PROLOG

Prolog

- A logic programming language created in 1972
- PROgramming in LOGic
- Restricted to Horn clauses
  - Head:- body
- Inference
  - Backward chaining
- Closed world assumption

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

KnowledgeBase - 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).
```

You can type

?- debug.

and enter a debug mode where you can see what rule is used in the matching.

- If you save these in a file: mortal.pro (a prolog program), you can TELL these to the interpreter via (or when you click on file name and select run-as single interpreted file, when listener window is closed):
  - `?- consult(mortal).`
  - `Yes`
- If you type "listing", Prolog will list all the facts and rules you just "read in" (consulted).
  - `?- listing.`
  - `male(yasin)`
  - `...`

KnowledgeBase - rules

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

- If you save these in a file: mortal.pro (a prolog program), you can TELL these to the interpreter via:
  - `?- consult(mortal).`
  - `Yes`
- ...
- Now we can test the program inside the Listener with prolog queries:
  - `?- mortal(araba).`
  - `no`
  - `?- mortal(yasin).`
  - `yes`

Rules - Logical AND

- `dances(vincent) :- happy(vincent) , listensToMusic(vincent).`
  - , is used to indicate Logical AND
  - Equivalent to:
    - happy(vincent) ∧ 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

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

- Indicates LOGICAL OR
- 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 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.

assert, retract, tell, told...

- You can dynamically add new facts and rules into the interpreter (not into the Prolog file) by assert:
  - `?- assert(person(ali,male,ayse,veli,fatma)).`
- You can dynamically retract facts within the interpreter (not into the Prolog file) by retract:
  - `?- retract(person(ali,male,ayse,veli,fatma)).`
- You can write into a file by tell:
  - `?- tell('out.txt').` //now anything you write goes to that file
- You can now close that file by told:
  - `?- told.` //stops the output

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).

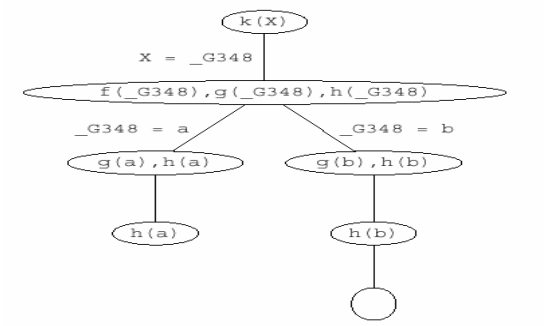
  - g(a).
  - g(b).

  - h(b).

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

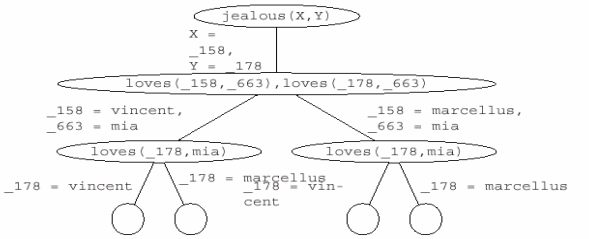
- Pose the query k(X).
- You will probably see that there is only one answer to this query, namely k(b),but how exactly does Prolog work this out?

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

- So who is jealous? How to fix it?

Consulting

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

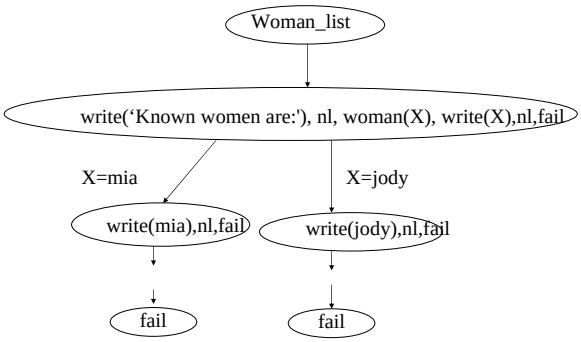
- ?- woman(X).
- X = mia
- ?- ;
- X = jody
- ... Any better way?

Consulting

- 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,...)



Negation and Cut

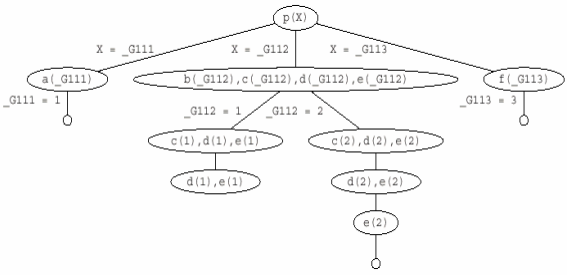
```
p(X) :- a(X).
p(X) :- b(X),c(X),d(X),e(X).
p(X) :- f(X).

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



Cuts

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

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

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

  X =1;
  no

Cuts

- The ! goal succeeds (it always does) 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** (p 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.

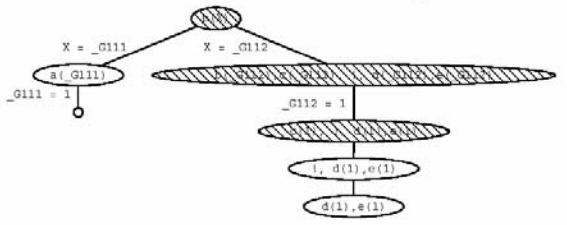
Cuts

- 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.

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.



Why are cuts useful

- Imagine a max function that returns the max of two numbers:, defined as:

```
max(X,Y,Y) :- X =< Y.
max(X,Y,X) :- X>Y.
```

- If max(3,4,Y) is queried, the program will correctly set Y=4.
- But now consider what happens if at some stage backtracking is forced. The program will try to resatisfy max(3,4,Y) using the second clause. Of course, this is completely pointless: the maximum of 3 and 4 is 4 and that's that. There is no second solution to find. To put it another way: the two clauses in the above program are mutually exclusive: if the first succeeds, the second must fail and vice versa. So attempting to resatisfy this clause is a complete waste of time.
- Solution:

```
max(X,Y,Y) :- X =< Y,!.
max(X,Y,X) :- X>Y.
```

Exceptions by Cut-Fail

- We want to say that vincent likes all burgers except for big_kahuna_burgers. Lets try:

```
enjoys(vincent,X) :- big_kahuna_burger(X),!,fail.
enjoys(vincent,X) :- burger(X).

burger(X) :- big_mac(X).
burger(X) :- big_kahuna_burger(X).
burger(X) :- whopper(X).

big_mac(a).
big_kahuna_burger(k).
big_mac(c).
whopper(d).
```

- When we pose the query enjoys(vincent,k)

- the first rule applies (enjoys := big_kahuna...), and we reach the cut.
- this commits us to the choices we have made, in particular (*enjoy using the second rule and X = k*), blocks access to the second rule. But then we hit fail. This tries to force backtracking, but the cut blocks it, and so our query fails.

- But enjoys(vincent,X) also fails, which is not what we want.
- See the right solution in the next slide.

Better way: Negation as Failure

- We want to say:
  " Vincent enjoys X if X is a burger and X is not a Big Kahuna burger. "
- Try:
  - neg(Goal) :- Goal,!,fail.
  - neg(Goal).
  - enjoys(vincent,X) :- burger(X), neg(big_kahuna_burger(X)).

If-Else

- We can achieve the same result using an if-else construct (but cuts are so widely used that you needed to learn what they are)

  if A then B else C is written as (A -> B ; C).

  - to Prolog this means: try A.
  - if you can prove it, go on to prove B and ignore C.
  - if A fails, however, go on to prove C ignoring B.

  The max predicate using the if-then-else construct looks as follows:
- max(X,Y,Z) :- (X =< Y -> Z = Y; Z = X).

Lists

- Similar to LISP:
  - ?- [Head|Tail] = [mia, vincent, june].
  - Head = mia
  - Tail = [vincent, june].
- To access the 2nd element of a list, you can type:
  - ?- [_ , X|Tail] = [mia, vincent, june].
  - X = vincent
- Writing a predicate to test membership of X in a List:
  - Member(X, [X|T]).
  - Member(X, [Y|T]) :- Member(X,T).
    - 1st rule says, X is a member of a list if it is the first element.
    - 2nd rule says, X is a member of a list if it is not the first element, but is a member of the rest.

Some important rules from gene.pro

```
delete(X) :-
  retract(person(X,_,_,_,_)).

close :-
  retractall(person(_,_,_,_,_)).

save(FileName) :-
  tell(FileName),
  listing(person),
  told.
```

Some important rules from gene.pro

```
%define all possible relation(ship)s in a list
relations( [parent, wife, husband, ancestor, descendent, full_sibling,
    half_sibling, sibling, sister, brother, step_sibling, uncle,
    aunt, mother, father, child, son, daughter, step_parent,
    step_child, step_mother, step_father, step_son, step_daughter,
    nephew, niece, cousin, grandmother, grandfather, grandparent,
    grandson, granddaughter, grandchild]).

%R(X,Y) holds if R is a relation
relation(R, X, Y) :-
    relations(Rs), member(R,Rs),      % if R is a relation(ship)
    Q =.. [R,X,Y],                    % results in Q=R(X,Y)
    call(Q).                          %tests R(X,Y)
```

The Rest is not covered in-depth

Semantic Integrity Checks on Update

```
%this does checks
add_person(Name,Gender,Mother,Father,Spouse) :-
  retractall(message(_)),
  dup_check(Name),
  add(Name,Gender,Mother,Father,Spouse),
  ancestor_check(Name),
  mother_check(Name, Gender, Mother),
  father_check(Name, Gender, Father),
  spouse_check(Name, Spouse).

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

ancestor_check(Name) :-
  ancestor(Name,Name),
  assert(message($Person is their own ancestor/descendent$)),
  !, fail.
ancestor_check(_).
```

```
mother_check(_, _, Mother) :- not(person(Mother)), !.
mother_check(_, _, Mother) :-
  male(Mother),
  assert(message($Person's mother is a man$)),
  !, fail.
mother_check(Name, male, _) :-
  mother(Name, X),
  assert(message($Person, a male, is someone's mother$)),
  !, fail.
mother_check(_,_,_).
```

Arithmetic

- There are more details about Prolog, but we will leave it at that. Those doing Prolog projects need to research and learn further.
- In particular, there is support for arithmetic and numbers, as well as lists:
- Basics of numbers:
  - `?- 8 is 6+2.`
  - `Yes`
  - `?- X is mod(7,2).  //must be an unbound variable`
  - `X=1`
  - `?- 2 < 4.`
  - `Yes`

Arithmetic

- `positive(N) :- N>0.`
- `non_zero(N) :- N<0 ; N>0.`

- `?- X is sqrt(9), Y is 2 ** 4, Z is floor(3.14).`
- `X = 3.0`
- `Y = 16.0`
- `Z = 3`

- `minimum(X,Y,X) :- X<Y.`
- `minimum(X,Y,Y) :- X>=Y.`

It's a bit like:

```
void minimum(int x, int y, int & z)
{
     if  (x < y)
          z = x;
     else z = y;
}
```