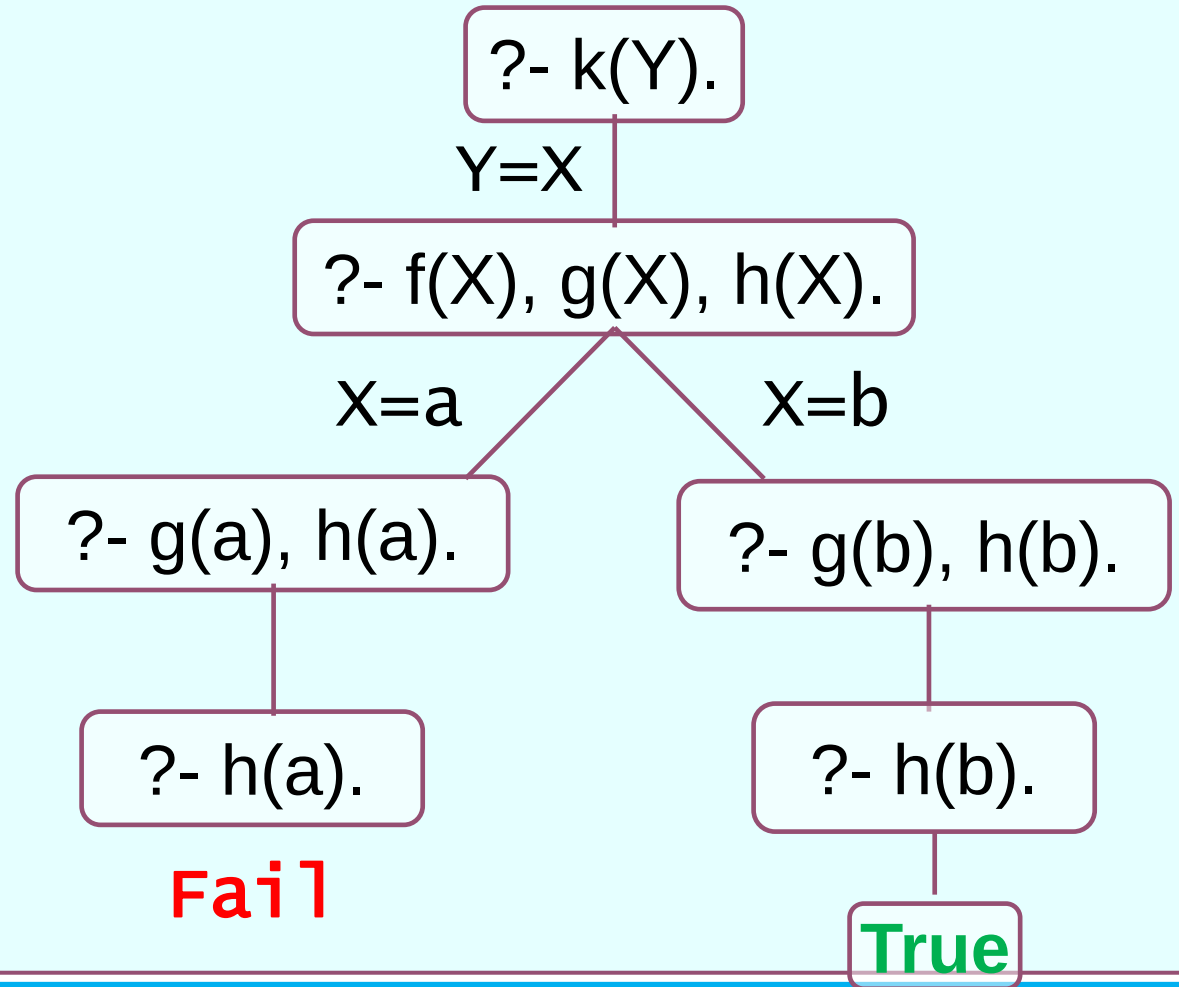
?- k(Y).
Y=b



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).
```

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

X=A

Y=B

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

A=vincent
C=mia

```
?- loves(B,mia).
```

Another example

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

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

?- jealous(X,Y).

X=A

Y=B

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

A=vincent
C=mia

?- loves(B,mia).

True

B=vincent

Another example

```
loves(vincent,mia).  
loves(marsellus,mia).  
jealous(A,B):- loves(A,C), lo  
ves(B,C).
```

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

?- jealous(X,Y).

X=A

Y=B

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

A=vincent
C=mia

?- loves(B,mia).

True

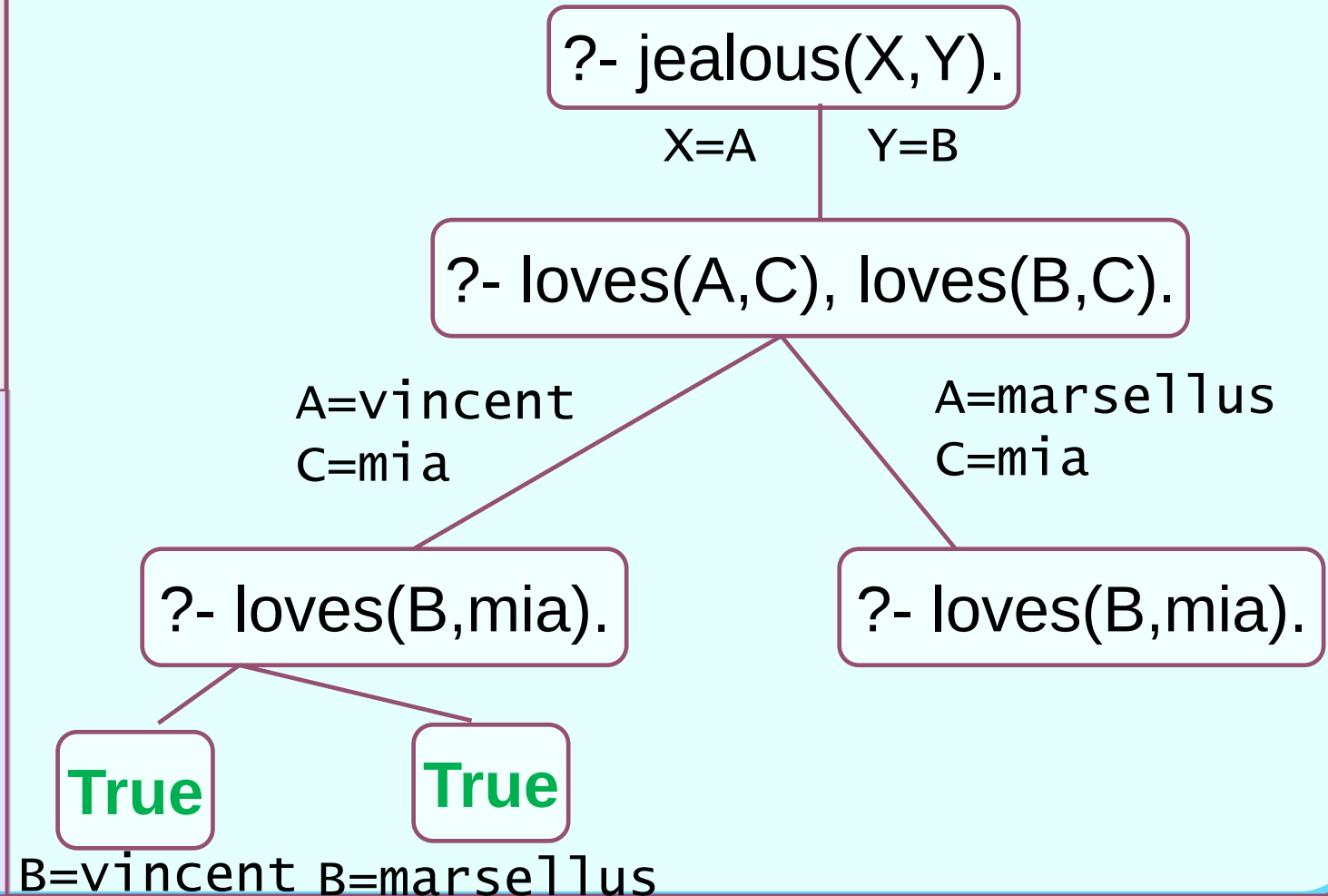
True

B=vincent B=marsellus

Another example

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

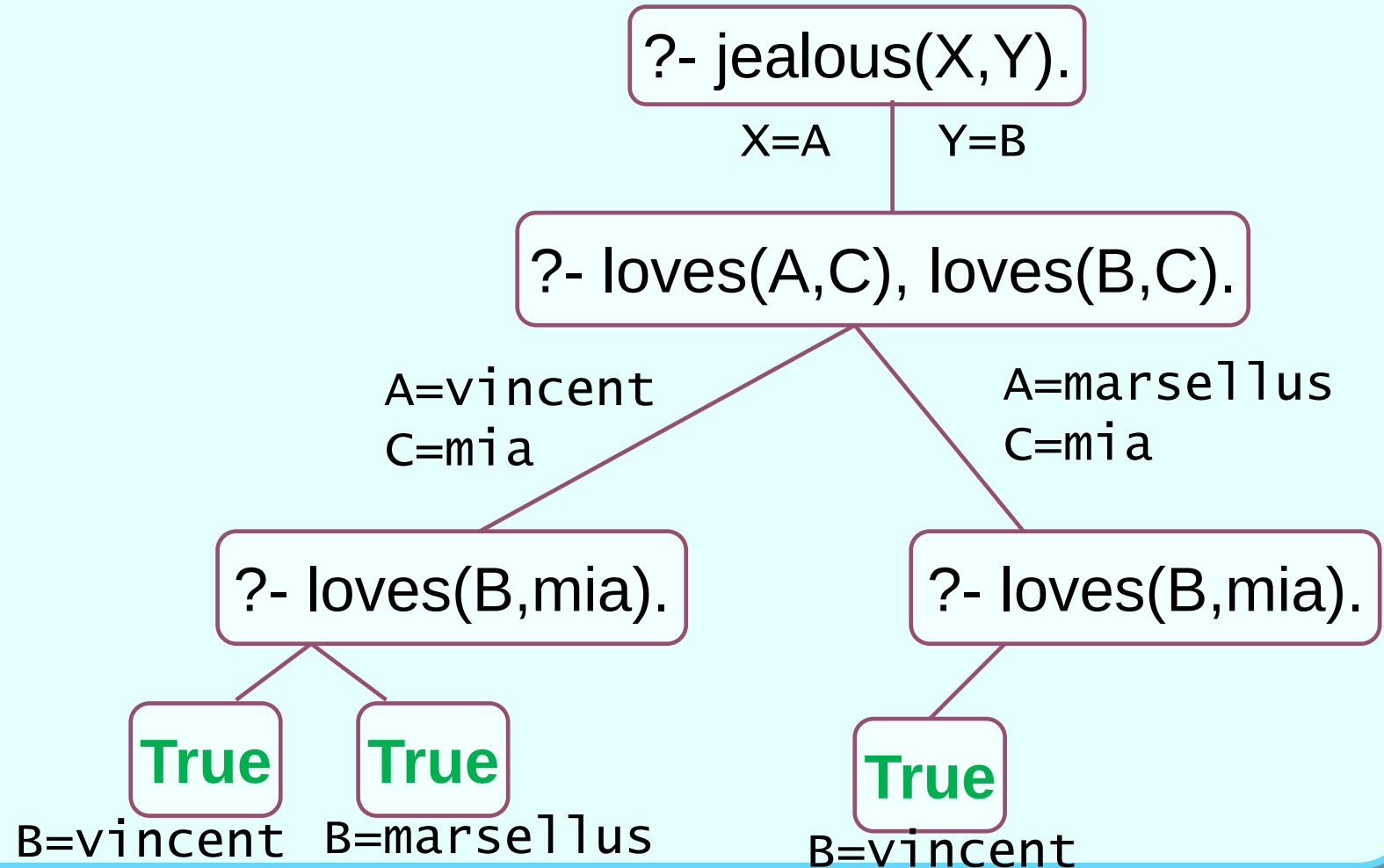
```
?- jealous(X,Y).  
X=vincent  
Y=vincent;  
X=vincent  
Y=marsellus;
```



Another example

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

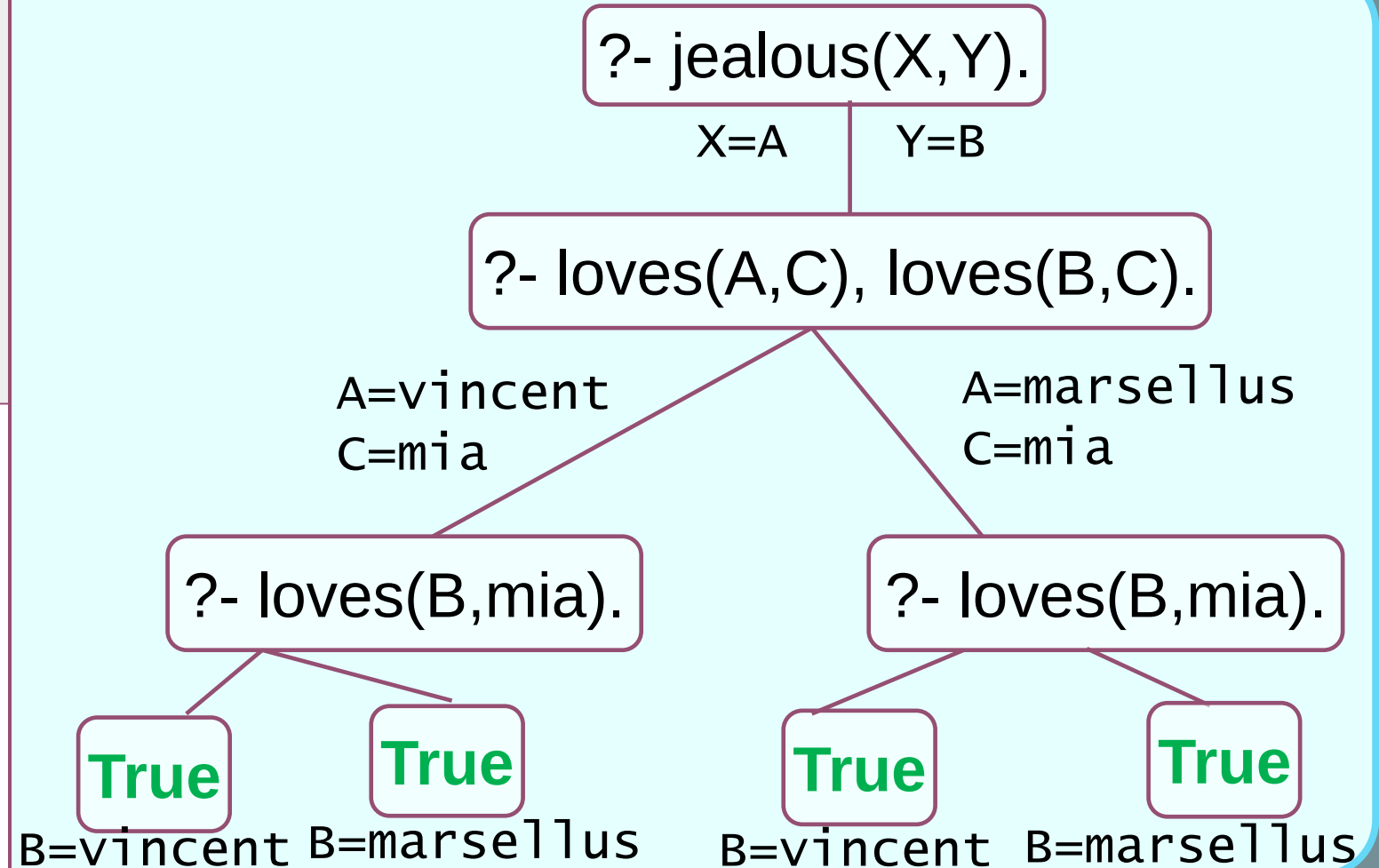
```
....  
X=vincent  
Y=marsellus;  
X=marsellus  
Y=vincent
```



Another example

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

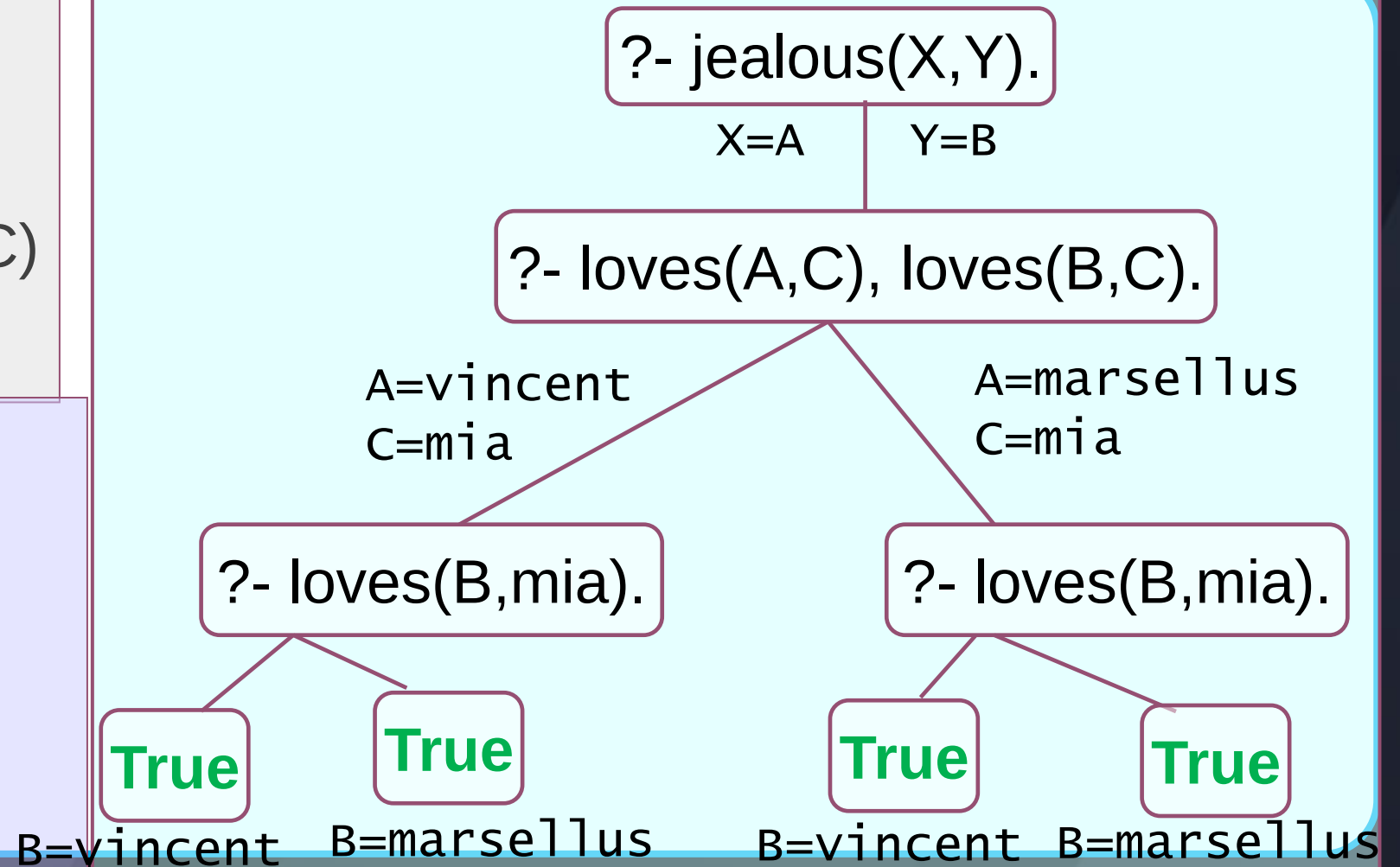
```
....  
X=marsellus  
Y=vincent;  
X=marsellus  
Y=marsellus
```



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
```



Prolog Download



Did you know? You can get **COLORED** text on the command line

Search Documentation:



SWI Prolog

Robust, mature, free. **Prolog for the real world.**

- HOME
- DOWNLOAD
- DOCUMENTATION
- TUTORIALS
- COMMUNITY
- COMMERCIAL
- WIKI

- Manual
- Packages
- FAQ
- Command line
- PLDoc
- Bluffers
- License
- Publications
- Rev 7 Extensions

SWI-Prolog offers a complete Prolog environment. Since its start in 1987, SWI-Prolog has been driven by the needs of real world applications. SWI-Prolog is used in research and education as well as commercial applications. [more ...](#)

SWI-Prolog offers a complete Prolog environment. Since its start in 1987, SWI-Prolog has been driven by the needs of real world applications. SWI-Prolog is used in research and education as well as commercial applications. [more ...](#)

[Download SWI-Prolog](#)

[Get Started](#)

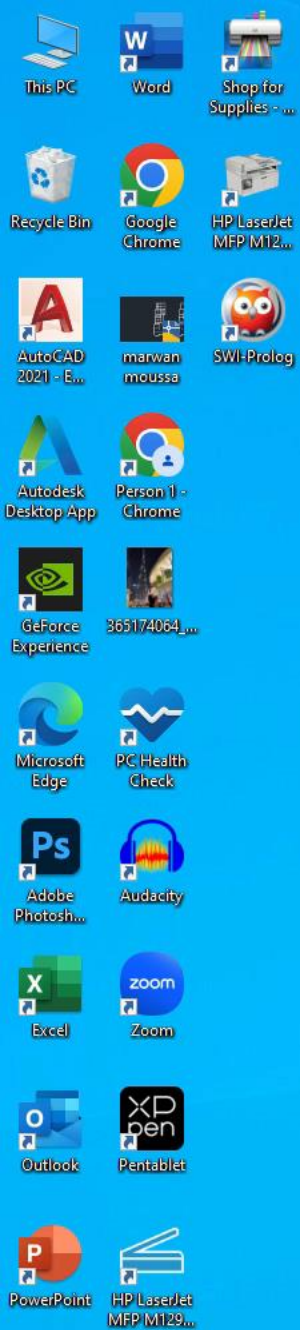
[Try SWI-Prolog online \(SWISH\)](#)

🔥 [Try SWI-Prolog in your browser \(WASM\)](#)

🌟 Star 845 💖 Sponsor

SEARCH DOCUMENTATION: [GO](#)

Prolog Usage



SUMMARY

Some important points of this section are:

- Prolog programs can be extended by simply adding new clauses.
- Prolog clauses are of three types: *facts*, *rules* and *questions*.
- *Facts* declare things that are always, unconditionally true.
- *Rules* declare things that are true depending on a given condition.
- By means of *questions* the user can ask the program what things are true.
- Prolog clauses consist of the *head* and the *body*. The body is a list of *goals* separated by commas. Commas are understood as conjunctions.
- Facts are clauses that have a head and the empty body. Questions only have the body. Rules have the head and the (non-empty) body.
- In the course of computation, a variable can be substituted by another object. We say that a variable becomes *instantiated*.
- Variables are assumed to be universally quantified and are read as 'for all'. Alternative readings are, however, possible for variables that appear only in the body. For example:

```
hasachild( X) :- parent( X, Y).
```

can be read in two ways:

- (a) *For all X and Y,*
if X is a parent of Y then
X has a child.
- (b) *For all X,*
X has a child if
there is *some* Y such that X is a parent of Y.



مجموعة جامعة
الجميع الذكية
للتعليم القابضة

MIDOCEAN
UNIVERSITY
BY EVERYONE'S SMART UNIVERSITY



Thank You

