PROLOG

Overview

- Great tutorial at (these slides are a compressed version of this tutorial):
  - http://www.coli.uni-sb.de/~kris/learn-prolog-now/html/prolog-notes.pdf
- There are several Prolog interpreters available (see next reference section).
- But we will **use SWI-SH,** as there is no need for installation and it is very easy to use.
  - It is limited but sufficient to get to learn Prolog basics.

For Reference:
Prolog Interpreters

Available Prolog Interpreters

1. AMZI Prolog: http://www.amzi.com/download/prolog_download_sites.htm
   - This is one the suggested integrated development environment (IDE) which is easy to use and has a nice help document and several sample files.
   - Once you install Amzi, you should watch the FirstAmziProgram.avi. It will really help you to get started.
   - You can also see instructions in coursepage/lectures/prolog.doc about working in the Amzi environment. Important points of this document are in the next slide.

2. SWI Prolog @ www.swi-prolog.org and SWI-SH.
   - We have found SWI more difficult to install, but it is very capable.
   - SWI-SH is immediate go and use, though a bit limited in some features.

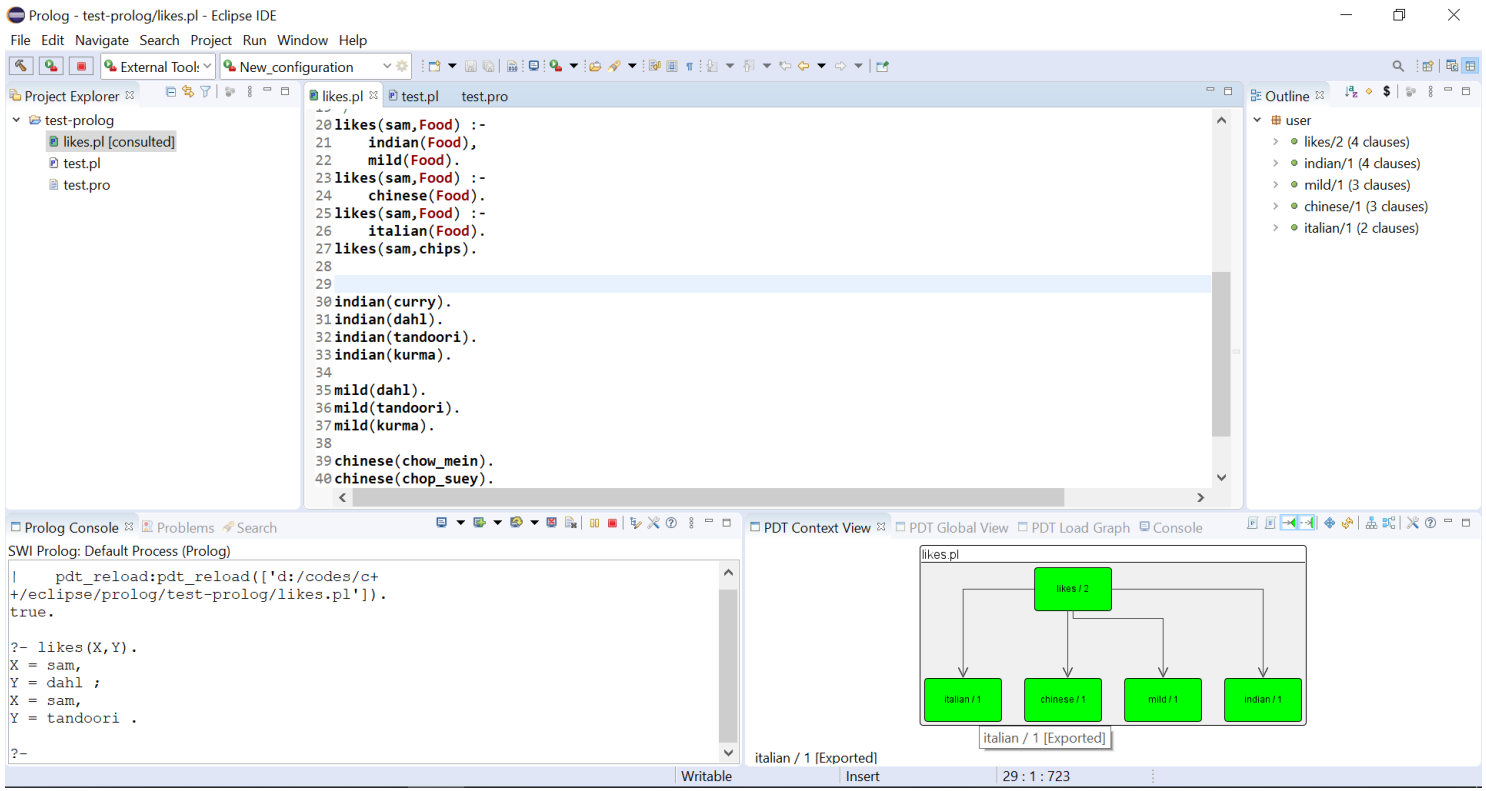
3. GNU prolog: This is a prolog interpreter working under the GNU licence.
   - Windows/UNIX support, comprehensive, free,
   - But seriously lacks in tutorial/introduction aspects. There are no samples whatsoever.
   - Still, the interface is simpler than that of Amzi's and it is quicker to start.

Getting Starting with Prolog using Amzi!

- Download, install, start, select Exit/Run Free version (option on top right)
- Select "New-Project-Prolog Project" to start a Project
- Choose "File-New-File" (file name should end with .pro) and copy and paste your gene.pro (or other prolog file) or write sample sentences into the IDE main window (top-middle part of the screen)
- Choose project name and select "Run as an Interpreted Project" (which automatically calls the listener and consults the .pro files in your project) or select a single file and choose "Run as Single Prolog File"
- Type in your queries or add new facts (e.g. "person(yasin)." or "parent(X,yasin).").
- Type "quit." in the Listener window (lower half of your screen) or hit the red button on upper-right of Listener window
- Watch the movie FirstAmziProgram.avi that demonstrate the process of starting to work with Amzi Prolog.

SWI-Prolog Development Tool (PDT)

- Prolog IDE for Eclipse provided as plug-in. All features are implemented for SWI-Prolog.
- Provides rich & user-friendly graphical user interface (GUI) on Windows, Linux & Mac OS.



SWI-Prolog Development Tool (PDT)

- To download and install, please visit:

  https://sewiki.iai.uni-bonn.de/research/pdt/docs/download

- Please ensure all those prerequisites are fulfilled, as described in the installation page.
  - May not be too simple but should be handle-able.

- Potential errors during/after installations:
  - On Windows, Eclipse cannot open swi-prolog console under 'Prolog' perspective:
    - Please make sure your Windows defender/firewall allows communication between Eclipse and SWI-Prolog, grant permission if necessary.
  - On Mac:
    - Eclipse cannot start due to 'Java Virtual Environment' failure:
      - Please make sure you are using the latest JDK version.
      - If using the latest Eclipse and JDK, please uninstall your current JDK, and install openjdk-8 (no later version).
      - If using later version of openjdk, please downgrade to openjdk-8 version.
    - Eclipse cannot open swi-prolog console:
      - Double-confirm your PATH includes your swipl executable location.
      - Make sure you have configured your swi-prolog executable correctly for your project (Eclipse-Preferences-PDT-Prolog Processes-Edit).
      - Make sure you have installed XQuartz AND running.

SWI-Prolog Development Tool (PDT)

- On Mac, if the ‘Apply’ button is grayed out during Prolog process configuration:
  - Make sure the package link you provided via ‘Help-Install New Software’ during PDT plug-in installation points to the latest Github repository (there are totally three links on the installation page), and re-install again.
  - If using Eclipse C/C++ Developer IDE, try to move to Eclipse IDE for Java.

- For other potential errors, please refer to the “Troubleshooting” section on PDT installation link above, or the FAQ page:

  https://sewiki.iai.uni-bonn.de/research/pdt/docs/faq

Prolog Basics

Prolog

- A logic programming language created in 1972
- PROgramming in LOGic
- Restricted to Horn clauses
  - Head:- body
- Inference
  - Backward chaining
- **Closed world assumption:** Facts not known to be true are assumed False.

Knowledge Base - facts

- Knowledgebase can have facts:
  - `woman(mia).`
  - `playsguitar(jane).`
  - …

Knowledge Base - facts

- Knowledgebase can have facts:
  - `woman(mia).`
  - `playsguitar(jane).`

- Consulting the KB is done in the Interpreter window:
  - Prolog listens to your queries and answers:
    - `?- woman(mia).` `//asking if mia is a woman`
    - `yes`

Consulting

- Consulting the KB:
  - Prolog listens to your queries and answers:
    - `?- woman(mia)`
    - `yes`

    - `?- woman(jane)`
    - `no`
      - doesn't follow from KB

    - `?- woman(alisa)`
    - `no`
      - doesn't know anything about alisa

Knowledge Base - rules

```
male(yasin).
female(aliye).
male(yusuf).

mortal(X) :- person(X).
person(X) :- female(X).
person(X) :- male(X).
```

head := body means body => head

e.g. person (X) => mortal (X)

KnowledgeBase - rules

- `male(yasin).`
- `female(aliye).`
- `male(yusuf).`
- `mortal(X) :- person(X).`
- `person(X) :- female(X).`
- `person(X) :- male(X).`

  - We can test the program inside the Listener with prolog queries:
    - `?- mortal(araba).`
    - `no`
    - `?- mortal(yasin).`
    - `yes`

Rules - Logical AND

- `,` is used to indicate Logical AND

- **`dances(vincent) :- happy(vincent) , listensToMusic(vincent).`**

is equivalent to the FOL statement of:

- **`happy(vincent)` $\wedge$ `listensToMusic(vincent)=>dances(vincent)`**
- "Vincent dances if he listens to music and he is happy".

- Other example:
- **`father(X,Y) :- parent(X,Y) , male(X).`**

Rules - Logical OR

- Indicates LOGICAL OR

- **dances(john) :- happy(john).**
- **dances(john) :- listensToMusic(john).**

  - Equivalent to:
    - happy(john) ∨ listensToMusic(john) => dances(john)
    - "John dances *either if* he listens to music, *or* if he is happy."

- This can also be stated as:
  - **dances(john) :- happy(john) ;**
    listensToMusic(john).

  - **where ; indicates OR.**

Consulting

File:

- `woman(mia).`
- `woman(jody).`
- `woman(yolanda).`
- `loves(vincent,mia).`
- `loves(marcellus,mia).`

In the interpreter window (?):

- `?- woman(X).`
- `X = mia`

Consulting

- `woman(mia).`
- `woman(jody).`
- `woman(yolanda).`
- `loves(vincent,mia).`
- `loves(marcellus,mia).`

- `?- woman(X).`
- `X = mia`
- `?- ;` (remember that ; means *OR* so this query means: "*are there any more women*?")
- `X = jody`
- `?- ;`
- `X = yolanda`
- `?- ;`
- `no` (No other match is possible)

Inference

- `woman(mia).`
- `woman(jody).`
- `woman(yolanda).`
- `loves(vincent,mia).`
- `loves(marcellus,mia).`

- `?- loves(marcellus,X),woman(X).`
- `...`
- Note: we are querying for a conjunct.

```
loves(vincent,mia).
loves(marcellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
jealous(X,Y) :- loves(X,Z),loves(Y,Z).
```

- Any jealous people?
- ?- jealous(marcellus,W).
  - apply Generalized Modus Ponens
  - ...
- Any other?

Wildcard

- In Prolog predicates, underscore (_) is the wildcard (matches anything):
  - `Mother(M,C) :- Person(C,_,M,_,_).`

  where Person predicate is defined as

  `Person(name, gender, mother, father, spouse).`

  It means, Mother(M,C) holds, if the predicate Person holds for C and M in the right positions, with anything else for the other parts.

Backward Chaining in Prolog

Proof Search – How does Prolog search?

- Suppose we are working with the following knowledge base
  - `f(a).`
  - `f(b).`
  - `g(a).`
  - `g(b).`
  - `h(b).`

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

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

- g(a).
- g(b).

- h(b).

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

- Pose the query `k(X).`
- You will probably see that there is only one answer to this query, namely `k(b)`,but how exactly does Prolog work this out?

Backward chaining algorithm

function FOL-BC-ASK($KB, goals, \theta$) **returns** a set of substitutions

inputs: KB, a knowledge base

$goals$, a list of conjuncts forming a query

θ, the current substitution, initially the empty substitution $\{\}$

local variables: $answers$, a set of substitutions, initially empty

if $goals$ is empty **then return** $\{\theta\}$

$q' \leftarrow \text{SUBST}(\theta, \text{FIRST}(goals))$

for each sentence r **in** KB

where $\text{STANDARDIZE-APART}(r) = (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$

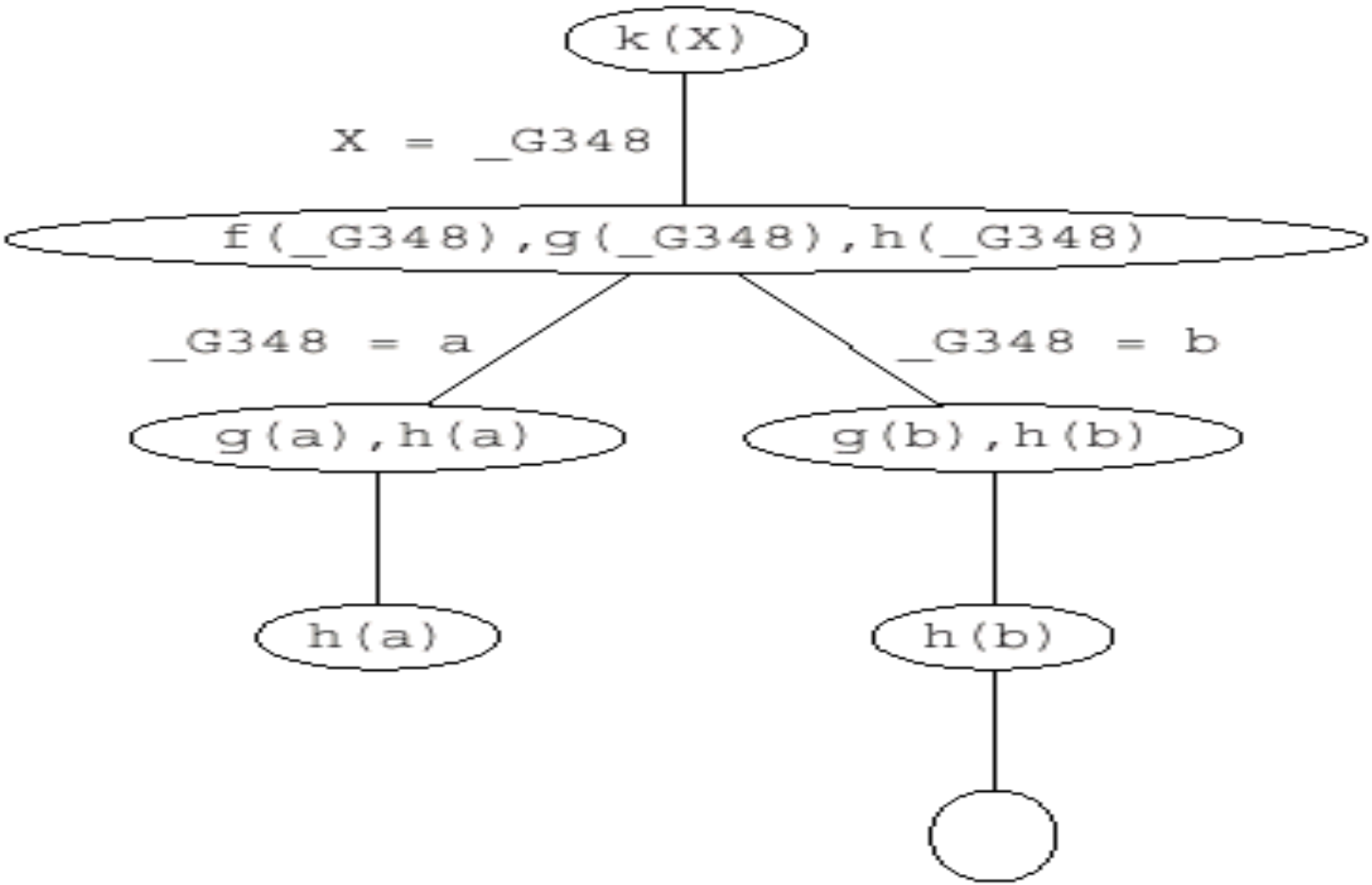
and $\theta' \leftarrow \text{UNIFY}(q, q')$ succeeds

$new\_goals \leftarrow [p_1, \ldots, p_n | \text{REST}(goals)]$

$answers \leftarrow \text{FOL-BC-ASK}(KB, new\_goals, \text{COMPOSE}(\theta', \theta)) \cup answers$

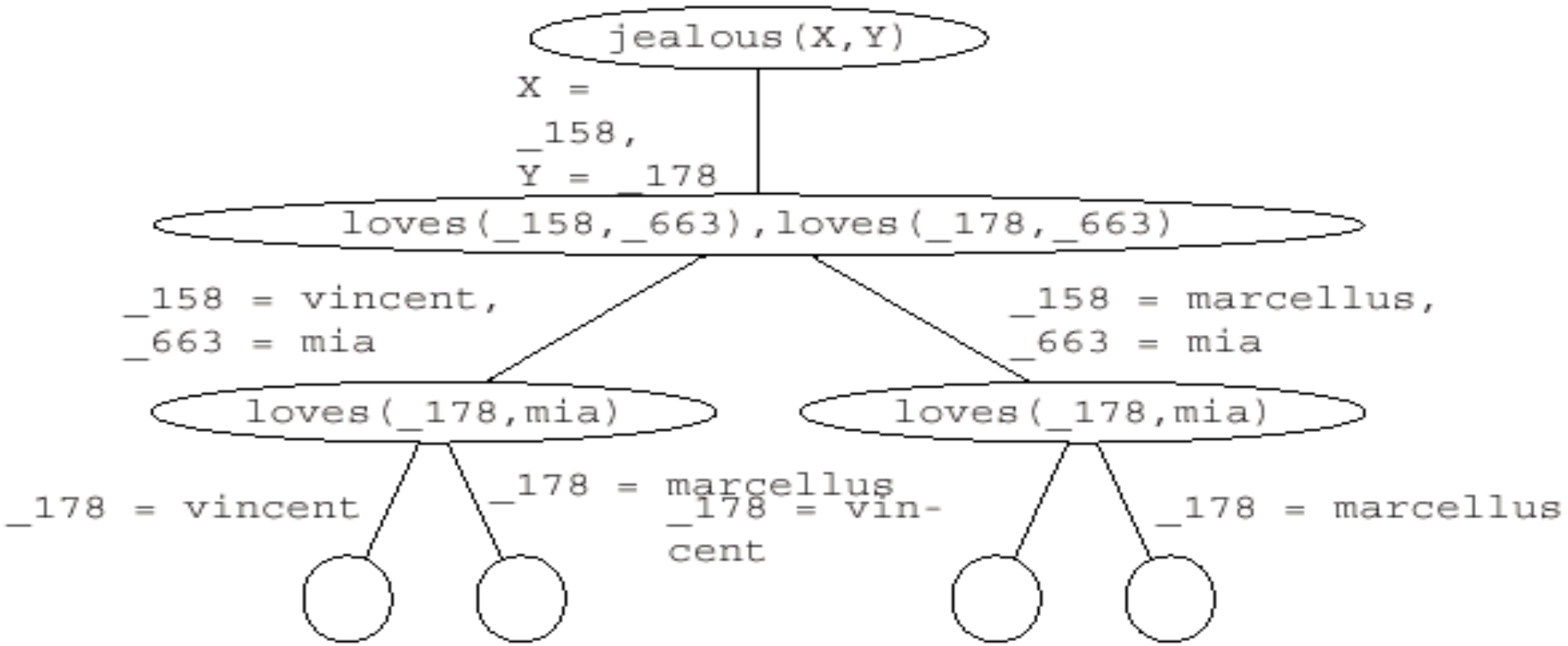
return $answers$

Backtracking search



Backtracking search

The search tree for this query looks like this:



There is only one possibility of matching `jealous(X,Y)` against the knowledge base. That is by using the rule

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

- 4 leave nodes with an empty goal list
  - four ways for satisfying the query.
  - the variable instantiation for each of them can be read off the path from the root to the leaf node.

- 1. `X = \_158 = vincent` and `Y = \_178 = vincent`
- 2. `X = \_158 = vincent` and `Y = \_178 = marcellus`
- 3. `X = \_158 = marcellus` and `Y = \_178 = vincent`
- 4. `X = \_158 = marcellus` and `Y = \_178 = marcellus`

- `We need to use X \= Y to eliminate 1 and 4.`

Consulting

- `woman(mia).`
- `woman(jody).`
- `woman(yolanda).`

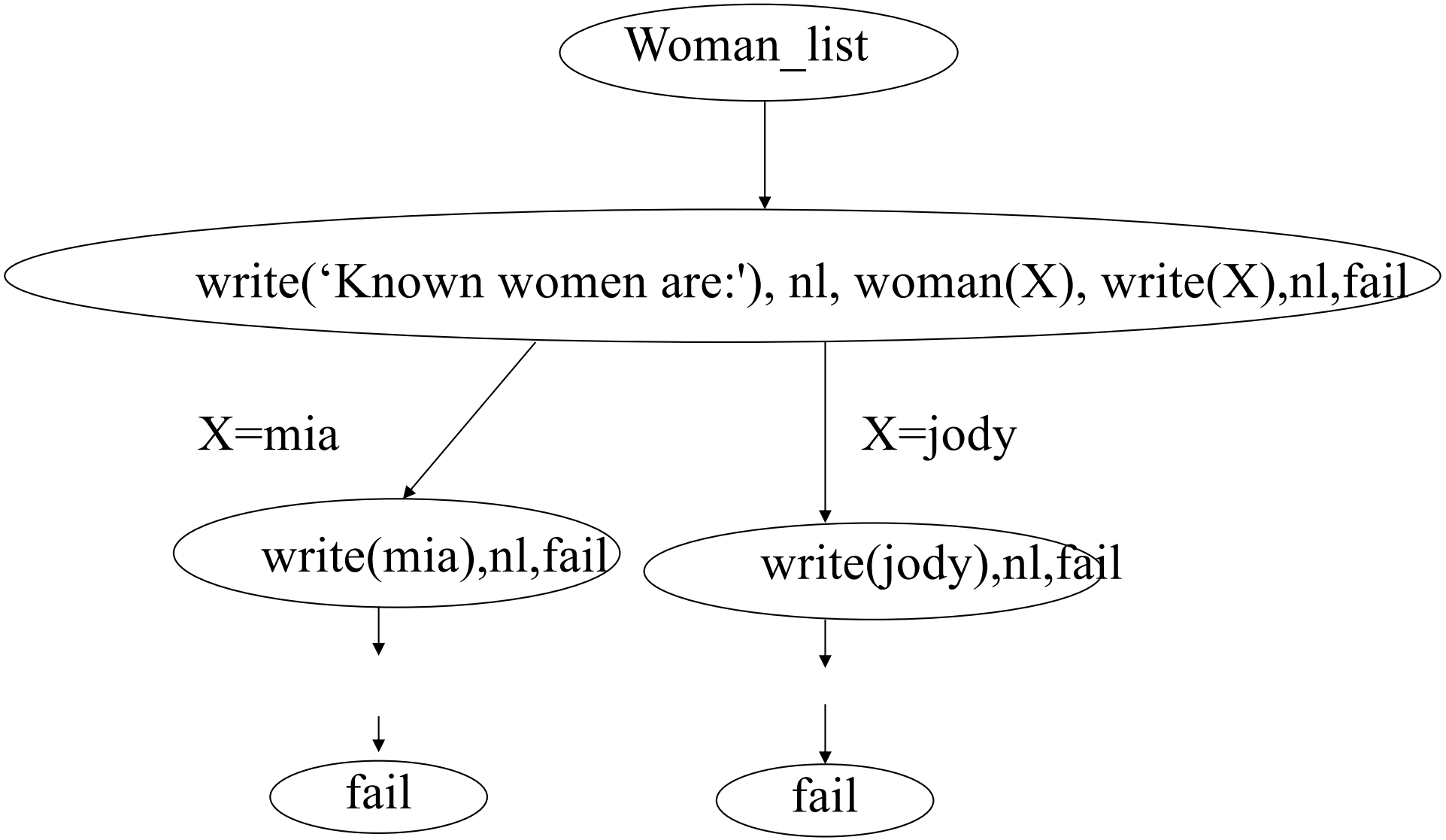
- `?- woman(X).`
- `X = mia`
- `?- ;`
- `X = jody`
- `... Any better way?`

fail predicate

- `woman(mia).`
- `woman(jody).`
- `woman(yolanda).`
- `loves(vincent,mia).`
- `loves(marcellus,mia).`

```
woman_list:-
  write('Known women are:'),nl,
  woman(X),
  write(X),nl,
  fail.
```

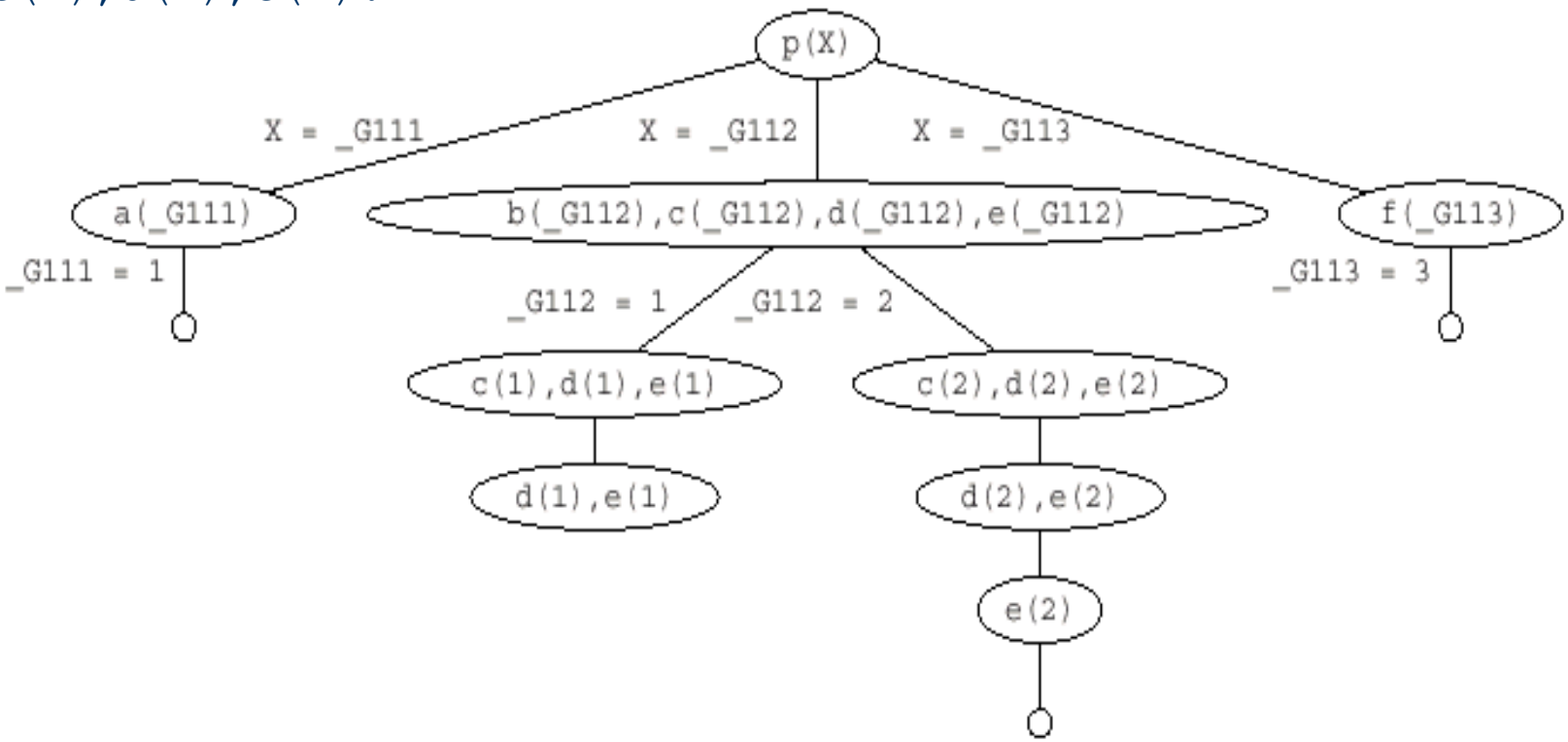
The first match (mia) is written and then the rule fails, forcing Prolog to backtrack and try different matches (jody, yolanda,...)


Woman_list
write(‘Known women are:'), nl, woman(X), write(X),nl,fail
X=mia
X=jody
write(mia),nl,fail
write(jody),nl,fail
fail
fail

Negation and Cut

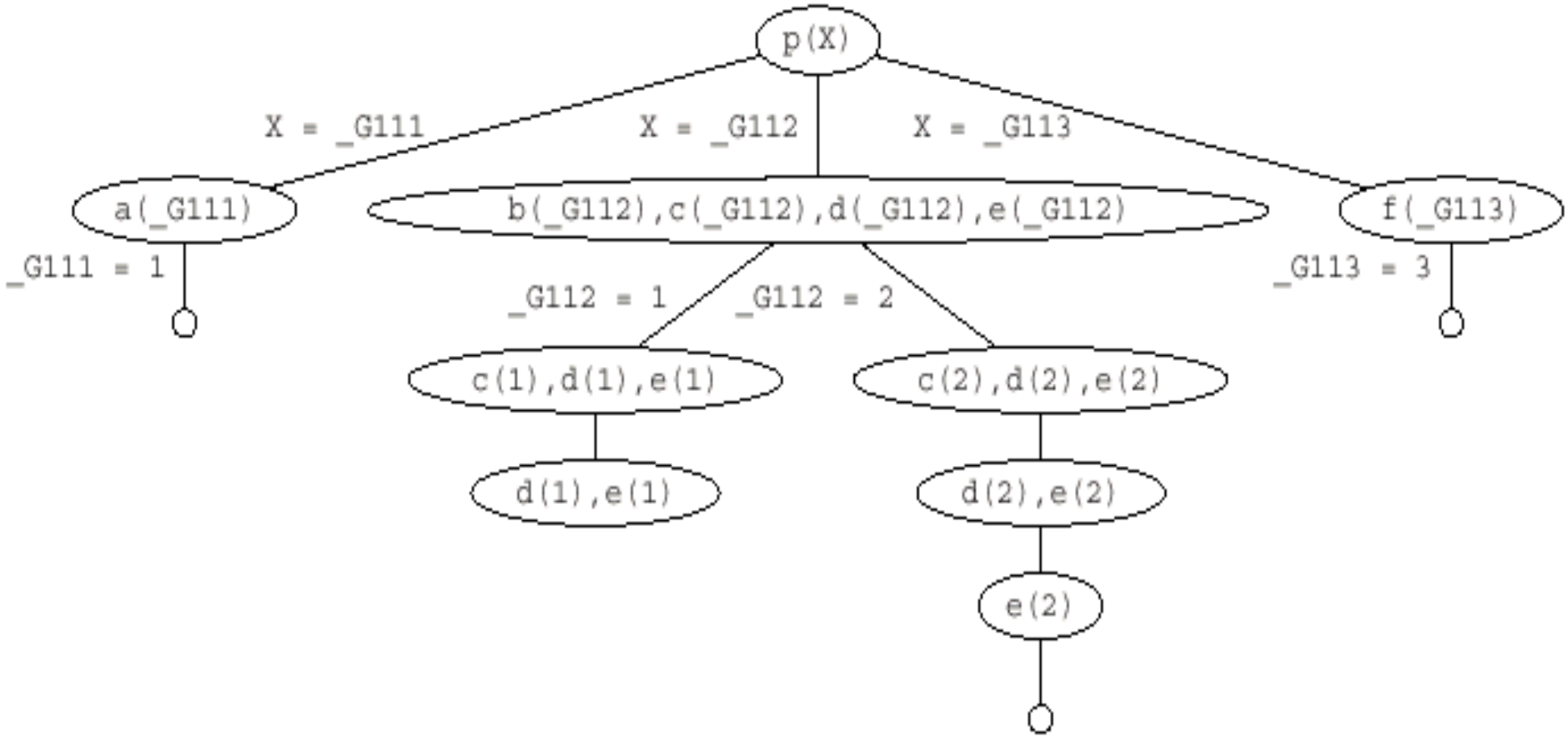
Negation and Cut

```
p(X) :- a(X).
p(X) :- b(X),c(X),d(X),e(X).
p(X) :- f(X).

a(1).
b(1).
c(1).

b(2).
c(2).
d(2).
e(2).

f(3).
```



If we pose the query p(X) we will get the following responses:

```
X =1;
X =2;
X =3;
no
```


p(X)
X = _G111
X = _G112
X = _G113
a(_G111)
b(_G112),c(_G112),d(_G112),e(_G112)
f(_G113)
_G111 = 1
_G113 = 3
_G112 = 1
_G112 = 2
c(1),d(1),e(1)
c(2),d(2),e(2)
d(1),e(1)
d(2),e(2)
e(2)

Cut operator

- But now supppose we insert a cut in the second clause:

```
p(X) :- a(X).
p(X) :- b(X),c(X),!,d(X),e(X).
p(X) :- f(X).
```

- If we now pose the query `p(X)` we will get the following responses:

```
X =1;
no
```

Cuts

The ! goal always succeeds and commits us to all the choices we have made so far.

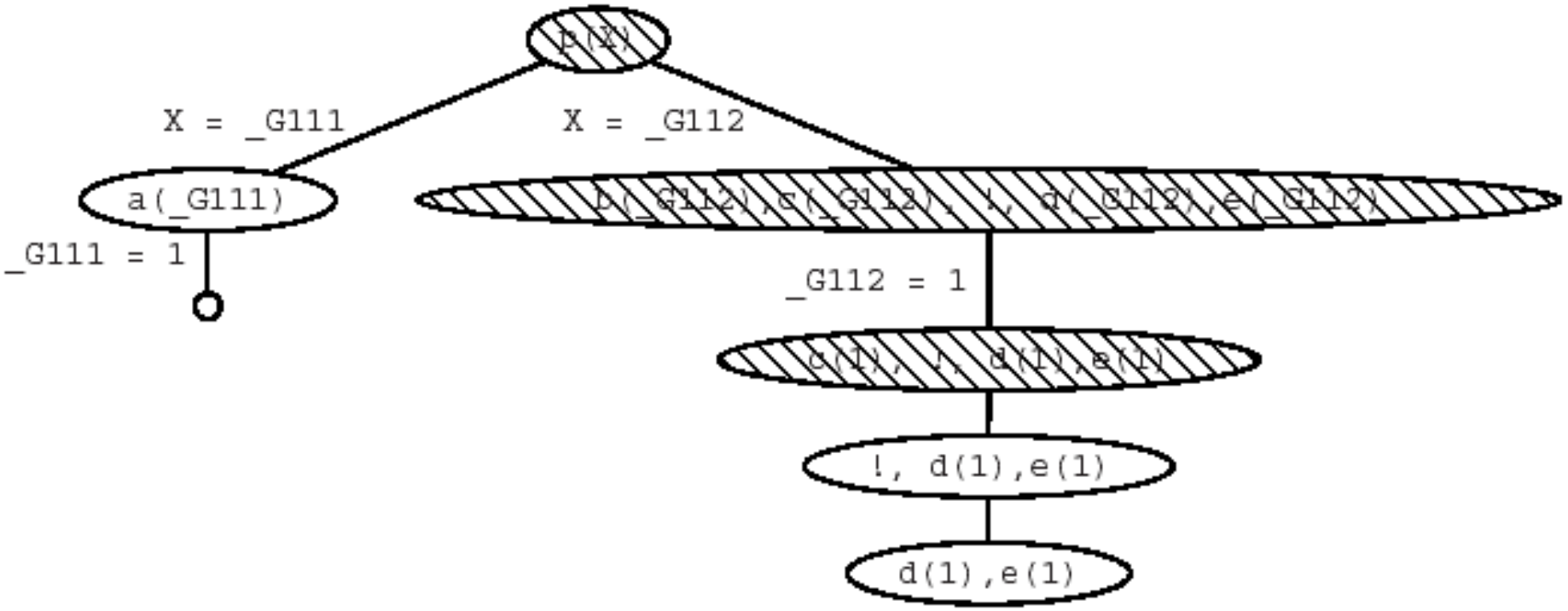
- All nodes above the cut, **up to the one containing the goal that led to the selection of the clause containing the cut** (second p clause in this case) are blocked.

- If we were allowed to try the third rule, we could also generate the solution X=3. But we can't do this: the cut has committed us to using the second rule.

```
p(X) :- a(X).
p(X) :- b(X),c(X),!,d(X),e(X).
p(X) :- f(X).
```

```
p(X)  :- a(X).
p(X)  :- b(X),c(X),!,d(X),e(X).
```

Looking at the search tree this means that search stops when the goal `d(1)` cannot be shown as going up the tree doesn't lead us to any node where an alternative choice is available. The red nodes in the tree are all blocked for backtracking because of the cut.



Cuts

- For example, in a rule of the form:
  - q :- p1,...,pn,!,r1,...,rm

Once we reach the the cut, it commits us to using this particular clause for q and it commits us to the choices made when evaluating p1,...,pn (remember as: everything to the left of the cut is fixed).

However, we are free to backtrack among the r1,...,rm and we are also free to backtrack among alternatives for choices that were made before reaching the goal q.

Use of Cuts

```
dup_check(Name) :-
  person(Name),
  assert(message($Person is already in database$)),
  !, fail.
dup_check(_).
```

Exceptions

We can also use cuts to implement exceptions:

How can we say something like “Ali likes all animals, except snakes” in Prolog?

```
likes(Ali,X) :- animal(X), not(snake(X)).
```

Negation in Prolog is defined as:

- `not(Goal) :- Goal,!,fail.`
- `not(Goal).`

- Prolog has a lot of intricacies which are beyond the scope of this course
- Homework 4 will be be given this weekend so that you can exercise what you have learned on FOL and Prolog.

Exceptions

How can we say something like “Ali likes all animals, except snakes” in Prolog?

This can also be written as:

```
Likes(Ali, X) :- animal(X), snake(X), !, fail.    % Cut and fail if X is a snake.
Likes(Ali, X) :- animal(X).                       % Otherwise, Ali likes the animal.
```

Negation as Failure

Negation in Prolog is implemented based on the use of cut.

Actually, negation in Prolog is the so-called *negation as failure:*

- to prove **not(p)**, one tries to prove **p**
- if **p is proved**, then its negation, **not (p), fails.**
- if **p fails** during execution, then **not(p) will succeed**.

```
dislikes(ali,X) :- not (likes(ali,X)).
```

\+ is more standard than “not”:

```
dislikes(ali,X) :- \+ likes(ali,X).
```

Very detailed(read only if interested):
https://cliplab.org/~vocal/public_info/seminar_notes/node52.html