Prolog, an introduction

Logic programming

- we defines facts and rules and give this to the logic program
- Ask it what we want to know
- It will look and reason, using available facts and rules, and then tells us an answer (or answers)

Fact and rule

- Comes from Horn clause
  - H←B1,B2,…Bn
  - Which means if all the Bs are true, then H is also true
- In Prolog, we write fact in the form
  - predicate(atom1,….)
  - Predicate is a name that we give to a relation
  - An atom is a constant value, usually written in lower case
  - Fact is the H part of horn clause
- Rule is in the form
  - predicate(Var1,…):- predicate1(…), predicate2(…), …
  - Where Var1 is a variable, usually begins with upper case
  - Yes, it's just a rewriting of
    - H←B1,B2,…Bn
  - Fact is a rule that does not have the right hand side.

Means "and"

Prolog reasoning

- If we have this fact and rule
  - rainy(london).
  - rainy(bangkok).
  - dull(X):- rainy(X).
  - We can ask (or query) prolog on its command prompt
    - ?- dull(C). (is there a C that makes this predicate true?)
    - It will automatically try to substitute atoms in its fact into its rule such that our question gives the answer true
    - in this example, we begin with dull(X), so the program first chooses an atom for X, that is london (our first atom in this example)
    - The program looks to see if there is rainy(london). There is!
    - So the substitution gives the result "true"
  - The Prolog will answer
    - C= london
  - To find an alternative answer, type ";" and "Enter"
  - It'll give C= bangkok
  - If it cannot find any more answer, it will answer "no"

How to ask question

- First, write a prolog program in a .pl file.
- Then load the file, using a prolog interpreter. Or use the consult command:

  ?- consult('file.pl').

  Do not forget this.

  If you want to load the same program again, use reconsult. -> prevent two copies in memory.

  A backslash in the file name may have to be written twice, such as c:\\myprog.pl

- Then you can ask question.
- To exit, use command:

  halt.

Example 2

```
/* Clause 1 */  located_in(atlanta,georgia).
/* Clause 2 */  located_in(houston,texas).
/* Clause 3 */  located_in(austin,texas).
/* Clause 4 */  located_in(toronto,ontario).
/* Clause 5 */  located_in(X,usa) :- located_in(X,georgia).
/* Clause 6 */  located_in(X,usa) :- located_in(X,texas).
/* Clause 7 */  located_in(X,canada) :- located_in(X,ontario).
/* Clause 8 */  located_in(X,north_america) :- located_in(X,usa).
/* Clause 9 */  located_in(X,north_america) :- located_in(X,canada).
```

- To ask whether atlanta is in georgia:

?- located_in(atlanta,georgia).

  – This query matches clause 1. So prolog replies "yes".

?- located_in(atlanta,usa).

This query can be solve by calling clause 5, and then clause 1. So prolog replies "yes".

?-located_in(atlanta,texas).

this query gets “no” as its answer because this fact cannot be deduced from the knowledge base.

The query **succeeds** if it gets a “yes” and **fails** if it gets a “no”.

Prolog can fill in the variables

?- located_in(X, texas).

This is a query for prolog to find X that make the above query true.

- This query can have multiple solutions: both houston and austin are in texas.
- What prolog does is: find one solution and asks you whether to look for another.
- ->

- The process will look like

X = houston

More (y/n)? y

X = austin

More (y/n)? y

no

Some implementations let you type semicolon without asking any question.

Cannot find any more solution

Sometimes it won't let you ask for alternatives

- This is because:
  - Your query prints output, so prolog assumes you do not want any more solutions. For example:

    ?- located_in(X,texas), write(X). will print only one answer.

    Print out
  - Your query contains no variable. Prolog will only print "yes" once.

What about printing all solutions

- To get all the cities in texas:

?-located_in(X,texas), write(X), nl, fail.

New line

Rejects the current solution. Forcing prolog to go back and substitutes other alternatives for X.

located_in is said to be nondeterministic

- Because there can be more than one answer.

Any of the arguments can be queried

?- located_in(austin,X).

Gets the names of regions that contain austin.

?- located_in(X, texas).

Gets the names of the cities that are in texas.

?- located_in(X,Y).

Gets all the pairs that of located_in that it can find or deduce.

?-located_in(X,X).

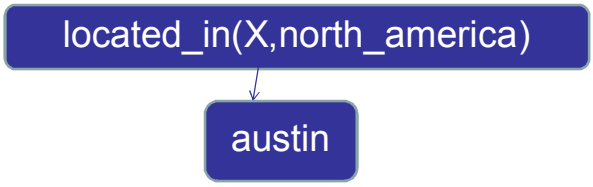
Forces the two arguments to have the same value, which will result in a fail.

Unification and variable instantiation

- To solve a query
  - Need to match it with a fact or the left hand side of a rule.
- Unification is the process of assigning a value to a variable.

?- located_in(austin,north_america).

unifies with the head of clause 8



The right hand side of clause 8 then becomes the new goal.

We can write the steps as follows.

Goal: ?- located_in(austin,north_america).
Clause 8: located_in(X,north_america) :- located_in(X,usa).
Instantiation: X = austin
New goal: ?- located_in(austin,usa).

Clause 5 is tested first but it doesn't work.(we skip that for now)

Goal: ?- located_in(austin,usa).
Clause 6: located_in(X,usa) :- located_in(X,texas).
Instantiation: X = austin
New goal: ?- located_in(austin,texas).

The new goal matches clause 3. no further query. The program terminates successfully.

If no match is found then the program terminates with failure.

<u>X that we substitute in the two clauses are considered to be different.</u>

- Each instantiation applies only to one clause and only to one invocation of that clause.
- X, once instantiated, all X's in the clause take on the same value at once.
- Instantiation is not storing a value in a variable.
- It is more like passing a parameter.

?- located_in(austin,X).

X = texas

?- write(X).

X is uninstantiated

No longer has a value. Since the value is gone once the first query was answered.

backtracking

?- located_in(austin,usa).

If we instantiate it with clause 5, we get:

?- located_in(austin,georgia). , which fails

So how does prolog know that it needs to choose clause 6, not clause 5.

It does not know!

- It just tries the rule from top to bottom.
- If a rule does not lead to success, it backs up and tries another.
- So, in the example, it actually tries clause 5 first. When fails, it backs up and tries clause 6.

?- located_in(toronto,north_america).

See how prolog solve this in a tree diagram.

- tree

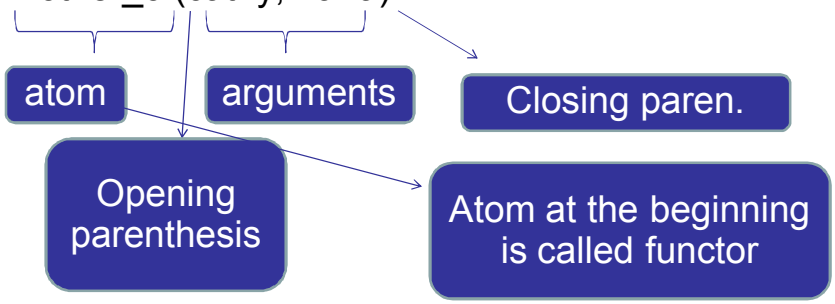
- Backtracking always goes back to the most recent untried alternative.
- Backtracking also takes place when a user asks for an alternative solution.
- The searching strategy that prolog uses is called depth first search.

syntax

- Atom
  - Names of individual and predicates.
  - Normally begins with a lowercase letter.
  - Can contain letters, digits, underscore.
  - Anything in single quotes are also atom:
    - ‘don’’t worry’
    - ‘a very long atom’
    - ‘’

- Structure

mother_of(cathy,maria)



An atom alone is a structure too.

- A rule is also a structure

a(X):- b(X). Can be written as

:- (a(X),b(X))

This is normally infix

- Variable
  - Begin with capital letter or underscore.
  - A variable name can contain letters, digits, underscore.

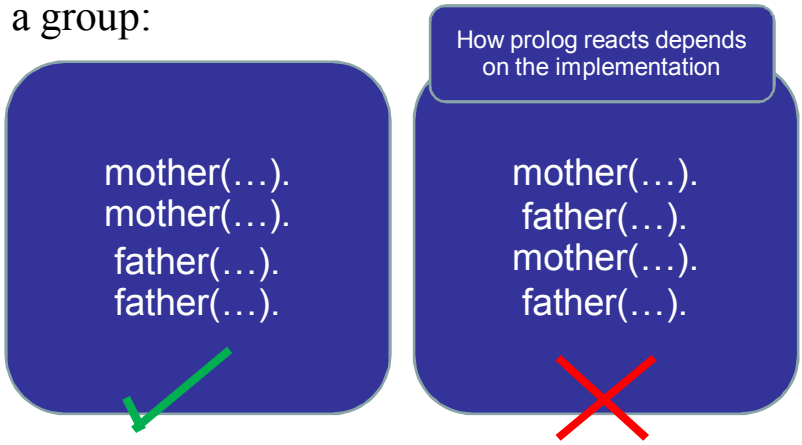
Which_ever
_howdy

- You can insert space or new line anywhere, except to
  - Break up an atom
  - Put anything between a functor and the opening paranthesis
- located_in(toronto,north_america).

Space here is not ok

Space here is ok

- Clauses from the same predicate must be put in a group:



Defining relations

- See example on family tree in the file family.pl

- It can answer questions such as:
  - Who is Cahty's mother?

```
?-mother(X,cathy).
X= melody
```

  - Who is Hazel the mother of?

```
?-mother(hazel, A).
A= michael
A= julie
```

- But there is more!

- We can define other relations in terms of the ones already defined. Such as:

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

The computer will try the first rule. If it does not work or an alternative is requested, it backs up to try the second rule.

-

Conjoined goals ("AND")

- Suppose we want to find out Michael's father and the name of that person's father.

and

?-father(F,michael), father(G,F).

F = charles_gordon G= charles

We get the same answer if we reverse the subgoals' order. But the actual computation is longer because we get more backtracking.

AND can be put in a rule

grandfather(G,C):- father(F,C), father(G,F).

grandfather(G,C):- mother(M,C),father(G,M).

Disjoint goals (“OR”)

- Prolog has semicolon, but it causes error very often, being mistook for comma.
- Therefore it is best to state two rules.
- Like the parent example.

Negative goals (“NOT”)

- \+ is pronounced “not” or “cannot prove”.
- It takes any goal as its argument.
- If g is any goal, then \+g succeeds if g fails, and fails if g succeeds.
- See examples next page

```
?- father(michael,cathy).
yes
?- \+ father(michael,cathy).
no
?- father(michael,melody).
no
?- \+ father(michael,melody).
yes
```

Negation as failure: You cannot state a negative fact in prolog.

So what you can do is conclude a negative statement if you cannot conclude the corresponding positive statement.

- Rules can contain \+ . For example:

```
non_parent(X,Y):- \+ father(X,Y), \+
   mother(X,Y).
```

"X is considered not a parent of Y if we cannot prove that X is a father of Y, and cannot prove that X is a mother of Y"

Here are the results from querying family.pl

?- non_parent(elmo,cathy).

yes

?- non_parent(sharon,cathy).

yes

non_parent fails if we find actual parent-child pair.

?- non_parent(michael,cathy).

no

- What if you ask about people who are not in the knowledge base at all?

?- non_parent(donald,achsa).

yes

Actually, Donald is the father of Achsa, but family.pl does not know about it.

This is because of prolog's CLOSED-WORLDS ASSUMPTION.

- A query preceded by \+ never returns a value.

?- \+ father(X,Y).

It attempts to solve father(X,Y) and finds a solution (it succeeds).

Therefore \+ father(X,Y) fails. And because it fails, it does not report variable instantiations.

Example of the weakness of negation

```
innocent(peter_pan).
Innocent(winnie_the_pooh).
innocent(X):-occupation(X, nun).

guilty(jack_the_ripper).
guilty(X):-occupation(X,thief).

?-innocent(saint_francis).
no
?-guilty(saint_francis).
no
```

Not in database, so he cannot be proven to be innocent.

guilty(X):- \+(innocent(X)). will make it worse.

- The order of the subgoals with \+ can affect the outcome.
- Let's add:

blue_eyed(cathy). Then ask:

cathy

?- blue_eyed(X), non_parent(X,Y).

X= cathy

Can be proven false because a value is instantiated.

?- non_parent(X,Y), blue_eyed(X).

no

This one fails! Because we can find a pair of parent(X,Y).

Negation can apply to a compound goal

```
blue_eyed_non_grandparent(X):-
    blue_eyed(X),
   \+ (parent(X,Y), parent(Y,Z)).
```

You are a blue-eyed non grandparent if: You have blue eye, and you are not the parent of some person Y who is in turn the parent of some person Z.

There must be a space here.

- Finally
- \+ cannot appear in a fact.

Equality

- Let's define sibling:
  - Two people are sibling if they have the same mother.

  Sibling(X,Y):-mother(M,X), mother(M,Y).

  When we put this in family.pl and ask for all pairs of sibling, we get one of the solution as shown:

  X = cathy Y = cathy

  Therefore we need to say that X and Y are not the same.

```
Sibling(X,Y):- mother(M,X), mother(M,Y), \+
   X == Y.
```

Now we get:

X=cathy Y=sharon

X = sharon Y=cathy

These are 2 different answers, as far as prolog is concerned.

- X is an only child if X's mother does not have another child different from X.

```
only_child(X):-mother(M,X),
     \+ (mother(M,Y), \+ X== Y).
```

- == tests whether its arguments already have the same value.
- = attempts to unify its arguments with each other, and succeeds if it can do so.
- With the two arguments instantiated, the two equality tests behave exactly the same.

Equality test sometimes waste time

```
parent_of_cathy(X):-parent(X,Y), Y = cathy.

parent_of_cathy(X):-parent(X,cathy).
```

This one reduces the number of steps.

- But we need equality tests in programs that reads inputs from keyboard since we cannot know the value(s) in advance.

?- read(X), write(X), X = cathy.

Anonymous variable

- Suppose we want to find out if Hazel is a mother but we do not care whose mother she is:

  ?- mother(hazel,_).

  Matches anything, but never has a value.

- The values of anonymous variables are not printed out.
- Successive anonymous variables in the same clause do not take on the same value.

- Use it when a variable occurs only once and its value is never used.

```
is_a_grandmother(X):-mother(X,Y),
parent(Y,_).
```

Cannot be anonymous because it has to occur in 2 places with the same value.

Avoiding endless computation

```
married(michael,melody).
married(greg,crystal).
married(jim,eleanor).
married(X,Y):-married(Y,X).
```

- Lets ask:

```
?- married(don,jane).
```

?- married(don,jane).

- The new goal becomes

?- married(jane,don).

Go on forever!

- We can solve it by defining another predicate that will take arguments in both order.

```
?-couple(X,Y):-married(X,Y).
?-couple(Y,X):-married(X,Y).
```

- loop in recursive call:

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

?-ancestor(cathy,Who).
```

- Prolog will try the first rule, fail, then try the second rule, which gets us a new goal:

```
?- ancestor(cathy,Z), ancestor(Z,Who).
```

This is effectively the same as before. Infinite loop follows.

- To solve it:

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

Force a more specific computation.

- New goal:

```
?-parent(cahty,Z), ancestor(Z,Who).
```

Now it has a chance to fail here.

```
positive_integer(1).
positive_integer(X):-Y is X-1,
   positive_integer(Y).

?- positive_integer(2.5).
```

Will cause infinite call.

Base case is not good enough.

- Two rules call each other:

```
human_being(X):-person(X).
person(X):- human_being(X).
```

- We only need to use one of the rule.

Using the debugger

```
?- spy(located_in/2).
yes
?- trace
yes
?-located_in(toronto,canada).
**(0) CALL: located_in(toronto,canada) ? >
** (1) CALL: located_in(toronto,ontario) ? >
**(1) EXIT: located_in(toronto,ontario) ? >
**(0) EXIT: located_in(toronto,canada) ? >
yes
```

Specify the predicate to trace.

Turn on the debugger.

Enter

Enter

Enter

Enter

```
?-located_in(What,texas).
**(0) CALL: located_in(_0085,texas) ? >
**(0) EXIT: located_in(houston,texas) ? >
What = houston ->;
**(0) REDO: located_in(houston,texas) ? >
**(0) EXIT: located_in(austin,texas) ? >
What = austin->;
**(0) REDO: located_in(austin,texas) ? >
**(0) FAIL: located_in(_0085,texas) ? >
no
```

Uninstantiated variable

Begin a query

Going for alternative solution.

A query has succeeded.

A query fails.

- You can type s for skip and a for abort.
- To turn off the debugger, type:
  ?- notrace.

Styles of encoding knowledge

- What if we change family.pl to:

```
parent(michael, cathy).
parent(melody, cathy).
parent(charles_gordon, michael).
parent(hazel, michael).
male(michael).
male(charles_gordon).
female(cathy).
female(melody).
female(hazel).
father(X,Y):- parent(X,Y), male(X).
mother(X,Y):- parent(X,Y), female(X).
```

Better because information is broken down into simpler concepts.

We know for sure who is male/female.

But you will have to define who is male/female.

- Which is faster the old family.pl or the new ones?
  - Depends on queries.

- Another style is data-record format:

  person(cathy, female, michael, melody).

  father mother

- We can define the rules as follows:

male(X) :- person(X,male,_,_).

father(F,C):- person(C,_,F,_).

This is only good for a conversion from another database.

Example on class taking

```
takes(pai, proglang).
takes(pai, algorithm).
takes(pam, automata).
takes(pam, algorithm).
classmates(X,Y):- takes(X,Z), takes(Y,Z), X\==Y.
```

- When we ask Prolog, we can ask in many ways
  - ?takes(X, proglang).
  - ?takes(pam,Y).
  - ?takes(X,Y)
- Prolog will find X and Y that makes the predicate (that we use as a question) true.

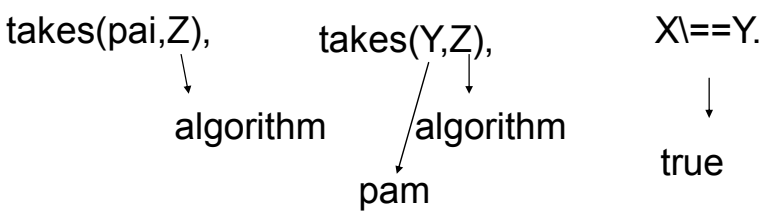
Let's ask ?-calssmates(pai,Y).

- By the rule, the program must look for
  - takes(pai,Z), takes(Y,Z), pai\==Y.
- Consider the first clause, Z is substituted with proglang (because it is in the first fact that we find). So, next step, we need to find
  - takes(Y,proglang). Y is substituted by pai because it is the first fact in the fact list
  - so we will get takes(pai,proglang), takes(pai,proglang), pai\==pai.
  - The last predicate (pai\==pai) will be wrong, so we need to go back to the previous predicate and change its substitution
  - Y cannot have any other value, because only pai studies proglang, so we have to go back to re-substitute Z

- Z is now substituted with algorithm. So, next step, we need to find
  - takes(Y,algorithm). Y is substituted by pai
  - so we will get takes(pai, algorithm), takes(pai, algorithm), pai\==pai.
  - The last predicate (pai\==pai) will be wrong, so we need to go back to the previous predicate and change its substitution
  - Y is re-substituted by pam (her name is next in a similar predicate)
  - so we will get takes(pai, algorithm), takes(pam, algorithm), pai\==pam.
- This is now true, with Y = pam
- So the answer is Y = pam

```
takes(pai, proglang).
takes(pai, algorithm).
takes(pam, automata).
takes(pam, algorithm).
classmates(X,Y):- takes(X,Z), takes(Y,Z), X\==Y.
```

- ?-classmates(pai, Y).



Small point: Testing equality

- ? – a=a.
  - Prolog will answer yes
- ? – f(a,b) = f(a,b).
  - Prolog will answer yes
- ?- f(a,b) = f(X,b).
  - Prolog will answer X=a
  - If we type “;” , it will answer no (because it cannot find anything else that match)

Small point 2:arithmetic

- If we ask ?- (2+3) = 5
- Prolog will answer “no” because it sees (2+3) as a +(2,3) structure
- Prolog thus have a special function
- is(X,Y) this function will compare X and the arithmetic value of Y (there are prefix and infix versions of this function)
- So, asking ?-is(X,1+2). will return X=3
- But asking ?- is(1+2,4-1). will return “no”
  - Because it only evaluates the second argument :P
  - so we should ask ?- is(Y,1+2), is(Y,4-1) instead