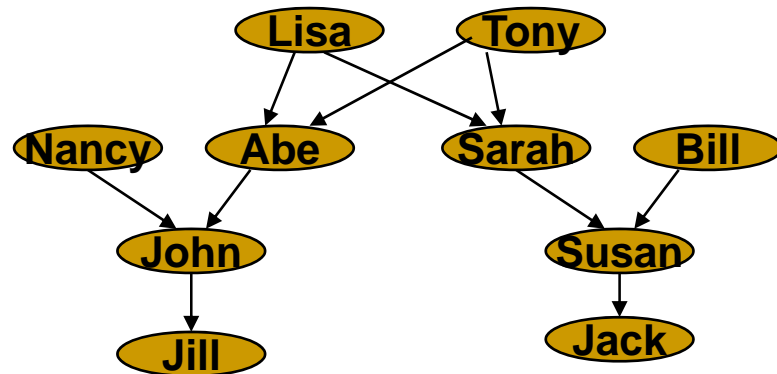

INTRODUCTION TO LOGIC PROGRAMMING

(Lecture 2)

- SRIVIDYA KONA

Database programming - Example

mother(lisa, abe).
mother(lisa, sarah).
mother(nancy, john).
mother(sarah, susan).
mother(susan, jack).
father(tony, abe).
father(tony, sarah).
father(abe, john).
father(bill, susan).
father(john, jill).

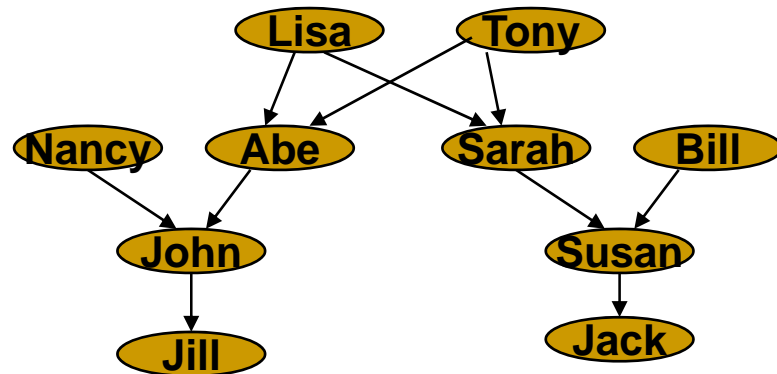


```
% pl (SWI Prolog)
% sicstus (Sicstus Prolog)
?- consult('family.pl').

?- mother(lisa, abe). -- > query
yes
?- mother(lisa, X). -- > query
X = abe;
X = sarah;
```

Database programming - Example

mother(lisa, abe).
mother(lisa, sarah).
mother(nancy, john).
mother(sarah, susan).
mother(susan, jack).
father(tony, abe).
father(tony, sarah).
father(abe, john).
father(bill, susan).
father(john, jill).



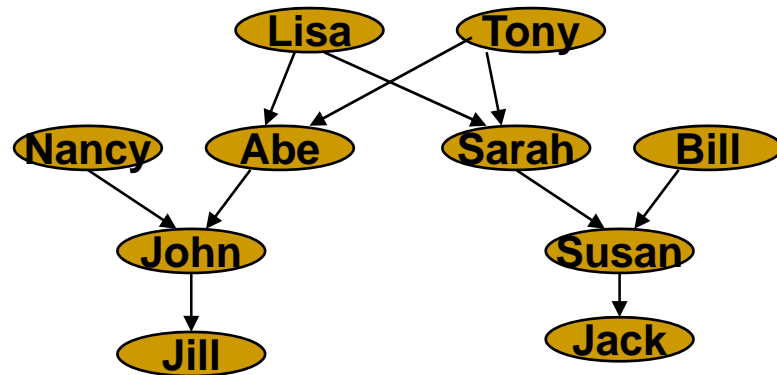
?- mother(lisa, john). -- > query
no

?- mother(X, sarah). -- >
X = lisa;

?- mother(X, Y). -- > query
5 answers

parent rule

mother(lisa, abe).
mother(lisa, sarah).
mother(nancy, john).
mother(sarah, susan).
mother(susan, jack).
father(tony, abe).
father(tony, sarah).
father(abe, john).
father(bill, susan).
father(john, jill).



parent(X,Y) is true if X is parent of Y

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

?- parent(X, sarah).

X = lisa

X = tony

grandparent rule

X is the grand parent of Y if:
X is the parent of Z and
Z is the parent of Y.

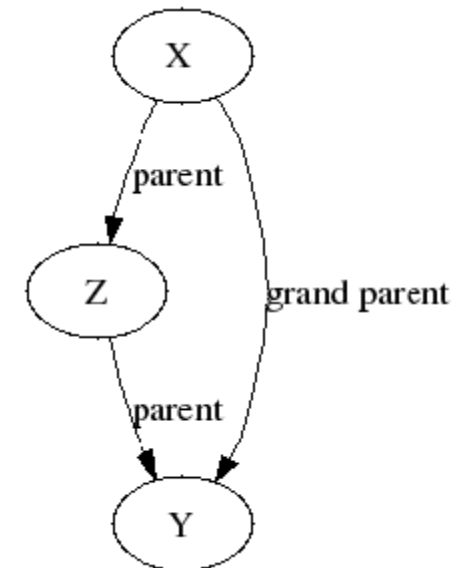
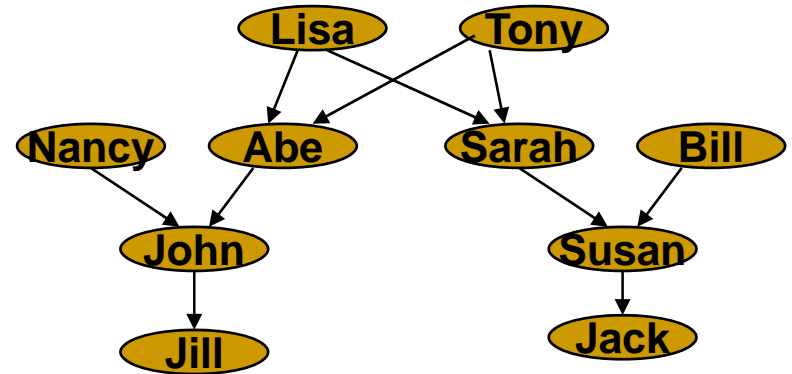
grandparent(X,Y):-
parent(X,Z), parent(Z,Y).

Who is the grandparent of john?

?- grandparent(X,john).

X = lisa;

X = tony



greatgrandparent rule

Define greatgrandparent using existing rules:

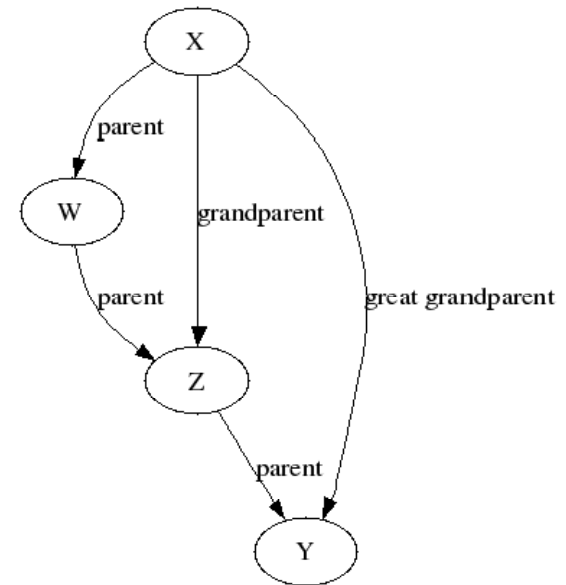
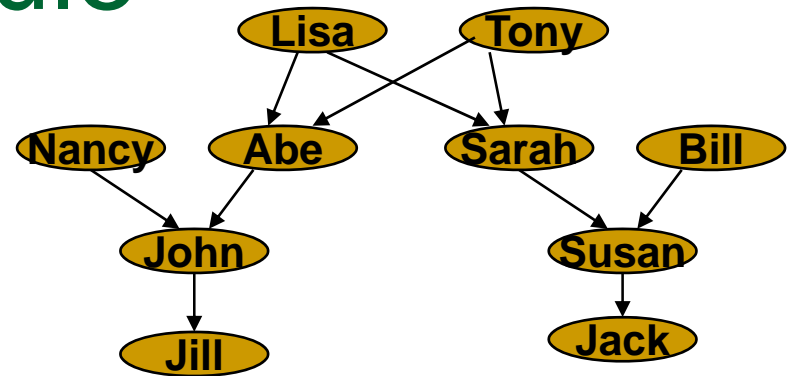
```
greatgrandparent(X,Y):-  
    grandparent(X,Z),  
    parent(Z,Y).
```

Who is the greatgrandparent of jill?

?- greatgrandparent(X,jill).

X = lisa;

X = tony



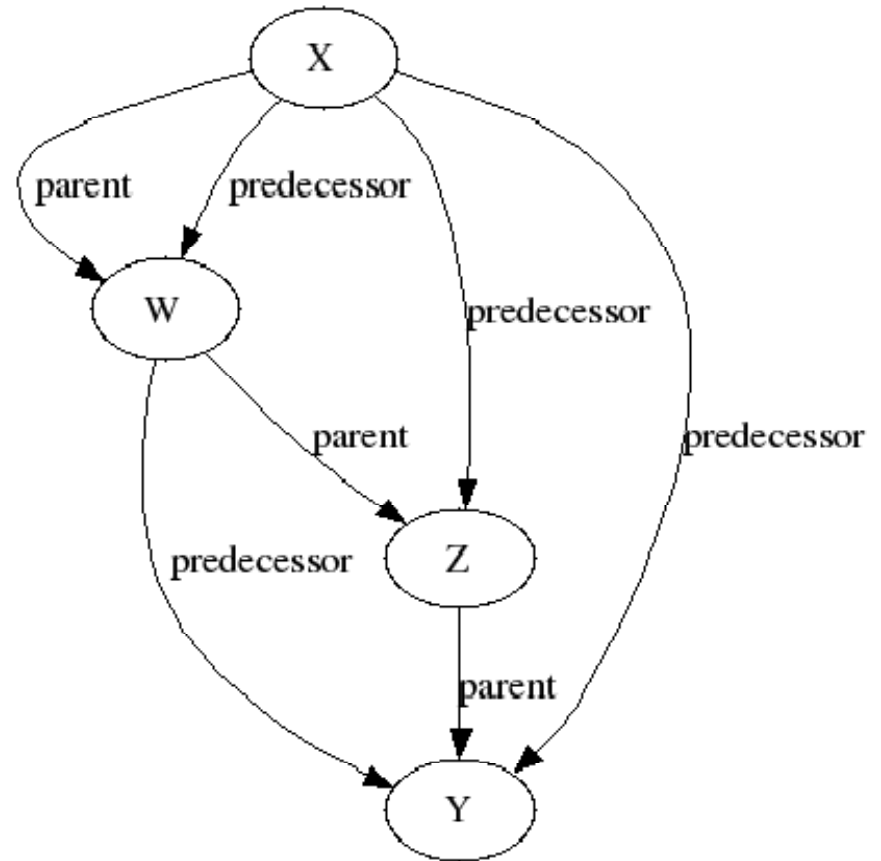
predecessor rule

Grandparent and great grandparent are specializations of a *predecessor* relation.

Definition of predecessor rule:

```
/* rule 1: the terminate condition */  
predecessor(X,Z) :-  
    parent(X,Z).
```

```
/* rule 2: the continue condition */  
predecessor(X,Z) :-  
    parent(X,Y),  
    predecessor(Y,Z).
```



Lists

- Common data structure in nonnumeric programming.
- Ordered sequence of elements that can have any length.
- Ordered: The order of elements in the sequence matters.
- Elements of a list are terms:
 - Constants
 - Variables
 - Structures
 - Lists.
- Can represent practically any kind of structure used in symbolic computation.

Lists

A list in PROLOG is either

- the empty list `[]`, or
- a structure `.(h, t)` where `h` is any term and `t` is a list.
- `h` is called the head and `t` is called the tail of the list `.(h, t)`.

Examples:

`[]`.

`.(a, [])`.

`.(a, .(b, []))`.

`.(a, .(a, .(1, [])))`.

`.(.(f (a,X), []), .(X, []))`.

`.([], [])`.

List Notation

Syntactic sugar provided by Prolog for lists:

- Elements separated by comma.
- Whole list enclosed in square brackets.

Example

`.(a, [])` is written as `[a]`

`.(.(X, []), .(a, .(X, [])))` is written as `[[X], [a,X]]`

`.([], [])` is written as `[[]]`

List Manipulation

Splitting a list L into head and tail:

- Head of L — the first element of L .
- Tail of L — the list that consists of all elements of L except the first.

Special notation for splitting lists into head and tail:

- $[X|Y]$, where X is the head (term) and Y is the tail (list)
- $[a|[b,c,d]]$ is the list $[a,b,c,d]$
- $[a|b]$ is a Prolog term that corresponds to $.(a, b)$. It is not a list

Lists – ‘islist’ predicate

- Check if a given input is a list:
 - Empty list is a list
 - Non-empty list, check the first element and then recursively check if the remaining tail is a list

`islist([]).`

`islist([Head|Tail]) :- islist(Tail).`

- Examples:

`?- islist([1,3]).`

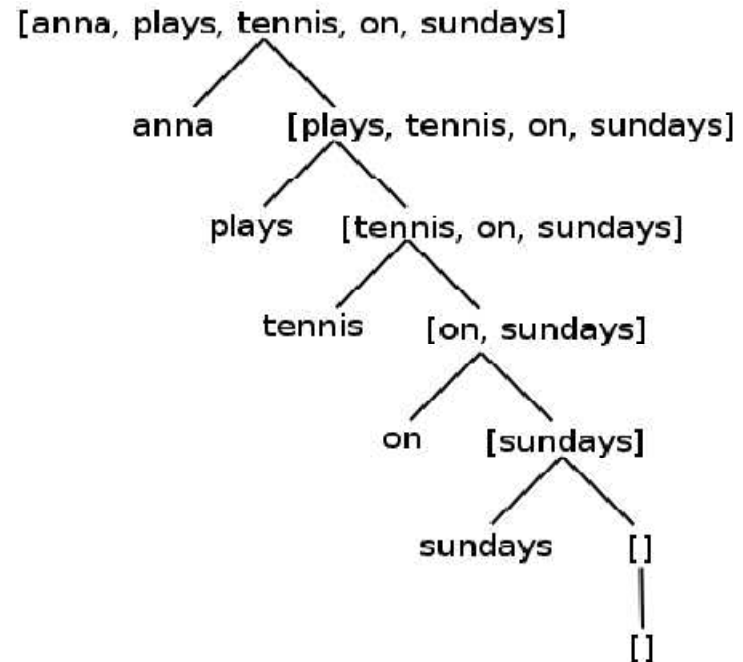
yes

`?- islist(f(a)).`

no

Lists represented as a tree

- Tree representation of a list



Lists – ‘member’ predicate

`member(X,Y)` is true when `X` is a member of the list `Y`.

- `X` is a member of the list if `X` is the same as the head of the list `Y`
- `X` is a member of the list if `X` is a member of the tail of the list `Y`

`member(X, [X|_]).`

`member(X, [_|Y]) :- member(X, Y).`

Lists – ‘member’ predicate trace

?- member(a, [a,b,c]).

Call: member(a, [a,b,c]) ?

Exit: member(a, [a,b,c]) ?

yes

?- member(b, [a,b,c]).

Call: (1) member(b, [a,b,c]) ?

Call: (2) member(b, [b,c]) ?

Exit: (2) member(b, [b,c]) ?

Exit: (1) member(b, [a,b,c]) ?

yes

Lists – ‘member’ predicate trace

?- member(d, [a,b,c]).

Call: (1) member(d, [a,b,c]) ?

Call: (2) member(d, [b,c]) ?

Call: (3) member(d, [c]) ?

Call: (4) member(d, []) ?

Fail: (4) member(d, []) ?

Fail: (3) member(d, [c]) ?

Fail: (2) member(b, [b,c]) ?

Fail: (1) member(b, [a,b,c]) ?

no

Lists – ‘sorted’ predicate

Define sorted(X)

– checks if X is a sorted list (ascending order)

- An empty list is sorted
- A list with a single element is sorted
- A compound list is sorted if the first 2 elements are in order and the remaining list (after the first element) is sorted

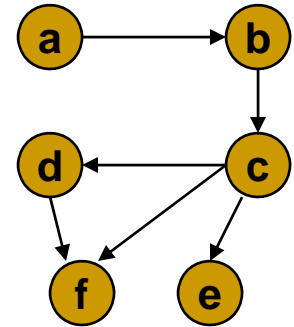
sorted([]).

sorted([X]).

sorted([A, B | T]) :- A =< B, sorted ([B|T]).

Recursion – ‘connected’ predicate

edge(a, b). edge(b, c).
edge(c, d). edge(c, e).
edge(c, f). edge(d, f).



Define connected (X, Y)

- is true if node X is connected to node Y
- there is a sequence of edges starting at node X, and finishing at node Y, which together define a path from X to Y.

connected(X, Y) :- edge(X, Y).

connected(X, Y) :- edge(X, Z), connected(Z, Y).

- **base case** of the recursion: the simplest way in which there is a path between two nodes is if they are directly connected to each other by an edge.
- **recursive case**: expresses that for two nodes X and Y to be connected, is if there is some node Z, to which X is connected, which is in turn connected to Y.

Recursion - Termination problems

- Avoid circular definitions. The following program will loop on any goal involving parent or child:

```
parent(X,Y) :- child(Y,X).  
child(X,Y) :- parent(Y,X).
```

- Use left recursion carefully. The following program will loop on for the query ?- person(X)

```
person(X) :- person(Y), mother(X,Y).  
person(adam).
```

Recursion - Termination problems

- Order of the Rules matter.
- General heuristics: Put facts before rules whenever possible.
- Sometimes putting rules in a certain order works fine for goals of one form but not if goals of another form are generated:
 `islist([_|B]) :- islist (B).`
 `islist([]).`
- works for goals like `islist([1,2,3])`, `islist ([])`, `islist(f(1,2))` but loops for `islist(X)`.
- What will happen if you change the order of `islist` clauses?

Recursion – ‘append’ predicate

Predicate `append(L1, L2, L3)`

- is true if L3 is the result of appending L2 to L1

- if L1 is the empty list, then L3 is L2, or
- if L1 is a nonempty list, then the head of L3 is the head of L1 and the tail of L3 is L2 appended to the tail of L1.

Program:

```
append([],L,L).
```

```
append([X|T1], L2, [X|T3]) :- append(T1, L2, T3).
```

Recursion – ‘append’ predicate

?- append([a,b,c], [2,1], [a,b,c,2,1]).

Yes

?- append([a,b,c], [2,1], X).

X = [a,b,c,2,1]

?- append(X, [2,1], [a,b,c,2,1]).

X = [a,b,c]

%prefix

?- append([a,b,c], X, [a,b,c,2,1]).

X = [2,1]

%suffix

?- append(X,Y,[a,b,c,2,1]).

X = [], Y = [a,b,c,2,1];

X = [a], Y = [b,c,2,1];

X = [a,b], Y = [c,2,1];

X = [a,b,c], Y = [2,1];

X = [a,b,c,2], Y = [1];

X = [a,b,c,2,1], Y = [];

no

%generating

Recursion & Lists – length predicate

Predicate `listlen(L,N)` - succeeds if the length of list `L` is `N`.

- (Boundary condition) The empty list has length 0
- (Recursive case) The length of a nonempty list is obtained by adding one to the length of the tail of the list.

Program:

```
listlen([ ],0).
```

```
listlen([H|T],N):- listlen(T,N1), N is N1 + 1.
```

Recursion & Lists – length predicate

listlen ([a,b,c], N).

listlen([b,c], N1), N is N1 + 1.

listlen([c], N2), N1 is N2 + 1, N is N1 + 1.

listlen([], N3), N2 is N3 + 1, N1 is N2 + 1, N is N1 + 1.

N2 is 0 + 1, N1 is N2 + 1, N is N1 + 1.

N1 is 1 + 1, N is N1 + 1.

N is 2 + 1.

N = 3

Accumulator

- Frequent situation: Traverse a PROLOG structure, calculate the result which depends on what was found in the structure.
- At intermediate stages of the traversal there is an intermediate value for the result.
- Common technique: Use an argument of the predicate to represent the "answer so far". This argument is called an accumulator.

Lists – ‘length’ with accumulator

Predicate `listlenacc (L, A, N)`

- succeeds if the length of list `L`, when added the number `A`, is `N`.

- (Boundary condition) For the empty list, the length is whatever has been accumulated so far, i.e. `A`
- (Recursive case) For a nonempty list, add 1 to the accumulated amount given by `A`, and recur to the tail of the list with a new accumulator value `A1`

Program:

```
listlenacc([], A, A).
```

```
listlenacc([H|T], A, N):- A1 is A + 1, listlenacc(T,A1,N).
```

```
listlen(L, N) :- listlenacc(L, 0, N).
```

List – ‘reverse’ predicate

- Predicate `reverse(L, R)`
 - succeeds if R is the reverse of list L
- Program:
`reverse([], []).`
`reverse([H|T], R) :- reverse(T, R1), append(R1, [H], R).`

List – ‘reverse’ with accumulator

- Predicate reverse(L, R)
 - succeeds if R is the reverse of list L
- Program:

```
reverse(List, Rev) :- rev_acc(List, [ ], Rev).  
rev_acc([ ], Acc, Acc).  
rev_acc([X|T], Acc, Rev) :- rev_acc(T, [X|Acc], Rev).
```
- ```
rev_acc([1,2,3], [], R).
rev_acc([2,3], [1], R).
rev_acc([3], [2,1], R).
rev_acc([], [3,2,1], [3,2,1]).
```

# List – ‘delete’ predicate

- Predicate delete (X, L1, L2) (deletes a given element from the list)
  - succeeds when L2 is the list obtained by deleting element X from list L1
  - (termination condition) If the list L1 is empty, then resultant list L2 is also empty
  - If X is the variable we want to delete, and it is the head of the list L1, then recursively delete X from the tail of the list L1
  - If X is the variable that we want to delete and it is different from the head ‘H’ of L1, then ‘H’ becomes the head of the second list L2, and recursively delete X from tail of L1
- Program:

```
del(_, [], []).
del(X, [X|T], L) :- del(X,T,L).
del(X, [H|T1], [H|T2]) :- X \= H, del(X,T1,T2).
```

# List – ‘delete’ predicate trace

?- del(3,[1,2,3,4], List).

Call: (1) del(3, [1, 2, 3, 4], \_G1483) ?

Call: (2) del(3, [2, 3, 4], \_G1549) ?

Call: (3) del(3, [3, 4], \_G1552) ?

Call: (4) del(3, [4], \_G1552) ?

Call: (5) del(3, [], \_G1555) ?

Exit: (5) del(3, [], []) ?

Exit: (4) del(3, [4], [4]) ?

Exit: (3) del(3, [3, 4], [4]) ?

Exit: (2) del(3, [2, 3, 4], [2, 4]) ?

Exit: (1) del(3, [1, 2, 3, 4], [1, 2, 4]) ?

List = [1, 2, 4]

Yes

# List – ‘sumList’ predicate

- Predicate `sumList (L, S)`
  - succeeds when `S` is the sum of elements in list `L`
- Program:
  - `sumList([], 0).`
  - `sumList([H|T], R) :- sumList(T,R1), R is R1+H.`
- Tail Recursive Program:
  - `sumList(L, S) :- sumListAcc(L, 0, S).`
  - `sumListAcc([], A, A).`
  - `sumListAcc([H|T], A, R) :- A1 is A + H, sumListAcc(T, A1, R).`

# List – ‘prefix’ and ‘suffix’ predicate

- Predicate `prefix(P, L)`
  - succeeds when `P` is the prefix of the list `L`
- Program:  
`prefix([ ], _).`  
`prefix([P|Pt], [P|T]) :- prefix(Pt, T).`
  
- Predicate `suffix(S, L)`
  - succeeds when `S` is the suffix of the list `L`
- Program:  
`suffix(S, S).`  
`suffix(S, [H|T]) :- suffix(S,T).`