Prolog, an introduction

Acknowledgement

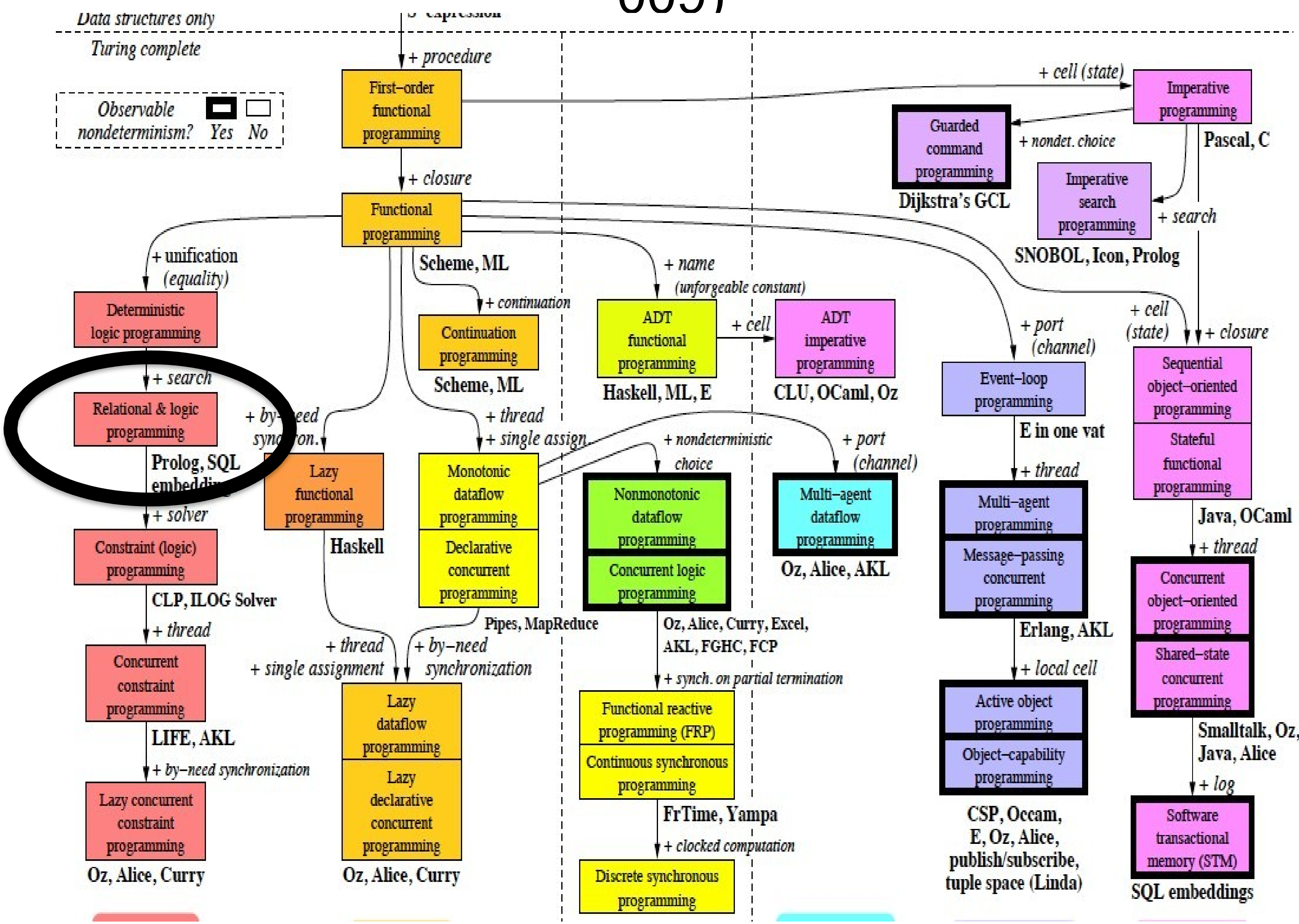
- Peter van Roy (2009): Programming Paradigms for Dummies: What Every Programmer Should Know. In G. Assayag and A. Gerzso (eds.) *New Computational Paradigms for Computer Music*, IRCAM/Delatour, France.
- Lean Sterling and Ehud Shapiro, The art of Prolog, MIT Press, 1999.
- Self teaching site : Learn Prolog now ! http://www.learnprolognow.org
- Free Systems
  - Eclipse Prolog : http://eclipseclp.org
  - SWI Prolog interpreter http://www.swi-prolog.org/

Why learn a new programming paradigm ?

- Is this saw relevant for pruning ?
  - Yes…
  - But to a certain extent only
    - You can extricate your car
    - But it will take hours, you will be more likely to hurt yourself, ...

- If you address a problem with an inapp ropriate programming paradigm, even if you eventually manage
  - it will have taken too much time
  - or/and the program will be of poor quality
    - Bugs, poor performances, …

Ontology of programming paradigms (VanRoy 2009)



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)

Prolog

- One of the most popular and widely used logical programming languages, which stands for "Programming in Logic."
- Prolog allows you to define rules and facts in a knowledge base, which the computer can then use to perform automated reasoning and search for solutions to a given problem.
- Prolog is particularly well-suited for applications such as natural language processing, expert systems, and artificial intelligence, where the ability to reason logically and deduce new knowledge is critical

Functional vs Logical

- While functional and logic programming are different paradigms, there are some similarities between them. For example, both paradigms emphasize the use of declarative programming, in which the programmer specifies what the program should do, rather than how it should do it.

- Additionally, both paradigms emphasize the use of pure functions, which do not have side effects and are deterministic (i.e., always return the same output for a given input).

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"

Is this a program ?

Draw the graph

lali parent_of soso.

lali parent_of ana.

gia parent_of ana.

gia parent_of soso.

ana parent_of mariam.

ana parent_of rezo.

Is this a program ?

lali parent_of soso.

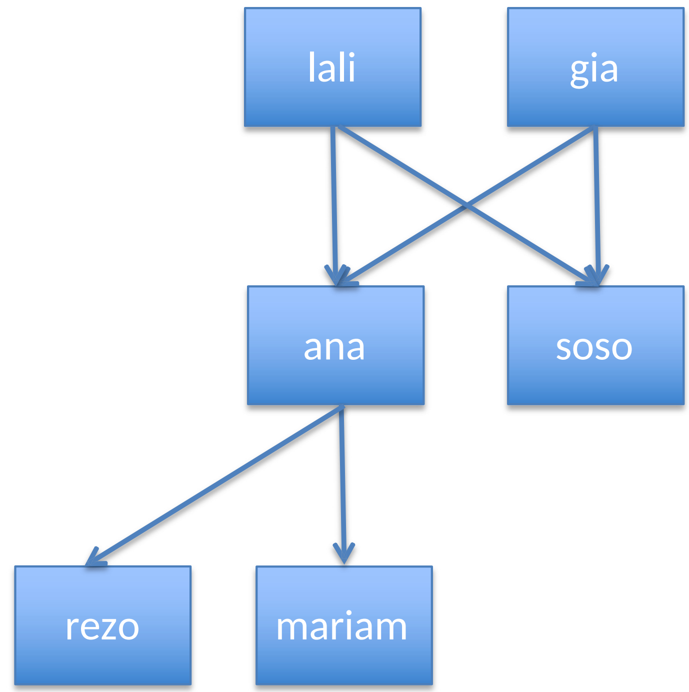
lali parent_of ana.

gia parent_of ana.

gia parent_of soso.

ana parent_of mariam.

ana parent_of rezo.



It is indeed a Prolog progra m !

Is lali a parent of ana?

?- lali parent_of ana
. Yes

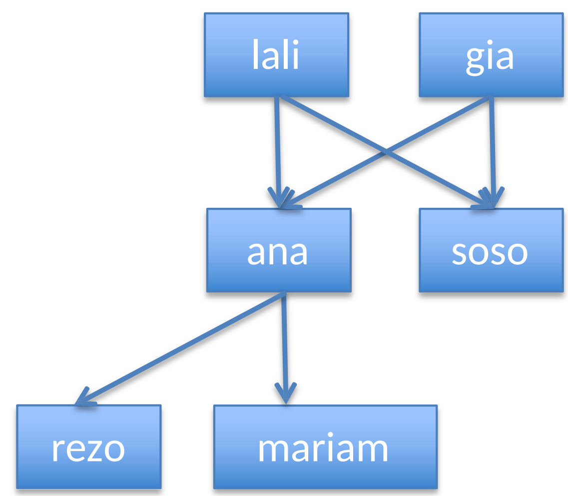
Of whom is lali a parent ?

?- lali parent_of dato.
No

?- lali parent_of X
. X = soso;
X = ana

?- Y parent_of ana
. Y = lali;
Y = gia

lali parent_of soso.
lal parent_of ana.
i gia
parent_of ana. p
gia
ana arent_of soso.
parent_of mariam. ana
parent_of rezo.



Enlarging the program

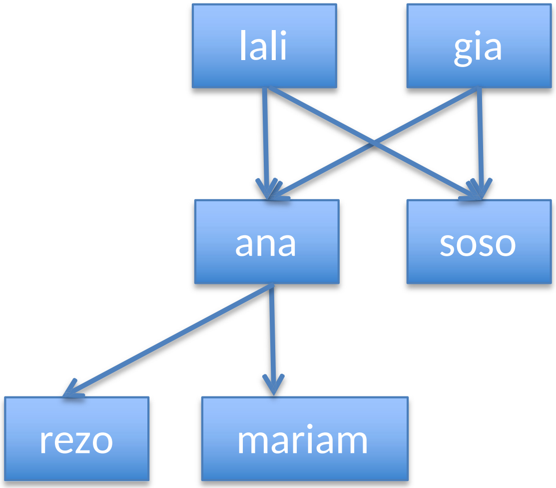
ancestor_of(A, C) :- parent_o
f(A, C). ancestor_of(A, C) :- p
arent_of(A, X),
ancestor_of(X, C).

?- ancestor_of (lali, ana). Ye
s
?- ancestor_of (lali, mariam)
. Yes
?- ancestor_of (soso, maria
m). No
?- ancestor_of(lali, X). X = s
oso ;
X = ana ;
X = mariam ; X = rezo

A person A is an ancestor of another person C if either

A is a parent of C or

A is a parent t of a third person X who is an ancestor of C



Back to the program

6 Facts

```
lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.
```

2 Rules

```
ancestor_of(A, C) :-  parent_of(A, C).
ancestor_of(A, C) :-  parent_of(A, X),
    ancestor_of(X, C).
```

- 8 clauses
  - 6 facts, 2 rules
  - Terminated by a « . »
- 2 **predicates**
  - parent_of
  - ancestor_of
- 6 atoms
  - lali, ana, soso, gia, mariam, rezo
  - They are constants
- 5 Variables
  - A (twice), C (twice), X
  - **Begin with uppercase**
  - **Local to a clause**

Back to the program

```
lali    parent_of
soso.  lali
parent_of          ana.  gia
        parent_of  ana.  gia
         parent_of  soso.
ana   parent_of  mariam.
 ana  parent_of  rezo.
  ancestor_of(A, C) :-
 parent_of(A, C).  anc
estor_of(A, C) :-  pare
              nt_of(A, X),
  ancestor_of(X, C).
```

Head

Body

Implication (⇐) Head is true if body is true

Conjunction (and)

Prolog term s

- Constants
  - Atoms
  - Numbers
- Strings
- Variables
- Lists
- Functors + arguments
  - Ex: parent_of(lali, X),
  - Ex: whatever(Name, another(Y), 3)
  - Number of arguments : **arity**

Declarative progra mming

- We defined **what** is a parent and an ancestor
- We used the program with diffe rent « modes »
  - All parameters instantiated
    - Verification
  - Some parameters or none instanti ated
    - Result generation
    - No predefined input or output

➢ Very powerful programming

```
lali    parent_of soso.
lali parent_of ana.  gia pare
nt_of ana.  gia    parent_of s
oso.
ana parent_of mariam.  ana
parent_of rezo.

ancestor_of(A, C) :-  parent_
of(A, C).
ancestor_of(A, C) :-  parent_
of(A, X),  ancestor_of(X, C).
```

The « magic » comes fr
om

- **Unification**
  - ex: p(23, Y) = p(X, hello) with X/23 and Y/'hello'

and

- **Search tree and Backtracking**
  - search for (more/all) solutions upon failure

Unification (
=)

1. If T_1 and T_2 are constants, then T_1 and T_2 unify if they are the same atom, or the same number
2. If T_1 is a variable and T_2 is any type of term, then T_1 and T_2 unify, and T_1 is instantiated to T_2 (and v
ice versa)
3. If T_1 and T_2 are complex terms then they unify if:
    1. They have the same functor and arity, and
    2. all their corresponding arguments unify, and
    3. the variable instantiations are compatible.

Unification examples

?- lali = lali. Yes

?- lali = ana. No

?- lali = X.

X = lali

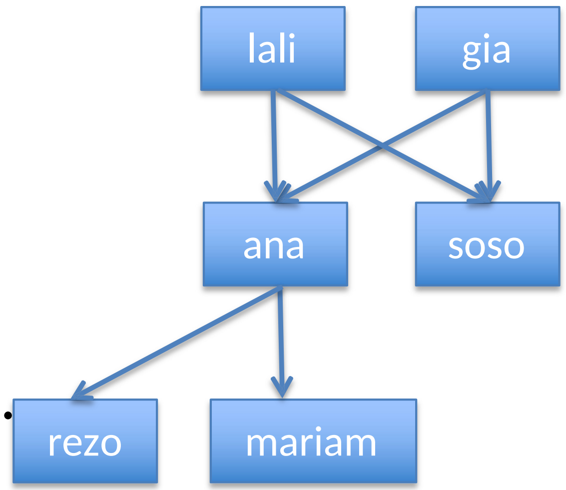
?- parent_of(lali, X) = ancestor_of(lali, ana).

No

?- parent_of(lali, X) = parent_of(lali, ana). X = ana

?- parent_of(lali, X) = parent_of(Y, ana).

X = ana Y = lali



More exampl
es

?- X = lali, X = ana.
No
?- [X | Y] = [a, b, c]
X = a
Y = [b, c]
?- [a | Y] = [X , b, c].
X = a
Y = [b, c]
?- [a | Y] = [X , b | Z]
.
X = a
Y = [b | Z]

The unification algorithm of Robinson

- Input
  - 2 terms T1 and T2 to be unified
- Output
  - θ the most general unifier of T1 and T2
  - or failure
- Initialisation
  - $\theta := \varnothing$, empty substitution
  - stack := [T1 = T2]

    - failure := false

Unification

Unification algorithm 2/2

- while the stack is not empty and `not failure`, pop X = Y, case of
  - X is a variable not occurring in Y: substitute X by Y in the st ack and in θ ; add X / Y in θ
  - Y is a variable not occurring in X: substitute Y by X in the st ack and in θ ; add Y / X in θ
  - X and Y are constants or identical variables: go on
  - X=f(X1, .., Xn) and Y=f(Y1, .., Yn) for a functor f and n > 0 : push Xi=Yi, i=1..n
  - else `failure := true`
- end-while
- if `failure` then return failure else return θ

Exercise 1.1

- Use the previous algorithm to (try to) unify
- 3 and 2+1
  - parent_of(lali, X) and parent_of(Y, ana, Z)
  - parent_of(lali, X) and parent_of(Y, ana)
- f(A,A) and f([3, 2],C)
  - father(X) and X

Prolog search tr ee

Definition: A search tree of a goal ***G*** with respect to a program ***P*** :

−The root is ***G***

−Nodes are goals (*resolvent*), with one <u>selected</u> goal

−There is an edge from a node ***N*** for each clause in the program wh ose head unifies with the selected goal of ***N***

- edges are labeled by the current substitution

Remarks

−Each branch from the root is a computation of ***P*** by ***G***

−Leaves are

- <u>success nodes</u>, where the empty goal has been reached, or
- <u>failure nodes</u>, where the selected goal cannot be further r educed

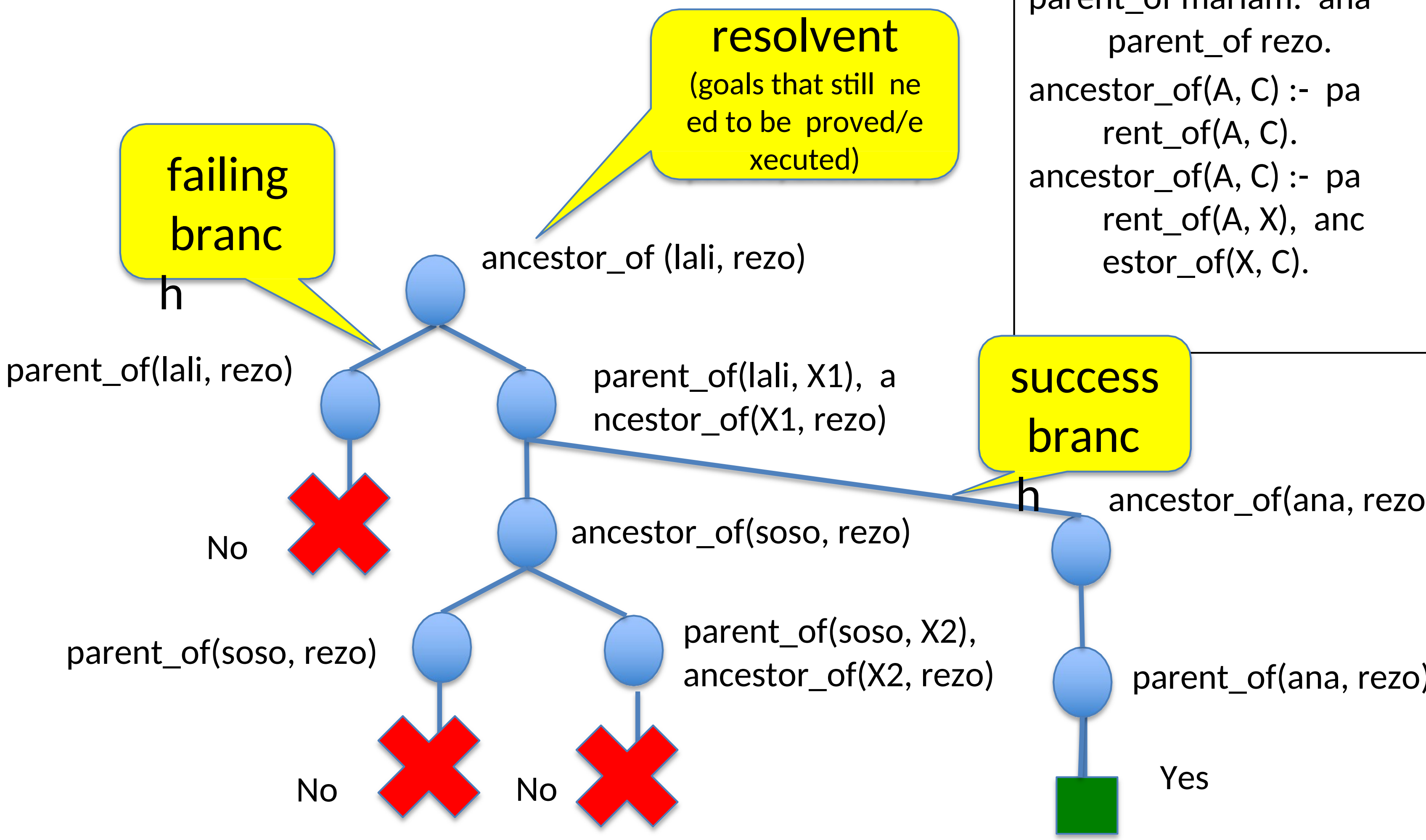
– Success nodes correspond to solutions of the root

Search tree

?- ancestor_of (lali, rezo).

Yes

lali parent_of soso.
lali
parent_of ana. gia
gia parent_of ana.
parent_of soso. ana
parent_of mariam. ana
parent_of rezo.
ancestor_of(A, C) :- pa
rent_of(A, C).
ancestor_of(A, C) :- pa
rent_of(A, X), anc
estor_of(X, C).

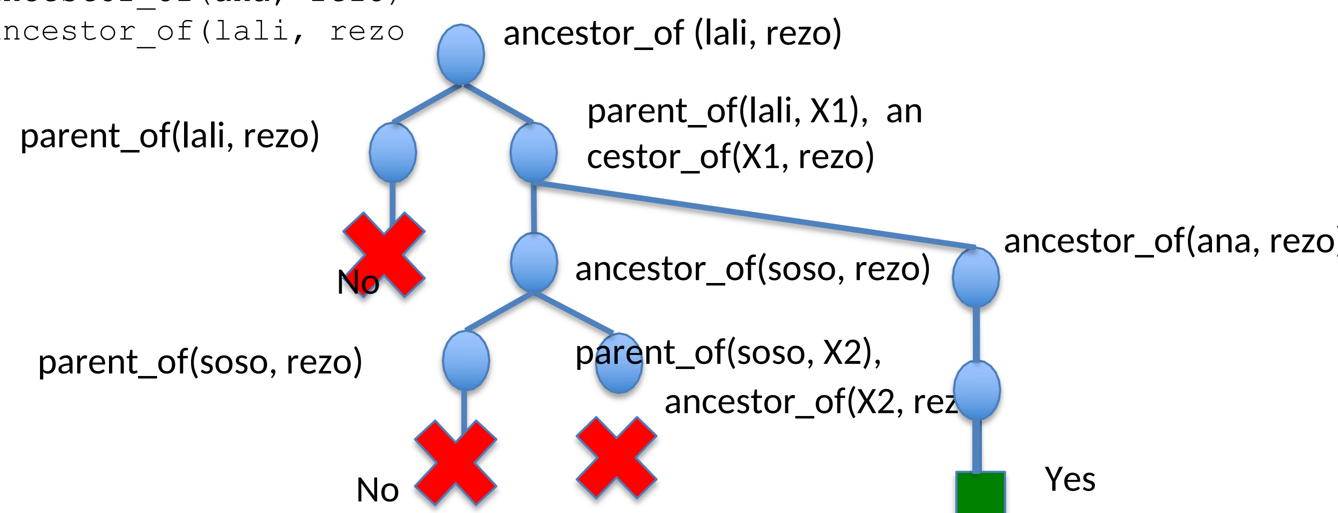


Prolog trace

```
ancestor_of(lali, rezo).
  (1)  1 CALL   ancestor_of(lali, rezo)
  (2)  2 CALL   lali parent_of rezo   anc
  (1)  1 NEXT   estor_of(lali, rezo)   la
  (3)  2 CALL   li parent_of _298  lali
  (3)  2 *EXIT  parent_of soso
  (4)  2 CALL   ancestor_of(soso, rezo)
  (5)  3 CALL   soso parent_of rezo
  (5)  3 FAIL   ... parent_of ...
  (4)  2 NEXT   ancestor_of(soso, rezo)
  (6)  3 CALL   soso parent_of _535
  (6)  3 FAIL   ... parent_of ...
  (4)  2 FAIL   ancestor_of(..., ...)
  (3)  2 REDO   lali parent_of _298
  (3)  2 EXIT   lali parent_of ana
  (7)  2 CALL   ancestor_of(ana, rezo)
  (8)  3 CALL   ana parent_of rezo
  (8)  3 EXIT   ana parent_of rezo
  (7)  2 *EXIT   ancestor_of(ana, rezo)
  (1)  1 *EXIT   ancestor_of(lali, rezo
                 )
```

lali   parent_of soso. lali
parent_of ana. gia
parent_of ana.
gia
parent_of soso. ana parent_of
mariam. ana parent_of rezo.

ancestor_of(A, C) :-
parent_of(A, C). ancestor_of(A,
C) :-
parent_of(A, X),
ancestor_of(X, C).



Exercise 1.2

- What happens if we exchange the first 2 lines of the program ?

?- ancestor_of (lali, rezo).

lali
parent_of soso. lali

parent_of ana. gia
gia parent_of soso.
ana parent_of ana.
parent_of mariam. ana
parent_of rezo.
ancestor_of(A, C) :-
parent_of(A, C). ancesto
r_of(A, C) :-
parent_of(A, X), ancesto
r_of(X, C).

?- ancestor_of (lali, rezo).

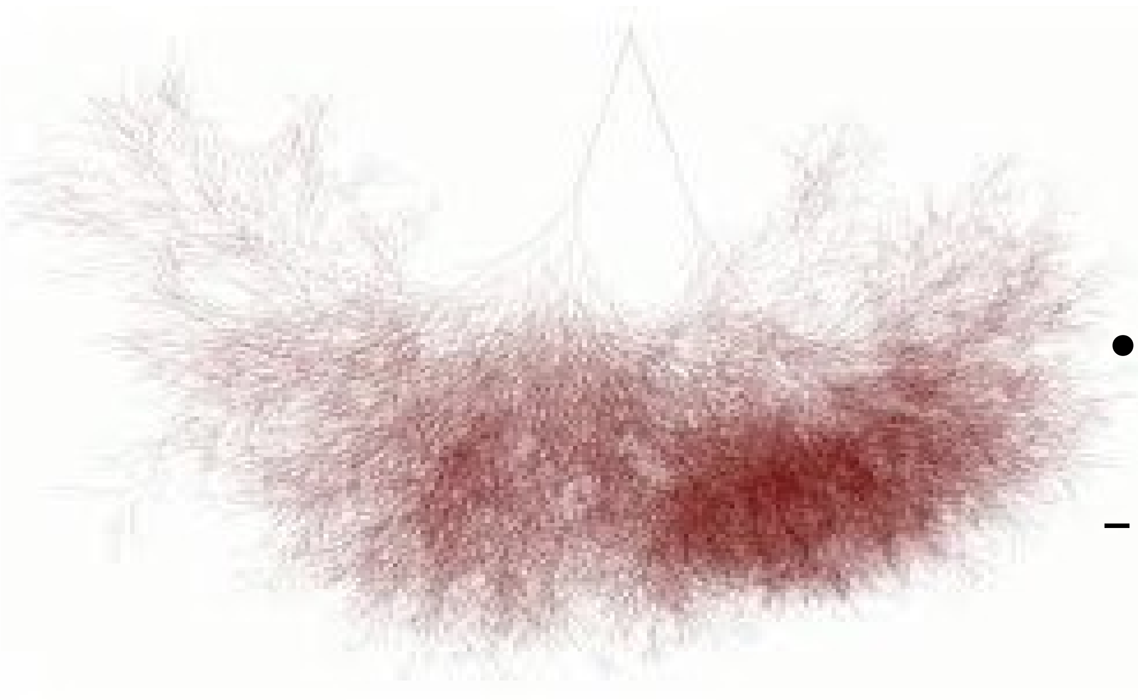
lali
parent_of ana. lali

parent_of soso. gia
gia parent_of soso.
ana parent_of ana.
parent_of mariam. ana
parent_of rezo.
ancestor_of(A, C) :-
parent_of(A, C). ancesto
r_of(A, C) :-
parent_of(A, X), ancesto
r_of(X, C).

Does it change the result ?

Does it change t he search tree ?

Execution trees can be large



- Prolog run time system can cope for **hundreds of thousands** of no des
- When search sp ace too large
  – time to consider **Co nstraint Logic Prog ramming**

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).

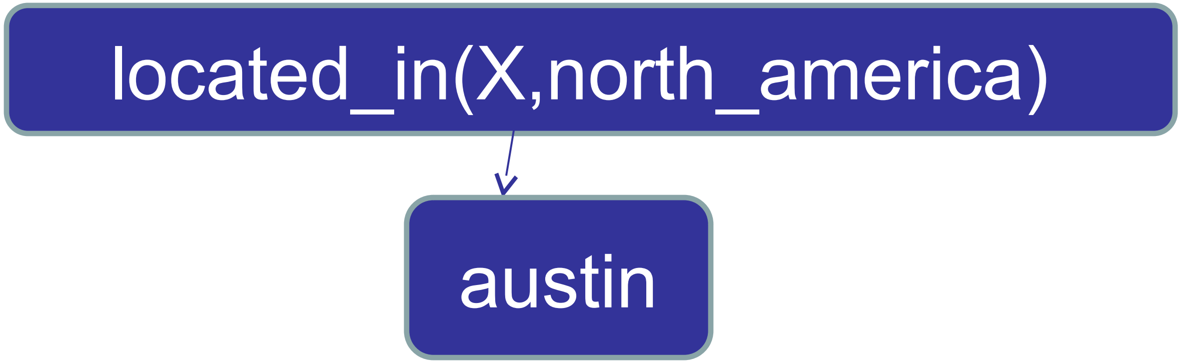
Forces the two arguments to have the same value, which will result in a fail.

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).

Goal: ?- located_in(austin,usa).

Clause 6: located_in(X,usa) :- located_in(X,texas

Instantiation: X = austin

New goal: ?- located_in(austin,texas).

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

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

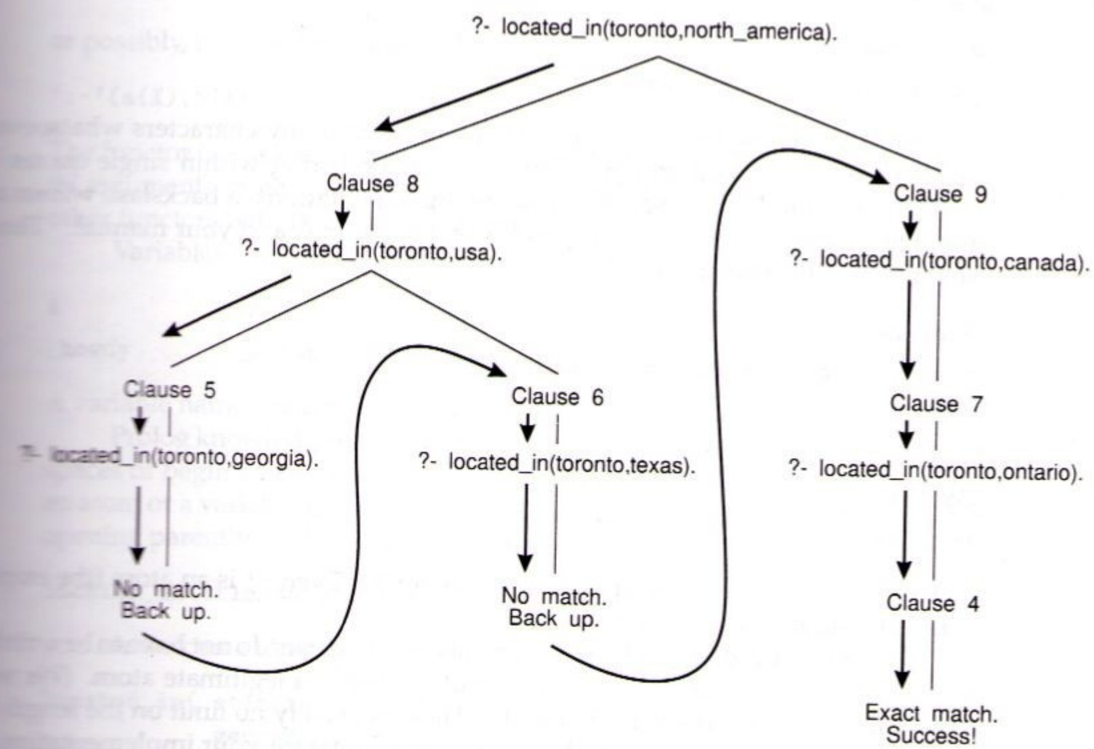


Figure 1.3 The computer searches the paths in this order.

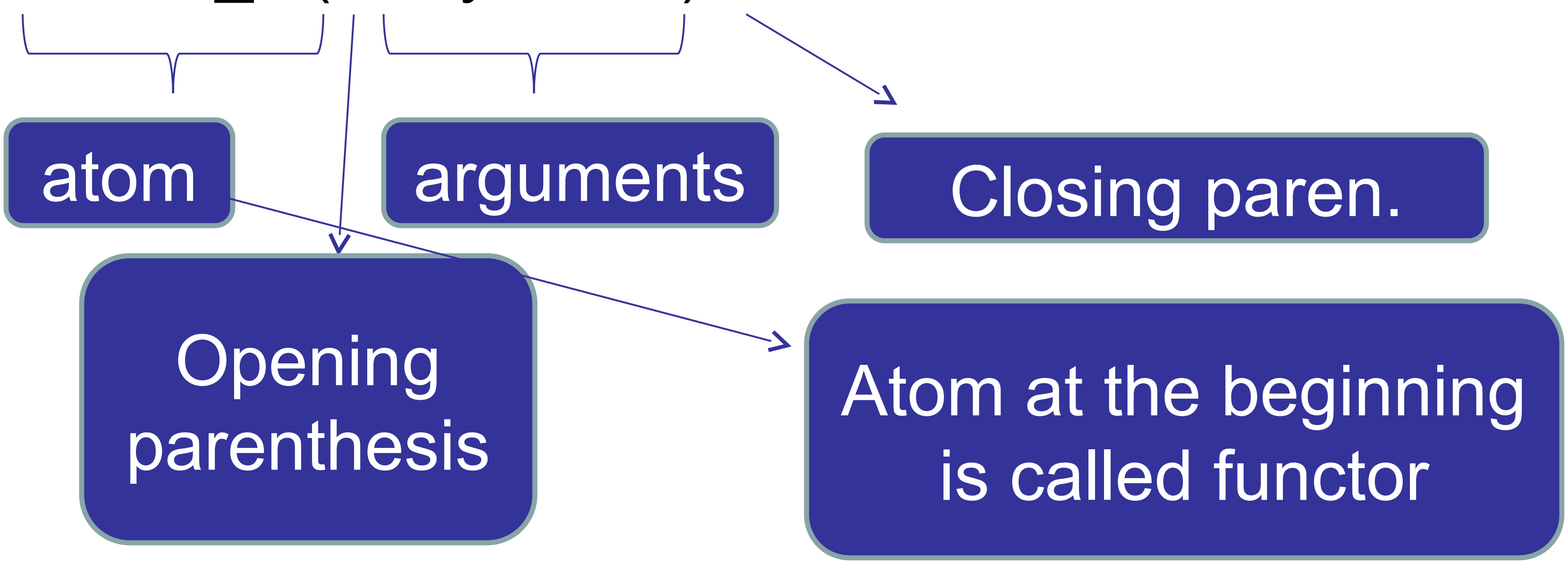
- Backtracking always goes back to the most recent untried alternative.
- Backtracking also takes place when a user asks for an alternative solution.
- The searching strategy that prolog uses is called depth first search.

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:

mother(…).
mother(…).
father(…).
father(…).

How prolog reacts depends on the implementation

mother(…).
father(…).
mother(…).
father(…).

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.

Exercise

- Write the “ancestor_of” Prolog code corresponding to **your own family**,
  - going back at least to grandparents
  - Include at least sisters, brothers, cousins, aunts and uncles
  - Start from the “ancestor.pl” file in the Moodle page
- Run the program
  - on EclipseClp
  - or SWI Prolog
    - Command : **`swipl`**
    - **`Or Online version : https://swish.swi-prolog.org`**

Exercise

- Now program rules to find/check
  - sibling/2
  - aunt_or_uncle/2
  - cousin/2
  - grandparent/2

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:

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

X= cathy

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

no

cathy

Can be proven false because a value is instantiated.

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).

Specify the predicate to trace.

yes

?- trace

Turn on the debugger.

yes

?-located_in(toronto,canada).

**(0) CALL: located_in(toronto,canada) ? >

Enter

** (1) CALL: located_in(toronto,ontario) ? >

Enter

**(1) EXIT: located_in(toronto,ontario) ? >

Enter

**(0) EXIT: located_in(toronto,canada) ? >

Enter

yes

?-located_in(What,texas).

Uninstantiated variable

Begin a query

**(0) CALL: located_in(_0085,texas) ? >

**(0) EXIT: located_in(houston,texas) ? >

What = houston ->;

**(0) REDO: located_in(houston,texas) ? >

Going for alternative solution.

**(0) EXIT: located_in(austin,texas) ? >

What = austin->;

**(0) REDO: located_in(austin,texas) ? >

**(0) FAIL: located_in(_0085,texas) ? >

A query has succeeded.

no

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.

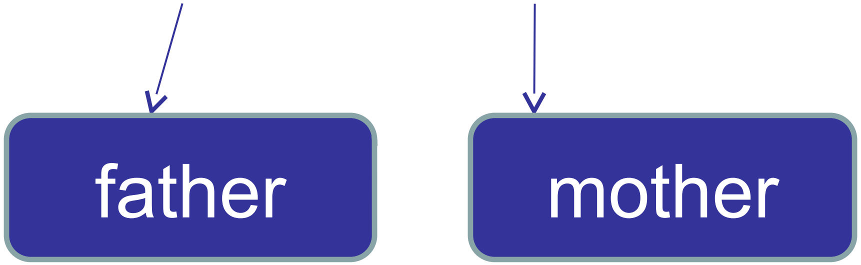
We know for sure who is male/female.

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).



- 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).

takes(pai,Z), takes(Y,Z), X\==Y.

algorithm algorithm true

pam

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