Prolog : first steps

a programming language for computational linguistics

Note: many of the overheads here are taken from, or adapted from, the excellent online course for learning Prolog: **www.learnprolognow.org** by **Patrick Blackburn, Johan Bos & Kristina Striegnitz.** Most of the places where this is done there is a copyright reference to them, so they are to be quoted and credited if you take anything from here!
There is also a book that you can buy giving the introduction step-by-step. Since we will only be using a little bit of this in our brief intro to Prolog, you do not need to work you way through this for this course, although if you are interested, you can certainly take it further on your own!

Prolog

- “Programming in Logic”
- Devised for writing algorithms in a way that allows us to focus on just what we need to say without any extra baggage
- Very simple; but very powerful.
- Just two essential mechanisms
  - backtracking
  - recursion

Session 1: basic operations

- knowledge base
- following rules
- working out answers
  - matching variables
  - searching
  - backtracking
  - recursion
  - lists
  - some basic programmes

see, for more details: http://www.learnprolognow.org/

Basic Objects

- facts
- rules
- queries

Knowledge Base: facts

- Consists first of a collection of ‘facts’, that is, simple assertions…
  - male(john).
  - female(mary).
  - male(peter).
  - childof(peter,mary).
  - female(sarah).
  - childof(john,sarah).
  - childof(sarah,peter).

Write these into a file and save to somewhere you can find again! The file should be called something like **`facts.pl`**

Knowledge Base: rules

- We can also write rules to apply to the knowledge base:
  - son(X) :- childof(X,Z), male(X).

How does Prolog know when to apply rules?

matching / unification

Basic Data Types

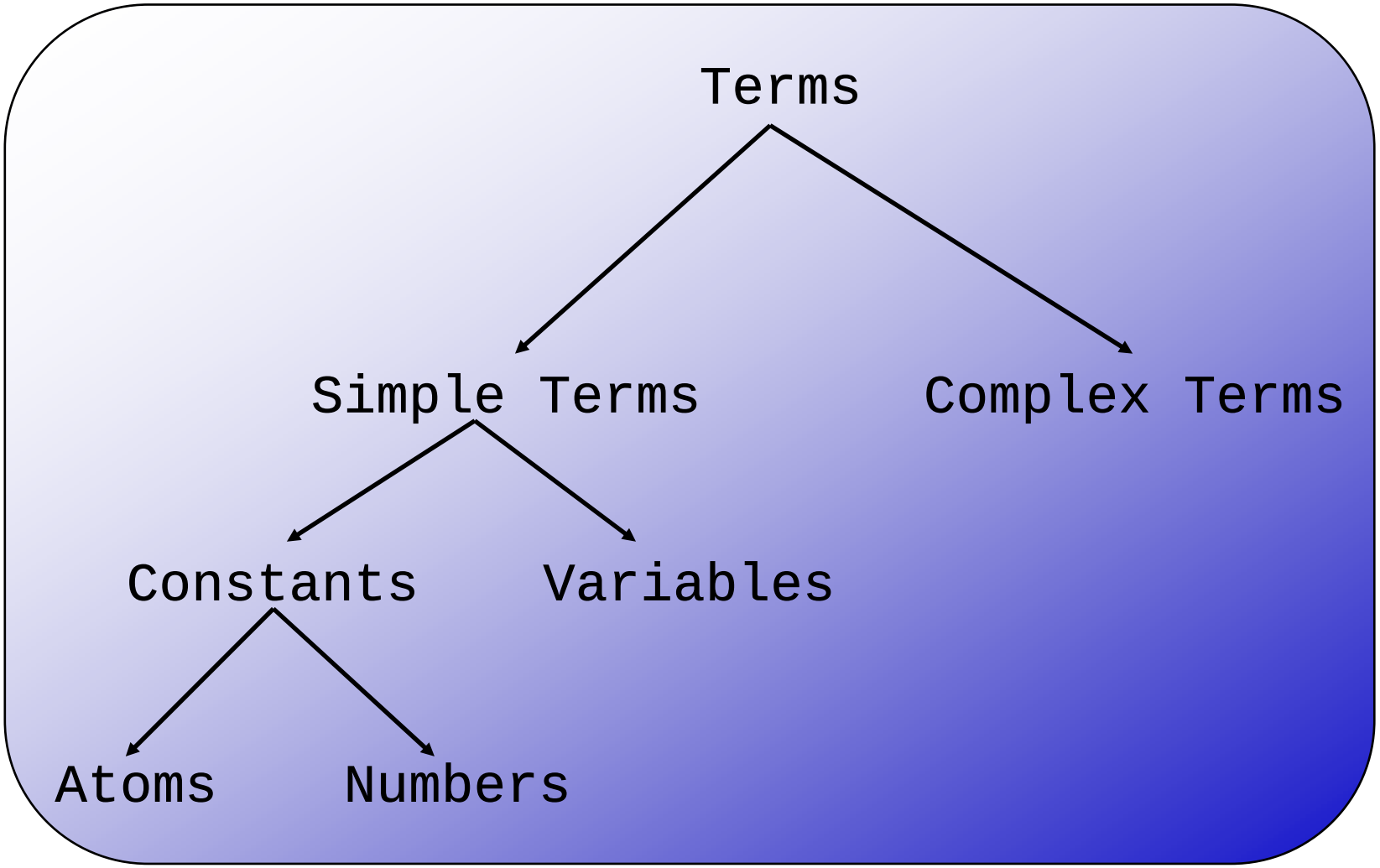
- simple terms
  - constants
    - atoms
    - numbers
  - variables

- complex terms (structures)
  are built out of:
  - functors (an atom) *and*
  - arguments

  *the number of arguments gives the functor's '**arity**'*

- lists

Basic Data Types



Matching / Prolog 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

Testing this out

- Startup Prolog and type in:
  - mia = mia.
  - vincent = mia.

Unification

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

i.e., when we use a VARIABLE?

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)**

How will Prolog respond?

?- X=mia, X=vincent.

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).

Unification

- The concept of unification is one of the main ideas behind logic programming (e.g., Prolog). It represents the mechanism of binding the contents of variables.
  - Variable x can be unified with an atom, a term, or another variable. In first-order logic, a variable cannot be unified with a term that contains it.
  - Two constants can only be unified if they are identical.
  - A term can be unified with another term if the top function symbols and arities of the terms are identical and if the parameters can be unified simultaneously.

Using our family knowledge base and writing more rules…

1. add the ‘son’ rule into your facts file if you have not already done so
2. add in a rule for ‘daughter’
3. load this into Prolog (this is called ‘consulting’ the file in Prologese).
4. check whether the rules work and do what you expect them to!
5. let’s then add some more…

One half of the heart of Prolog

backtracking

Backtracking

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

jealous(X,Y) :-
 loves(X,Z),loves(Y,Z).
```

Example

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

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

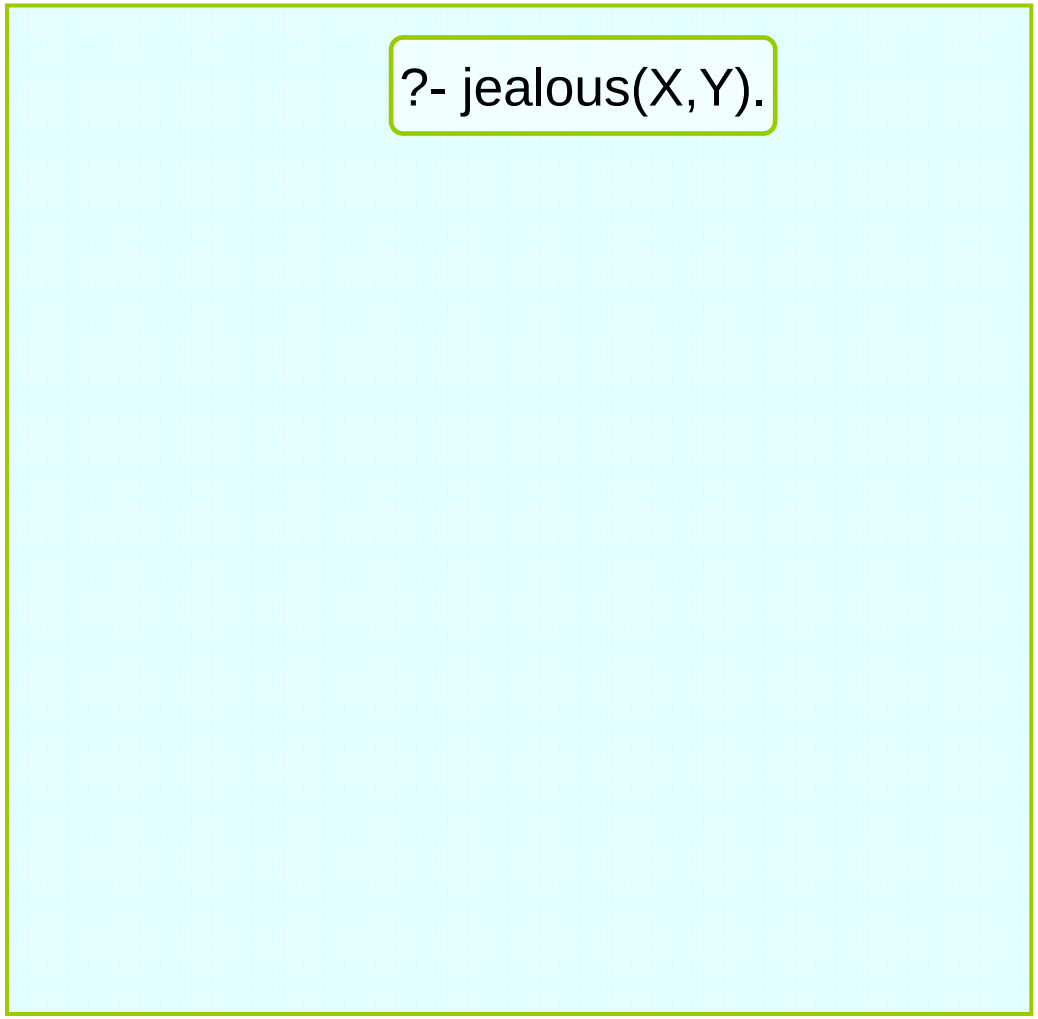
```
?- jealous(X,Y).
```

Another example

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

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

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

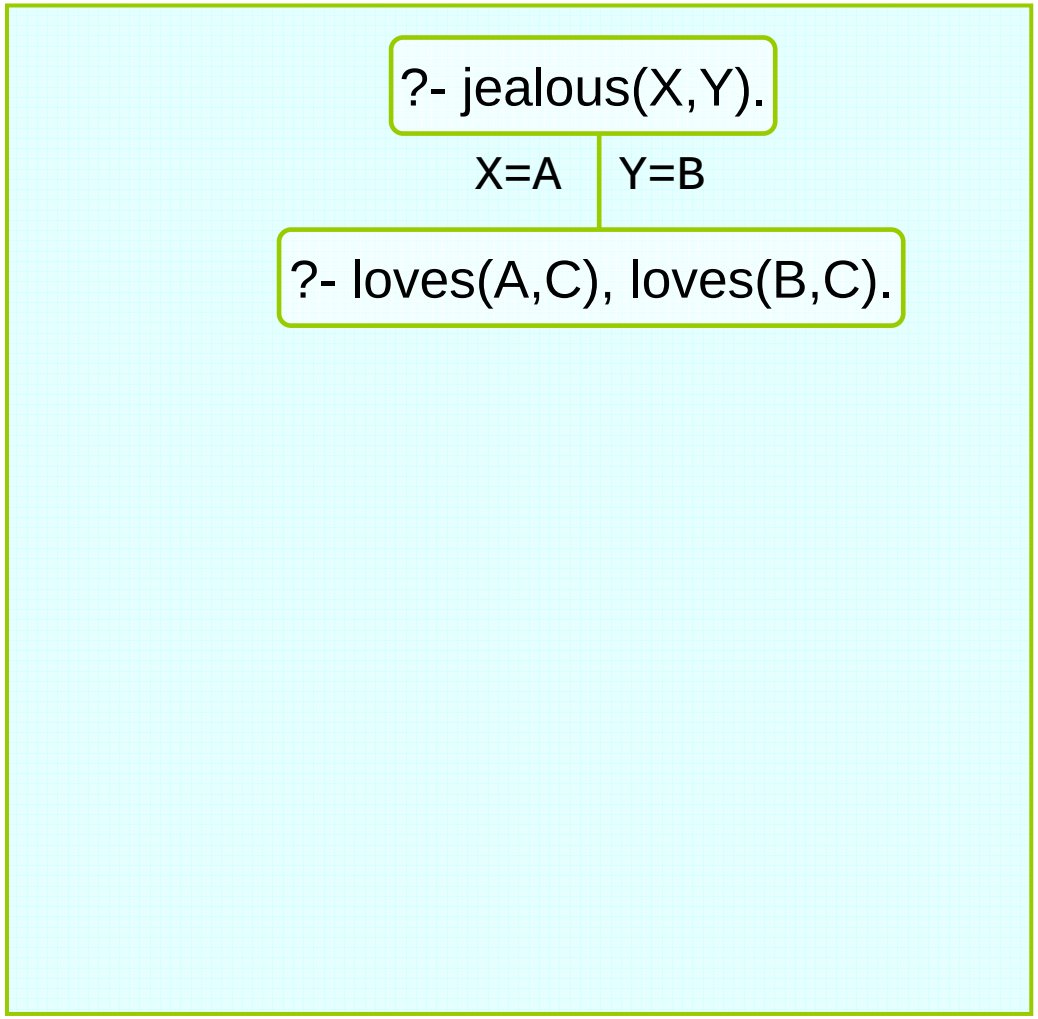


Another example

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

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

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

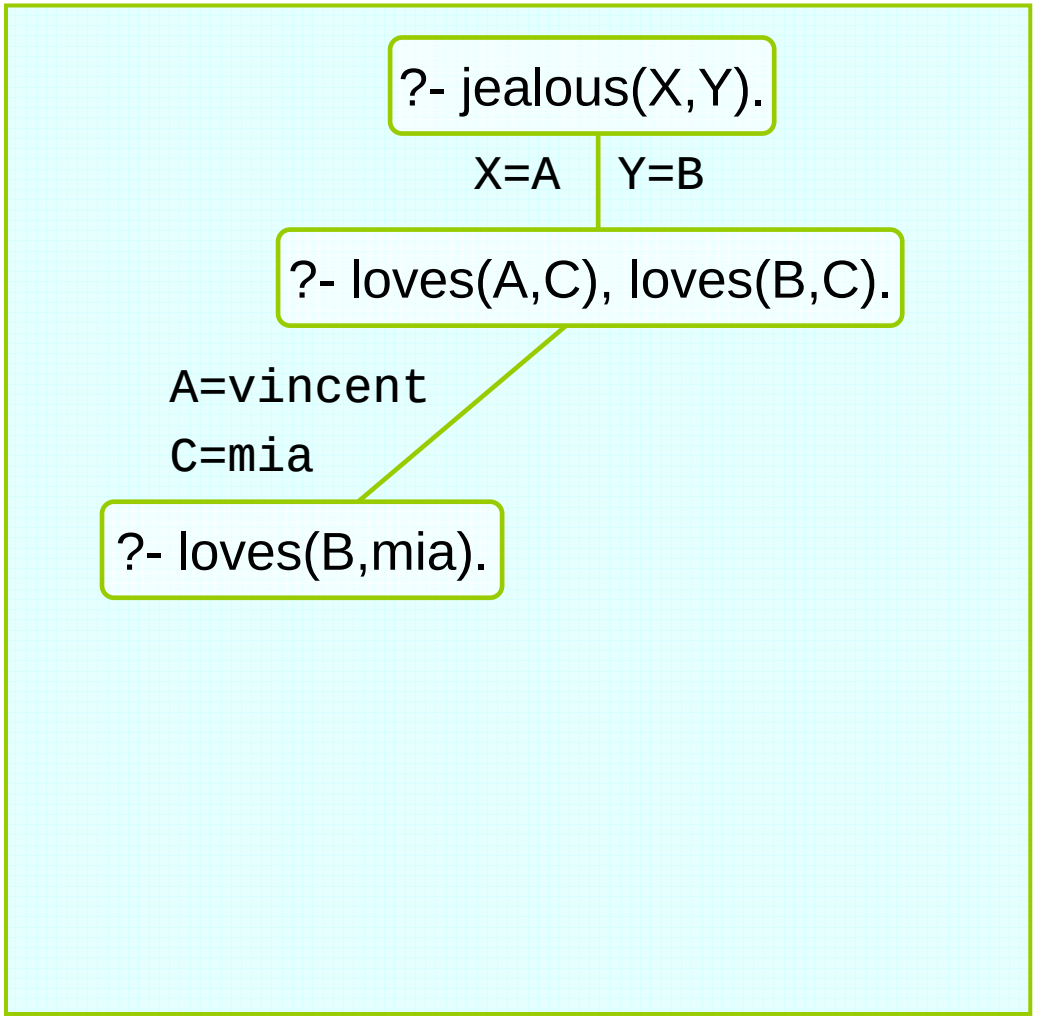


Another example

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

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

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

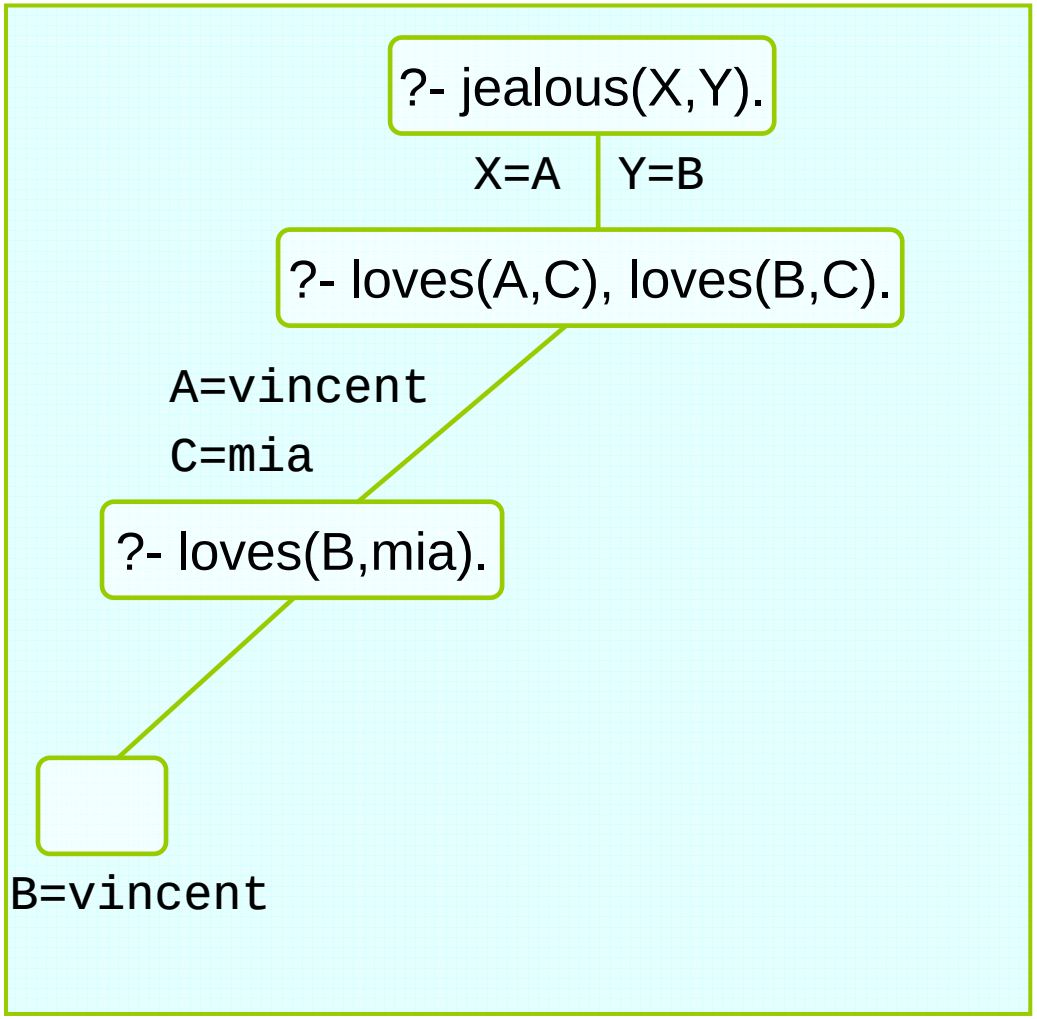


Another example

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

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

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

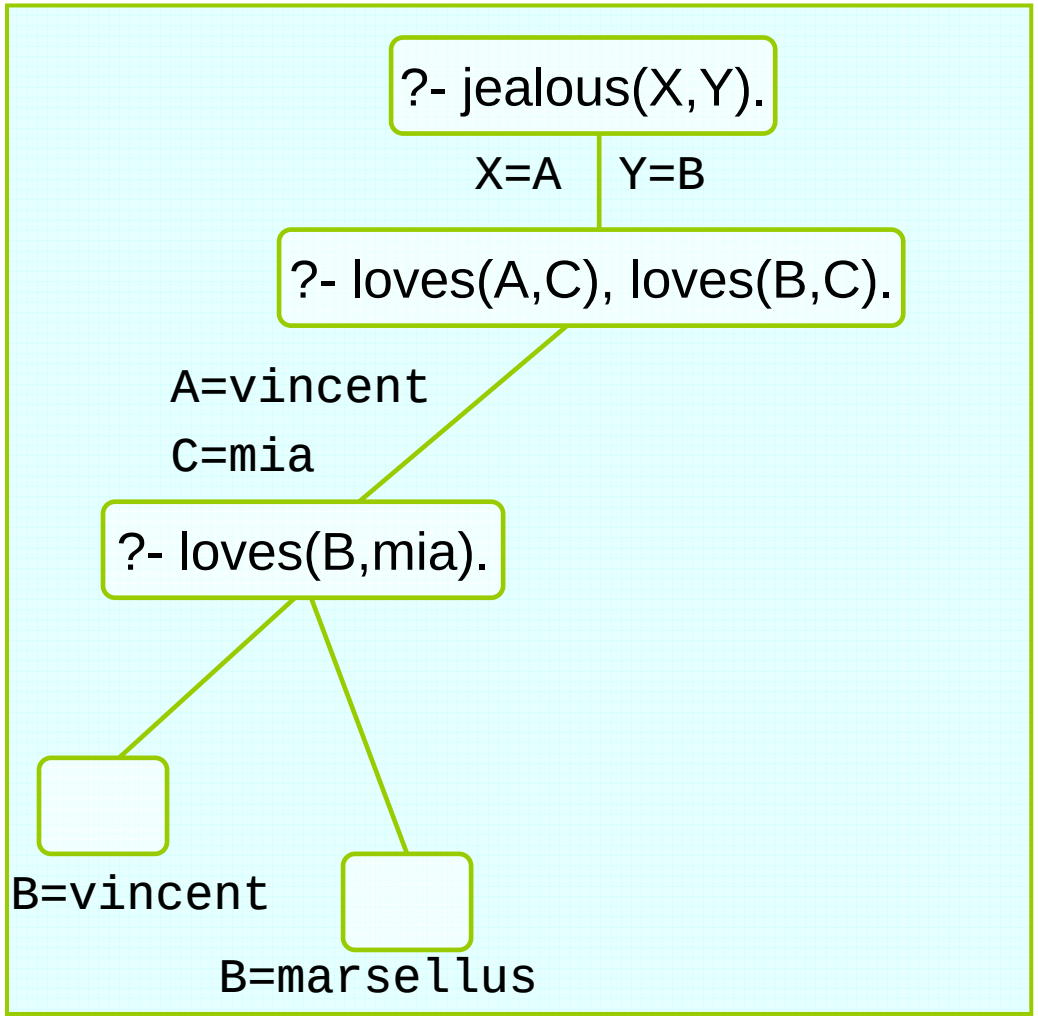


Another example

```
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
```

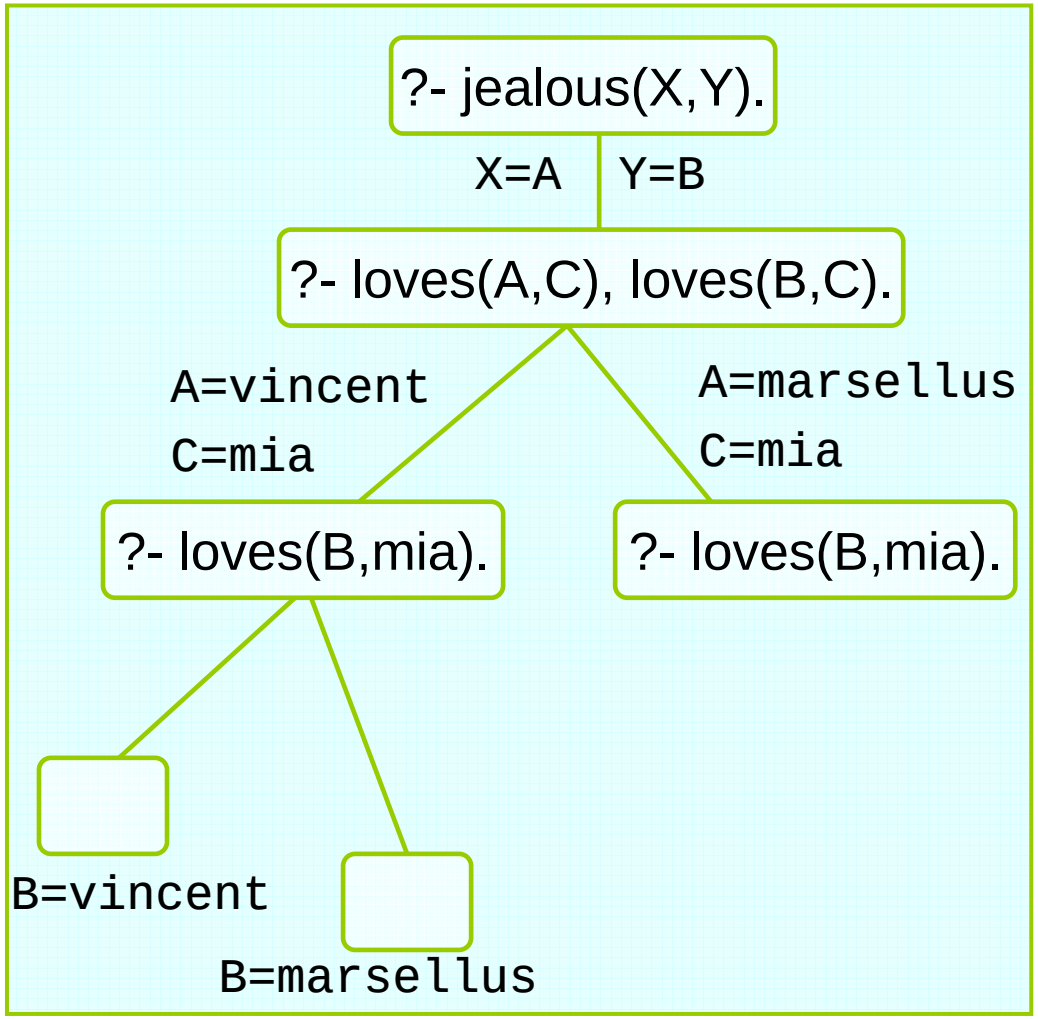


Another example

```
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;
```

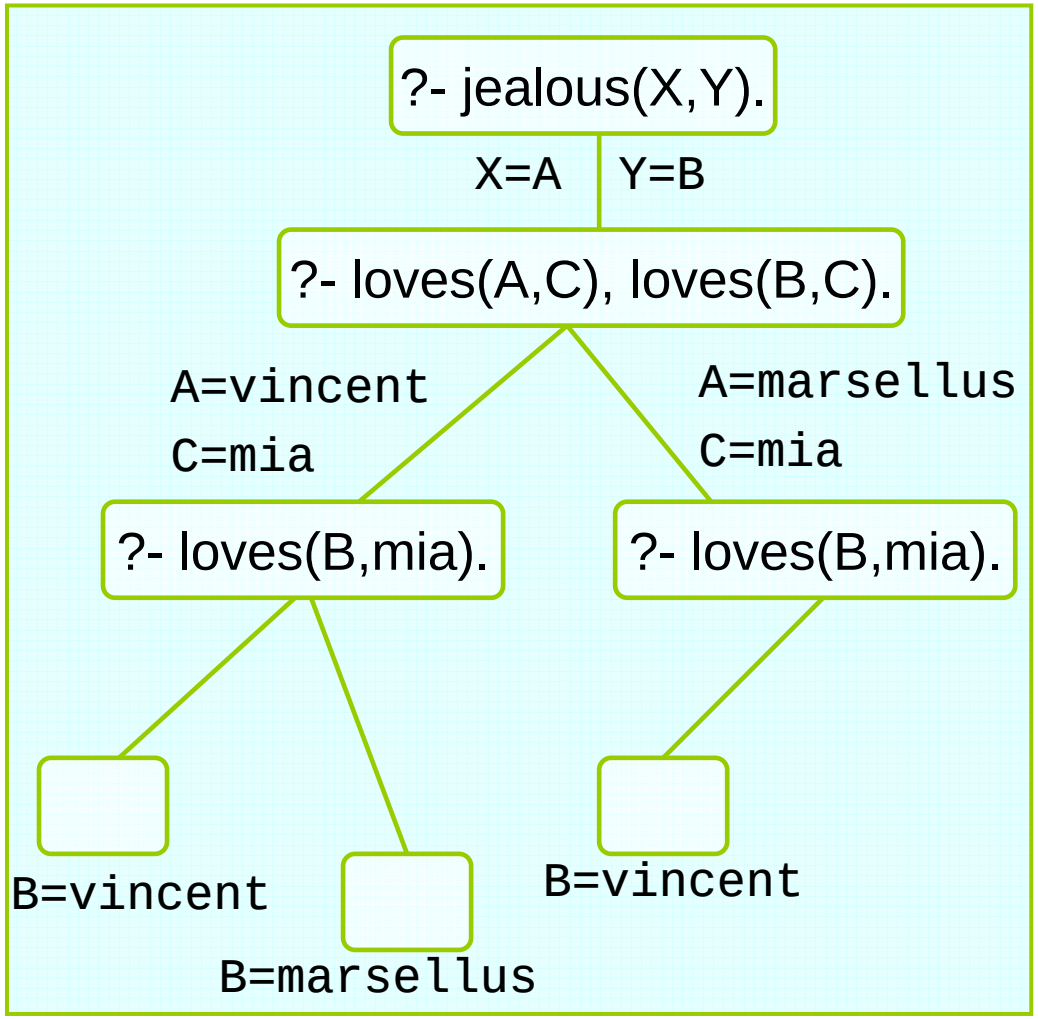


Another example

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

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

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

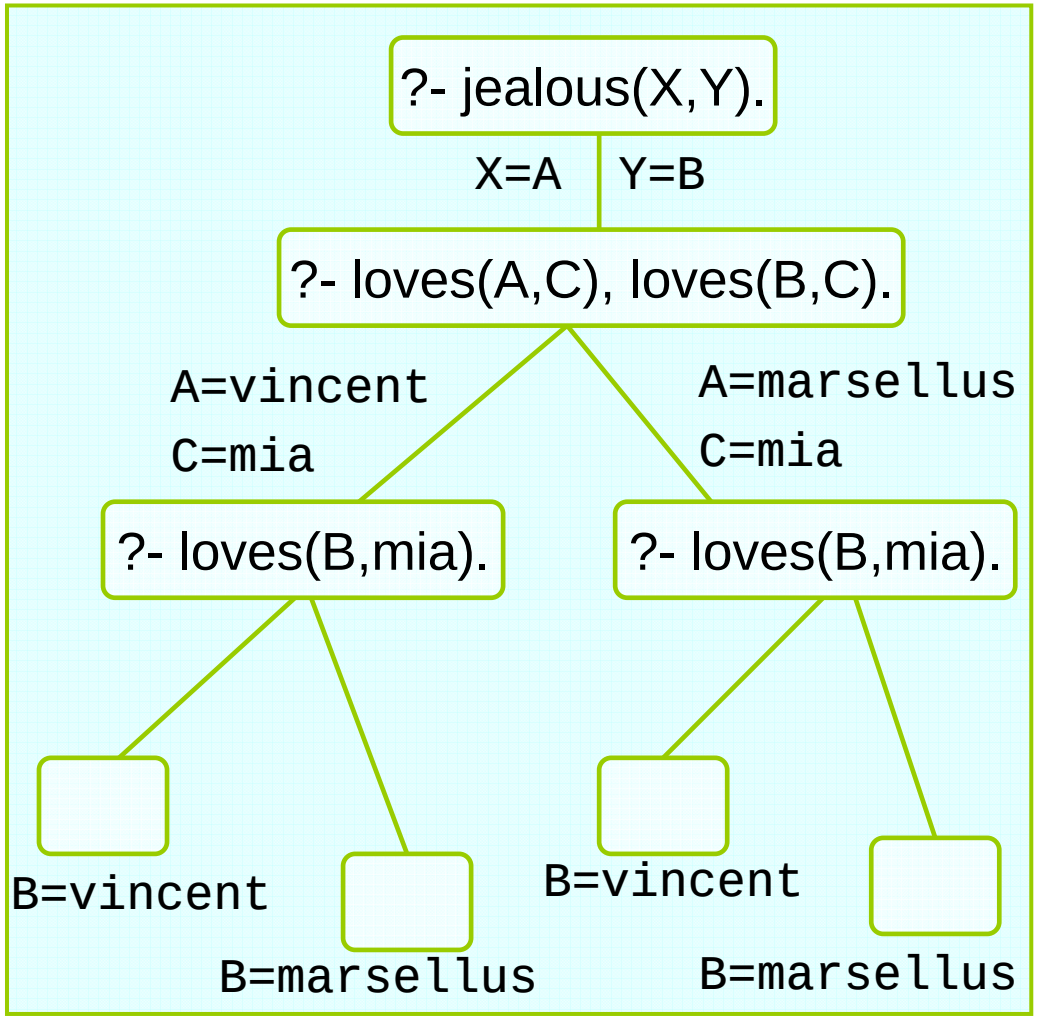


Another example

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

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

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



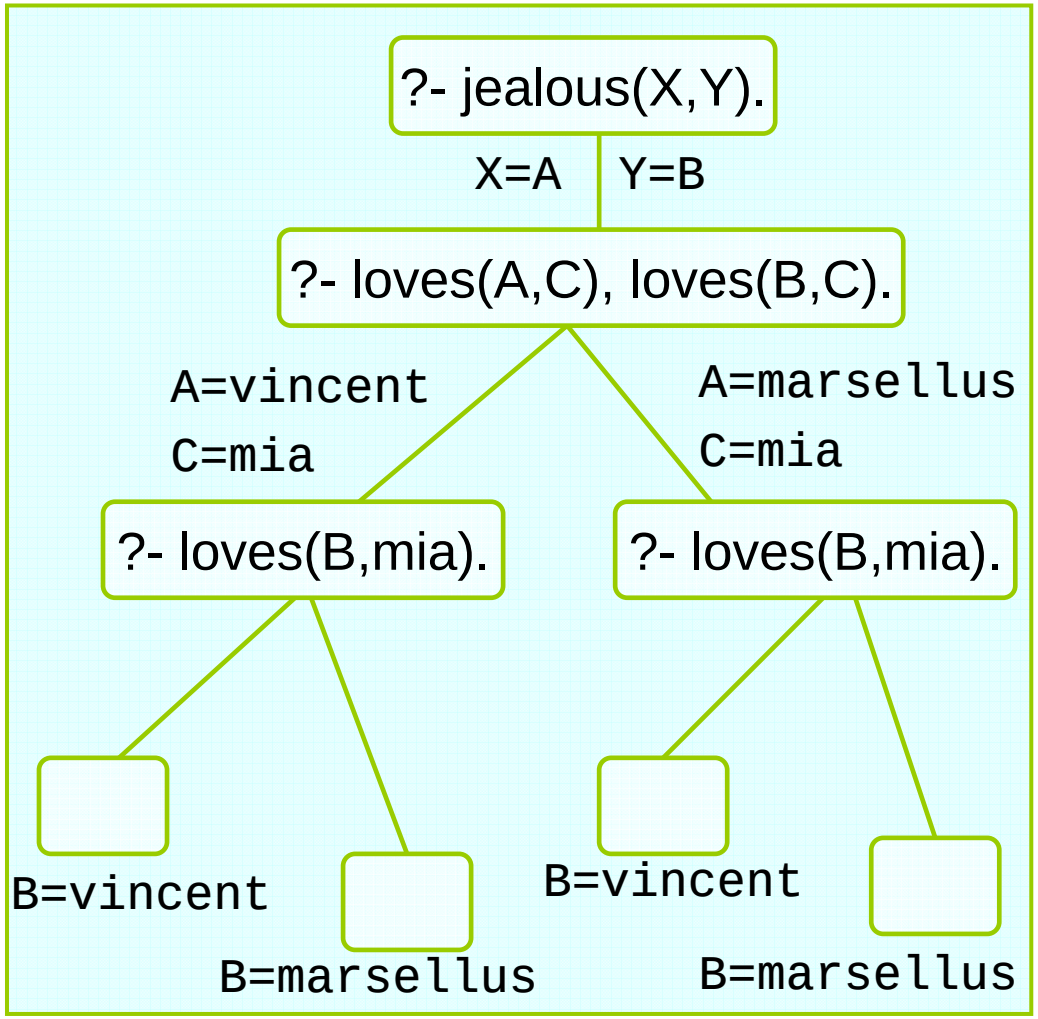
© Patrick Blackburn, Johan Bos & Kristina Striegnitz

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
```

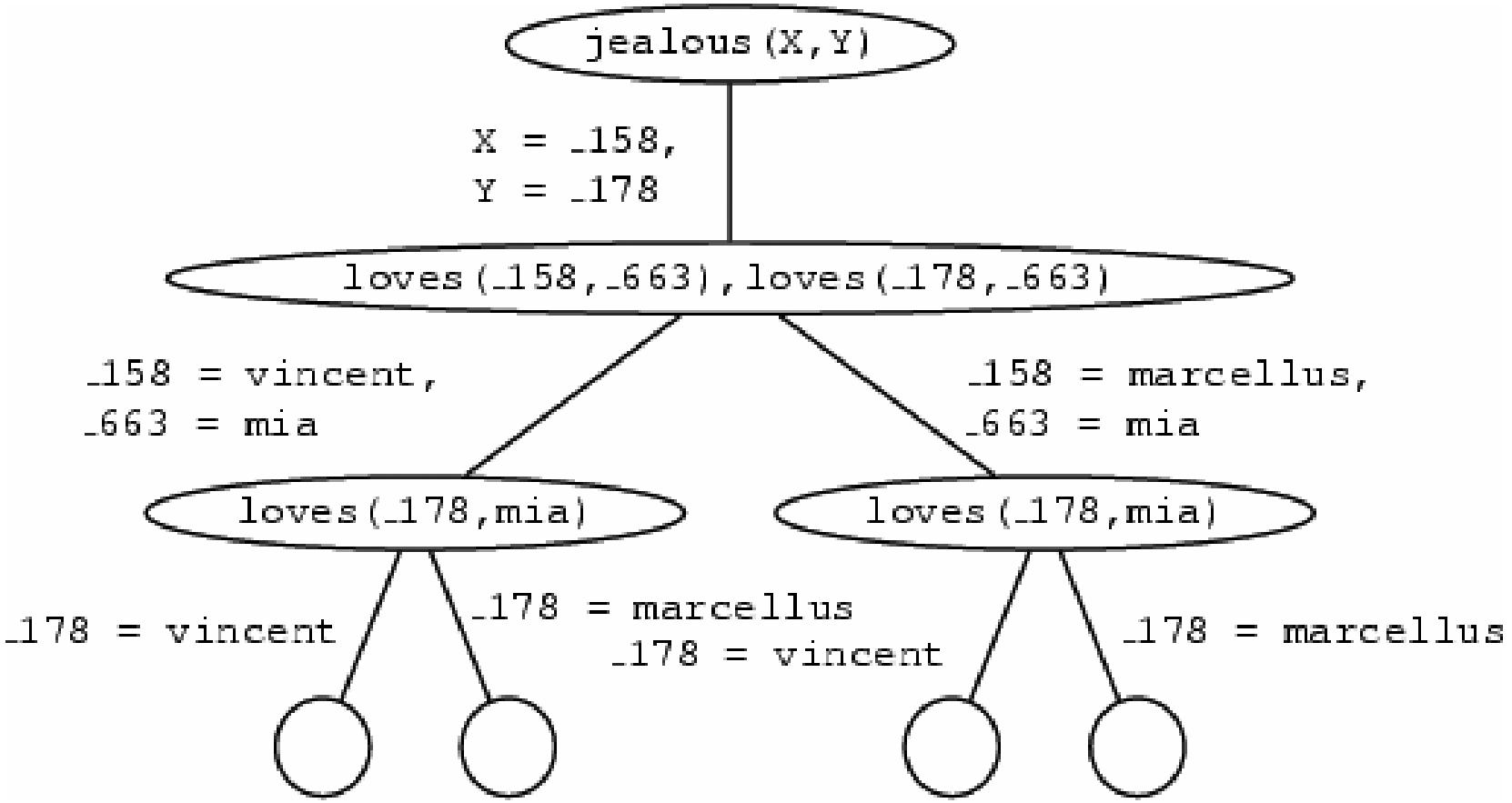


Backtracking: Search Tree

- Query: `jealous(X,Y).`

Backtracking: Search Tree

- Query: `jealous(X,Y).`



Backtracking

```
f(a).
f(b).

g(a).
g(b).

h(b).

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

Backtracking

- Query: `k(Y)`

Homework:

1. write out the full set of search trees forthis query to find out what Prolog should produce as solutions for Y.
2. check that Prolog produces the results you thought
3. see if you can following through using trace(k) the steps that Prolog actually went through: are they the same as your proof tree?