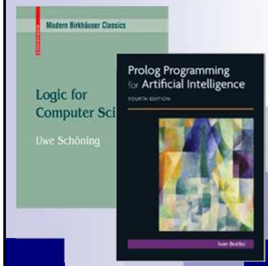


Logic Programming and PROLOG (III)

Dr. Antonio L. Bajuelos

FIU | School of Computing & Information Sciences

FIU | FLORIDA INTERNATIONAL UNIVERSITY

PROLOG. Our previous class. Summary

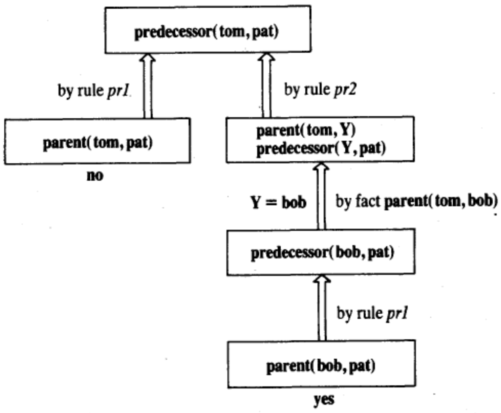
- How **PROLOG** answers questions?
  - **PROLOG** accepts **facts** and **rules** as a set of axioms, and the user's question/query as a conjectured **theorem**; then
  - **PROLOG** tries to prove this theorem - that is, to demonstrate that it can be logically derived from the axioms.
  - The **inference process** is done automatically by the **PROLOG** system and is, in principle, hidden from the user.

2

PROLOG. Our previous class. Summary

How **PROLOG** answers questions?

?- **predecessor(tom, pat).**




```
parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
female(pam).
male(tom).
male(bob).
female(liz).
female(ann).
female(pat).
male(jim).
different(X, Y) :- X \== Y.
offspring(Y,X) :- parent(X,Y).
mother(X,Y) :- parent(X,Y),
               female(X).
grandparent(X,Z) :- parent(X,Y),
                    parent(Y,Z).
sister(X,Y) :- parent(Z,X),
               parent(Z,Y),
               female(X),
               different(X,Y).
pr1 predecessor(X,Z) :- parent(X,Z).
pr2 predecessor(X,Z) :- parent(X,Y),
                        predecessor(Y,Z).
```

3

PROLOG. Our previous class. Summary

Monkey and Banana

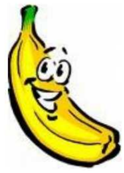
- The problem:
  - There is a **monkey** at the door into a room.
  - In the middle of the room a **banana** is hanging from the ceiling.
  - The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor.
  - At the window of the room there is a **box** the monkey may use.

Can the monkey get the banana?

4

PROLOG. Our previous class. Summary

Monkey and Banana (cont...)

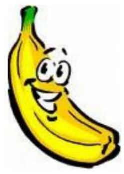


- The **monkey** can perform the following **actions**:
  - **Walk** on the floor
  - **Climb** the box
  - **Push** the box around (if it is already at the box)
  - **Grasp** the banana if standing on the box directly under the banana.

5

PROLOG. Our previous class. Summary.

Monkey and Banana (cont...)



- **Initial State:**
  - Monkey is at the door
  - Monkey is on floor
  - Box is at window
  - Monkey does not have banana
- In **PROLOG**

  state(atdoor, onfloor, atwindow, hasnot).

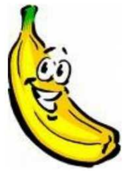
- The **goal of the game** is a situation in which the monkey has the banana; that is, any state in which the last component is **has**:

  state(_, _, _, has).

6

PROLOG. Our previous class. Summary.

Monkey and Banana (cont...)



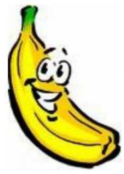
- Move from one state to another
  move(State1, M, State2)
  - **State1** is the state before the move.
  - **M** is the move executed and
  - **State2** is the state after the move.
- The move '**grasp**', can be specified in **PROLOG** as:

```
move(state(middle,onbox,middle,hasnot),   % Before move
     grasp,                               % Move
     state(middle,onbox,middle,has)).     % After move
```

7

PROLOG. Our previous class. Summary.

Monkey and Banana (cont...)



- The other two types of moves, '**push**' and '**climb**', can be similarly specified.
- Example:
  - The move **"push"**:

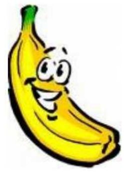
    ```
    move(state(P1, onfloor, P1, H),
    push(P1, P2),
    state(P2, onfloor, P2, H)).
    ```
  - The move **"climb"**:

    ```
    move(state(P, onfloor, P, H),
    climb,
    state(P, onbox, P, H)).
    ```

8

PROLOG. Our previous class. Summary.

Monkey and Banana (cont...)



- Main question our program will pose:

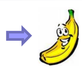
  Can the monkey in some initial state get the banana?

- In terms of PROLOG predicate:

  canget(State).

  where the argument State is a state of the monkey world.

PROLOG



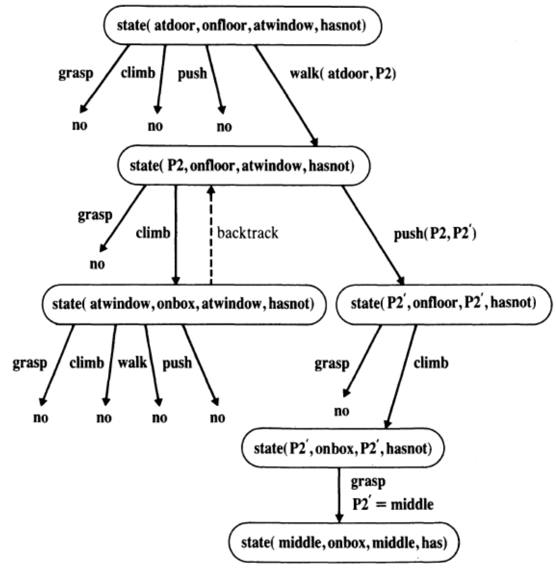
- Then the **Monkey and Banana PROLOG program** is:

```
move(state(middle,onbox,middle,hasnot),
     grasp,
     state(middle,onbox,middle,has)).

move(state(P,onfloor,P,H),
     climb,
     state(P,onbox,P,H)).

move(state(P1,onfloor,P1,H),
     push(P1,P2),
     state(P2,onfloor,P2,H)).

move(state(P1,onfloor,B,H),
     walk(P1,P2),
     state(P2,onfloor,B,H)).

% change(State): monkey can get banana is State

canget(state(_,_,_,has)).          % can 1: Monkey already has it

canget(State1) :-                  % can2: Do some work to get it
    move(State1,Move,State2),      % Do something
    canget(State2).                % Get it now
```

PROLOG

Monkey and Banana (cont...)

- **?- canget(state(atdoor,onfloor,atwindow,hasnot)).**



PROLOG

Order of Clauses and Goal

- Consider the following clause:

  p :- p.

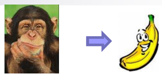
- This says that "**p is true if p is true**".
- This is **declarative perfectly** correct but **procedurally is quite inoperable**.
- In fact, such a clause can cause problems to **PROLOG**.
- Consider the question:

  ? - p.

- Using this clause, the goal **p** is replaced by the same goal **p**; this will be in turn replaced by **p**, etc.
- **PROLOG** will enter an **infinite loop!!!**

PROLOG

Order of Clauses and Goal (cont…)



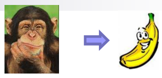
- In our **monkey_and_banana PROLOG** program we have the following clause order:
  - **Grasp**
  - **Climb**
  - **Push**
  - **Walk**
- Effectively says that the monkey <u>prefers grasping to climbing</u>, <u>climbing to pushing</u>, etc…
- This order of preferences helps the monkey to solve the problem.

But what could happen if the order was different?

13

PROLOG

Order of Clauses and Goal (cont…)



But what could happen if the order was different?

- Let assume that the new clause order is:

Walk – Grasp – Climb – Push

- Then the execution of our original goal:

?- canget(state(atdoor, onfloor, atwindow, hasnot)).

This results in an infinite loop!

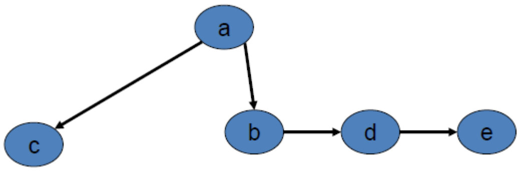
- As the first move the monkey chooses will always be move, therefore he moves aimlessly around the room.
- **Conclusion:**
  - A program in PROLOG may be declaratively correct, but procedurally incorrect (i.e. Unable to find a solution when a solution actually exists).

14

Exercise (homework):

Path finding in in a directed acyclic graph:

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

path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```



(a) If the above **PROLOG** program is problematic then, what is the correct solution?

(b) What if the graph is <u>undirected</u>?

List processing in PROLOG

Representation of lists:

- The **list** is a simple **data structure** commonly used in non-numeric programming.

Example: A list is a sequence of any number of items, such as:

ann, tennis, tom, ball

- This list can be written in **PROLOG** as:

[ann, tennis, tom, ball]

List processing in PROLOG

How can a list be represented as standard PROLOG object?

- For this we need to consider to consider two cases:
  - the list is **empty** or
  - the list is **non-empty**
- In the case of the empty list we write **[]**
- In the second case, the list can be viewed as consisting of:
  - The first item, called the **head** of the list;
  - The remaining part of the list, called the **tail**.
- For our example list:

  [ann, tennis, tom, ball]
- The **head** is **ann** and the **tail** of the list is **[tennis, tom, ball]**

17

List processing in PROLOG

How can a list be represented as standard PROLOG object?

- Let **L = [a,b,c]**
- **PROLOG** provides a notational extension, the **vertical bar**, which separates the **Head** and the **Tail**:

  L = [a | Tail]
- We can list any number of elements followed by '**|**' and the list of remaining items.
- So, alternative ways of writing the above list are:

  [a,b,c] = [a | [b,c]] = [a,b | [c]] = [a,b,c | []]

18

List processing in PROLOG

To summarize:

- A **list** a data structure that is either empty or consist of two parts: a **head** an a **tail**. The **tail** itself has to be a list.
- **PROLOG** provides a special notation for lists, thus accepting lists written as:

  [Item1, Item2, ...]

  or

  [Head | Tail]

  or

  [Item1, Item2, ... | Others]

List processing in PROLOG

Some operations on Lists. Membership

- Now we try to implement the **membership** relation as:

  member(X,L)

  where **X** is an object and **L** is a list.
- The goal **member(X,L)** is **True** is **X** occurs in **L**.
- For example:

  member(b,[a,b,c]) is **True**

  member(b,[a,[b,c]]) is **False**

  member([b,c],[a,[b,c]]) is **True**

List processing in PROLOG

Some operations on Lists. Membership (cont…)

- The program for the **membership relation** can be based on the following observation:
  - **X** is a **member** of **L** if either:

    (1) **X** is the **head** of **L**, or

    (2) **X** is a member of the **tail** of **L**.
- In **PROLOG** this can be written in two clauses: the first is a simple **fact** and the second is a **rule**:

```
member(X, [X|Tail]).
member(X, [Head|Tail]) :- member(X, Tail).
```

- One can read the clauses the following way, respectively:
  - **X is a member of a list whose first element is X.**
  - **X is a member of a list whose tail is Tail if X is a member of Tail.**

21

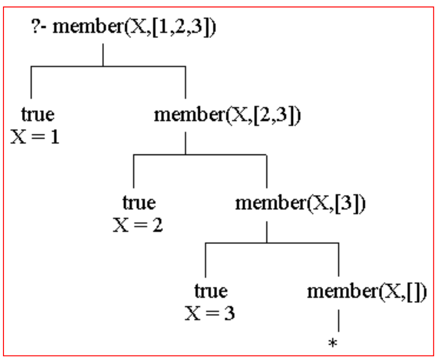
List processing in PROLOG

Some operations on Lists. Membership (cont…)

```
member(X, [X|Tail]).
member(X, [Head|Tail]) :- member(X, Tail).
```

- **Example:**

```
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
false
```



22

List processing in PROLOG

Some operations on Lists. Membership (cont…)

```
member(X, [X|Tail]).
member(X, [Head|Tail]) :- member(X, Tail).
```

- **Example:**

  ?- member([3,Y], [[1,a],[2,m],[3,z],[4,v],[3,p]]).
  Y = z ;
  Y = p ;
  false

  ?- member(X,[23,45,67,12,222,19,9,6]), Y is X*X, Y < 100.
  X = 9   Y = 81 ;
  X = 6   Y = 36 ;
  false

List processing in PROLOG

Some operations on Lists. Membership (cont…)

```
member(X, [X|Tail]).
member(X, [Head|Tail]) :- member(X, Tail).
```

- The definition for the predicate **member** can be written as:

  member(X,[X|_]).

  member(X,[_|Tail) :- member(X, Tail).

- Remember that '_' (underscore) designates a "**don't-care**" variable, usually called **anonymous** variables.
- In general, such variables have names whose first character is the underscore.
- Not having to bind values to anonymous variables saves a little run-space and run-time.

List processing in PROLOG

Some operations on Lists. Concatenation (Append)

- For **concatenation of lists** we will define the relation:

  conc(L1, L2, L3)

  where L1 and L2 are two lists, and L3 is their concatenation.

- For **example:**

  conc([a,b],[c,d],[a,b,c,d]) is **True**

  conc([a,b],[c,d],[a,b,d,c]) is **False**

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

conc(L1, L2, L3)

- For **conc** we have **two cases**, depending on the first argument, L1:

  (1) if the 1st argument is the empty list then the 2nd and the 3rd arguments must be the same. This is expressed by the following **PROLOG** fact:

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

  (2) If the 1st argument of **conc** is non-empty list then it has a **head** and a **tail** and must look like this:

  [X | L1]

- The result of the concatenation is the list **[X | L3]** where L3 is the concatenation of L1 and L2. In **PROLOG**:

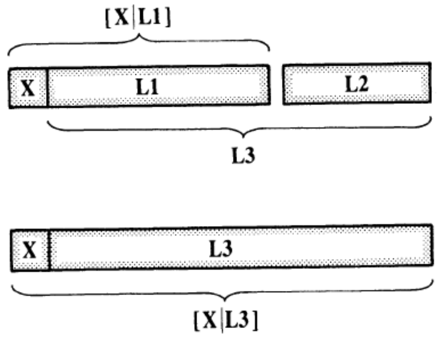
```
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

- Illustration of the concatenation of [X | L1] and some list L2.



27

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

- **Example:**

  ?- conc([a,b,c],[1,2,3],L).

  L = [a,b,c,1,2,3]

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

  L=[a,[b,c],d,a,[],b]

28

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

Example:

conc([1,2],[3,4],A).

conc([X|L1],L2,[X|L3]) :- conc(L1,L2,L3).

conc([1|[2]],[3,4],[1|V1]) :- conc([2],[3,4],V1).   A=[1|V1]

conc([X|L1],L2,[X|L3]) :- conc(L1,L2,L3).

conc([2|[]],[3,4],[2|V2]) :- conc([],[3,4],V2).   V1=[2|V2]

conc([],L,L).

conc([],[3,4],[3,4]).   V2 =[3,4]

List processing in PROLOG

Example:

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

SWI-Prolog (Multi-threaded, version 6.6.6)

File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.6)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
Warning: c:/users/leslie/documents/fiu/spring 2016/cot3541_spring_2016/prolog/list_c
        Singleton variables: [Tail]
Warning: c:/users/leslie/documents/fiu/spring 2016/cot3541_spring_2016/prolog/list_c
        Singleton variables: [Head]
Warning: c:/users/leslie/documents/fiu/spring 2016/cot3541_spring_2016/prolog/list_c
        Singleton variables: [L1,L2]
Warning: c:/users/leslie/documents/fiu/spring 2016/cot3541_spring_2016/prolog/list_c
        Singleton variables: [L1,L3]
% c:/Users/leslie/Documents/FIU/Spring 2016/COT3541_Spring_2016/Prolog/list_operatic
es
1 ?- trace.
true.

[trace] 1 ?- conc([a,b,c],[d,e,f],X).
   Call: (6) conc([a, b, c], [d, e, f], _G532) ? creep
   Call: (7) conc([b, c], [d, e, f], _G617) ? creep
   Call: (8) conc([c], [d, e, f], _G620) ? creep
   Call: (9) conc([], [d, e, f], _G623) ? creep
   Exit: (9) conc([], [d, e, f], [d, e, f]) ? creep
   Exit: (8) conc([c], [d, e, f], [c, d, e, f]) ? creep
   Exit: (7) conc([b, c], [d, e, f], [b, c, d, e, f]) ? creep
   Exit: (6) conc([a, b, c], [d, e, f], [a, b, c, d, e, f]) ? creep
X = [a, b, c, d, e, f].

[trace] 2 ?- notrace.
true.

[debug] 3 ?-
```

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

- We can use **conc** in the inverse direction for decomposing a given list into two list.
- **Example:**
  ?- conc(L1,L2,[a,b,c]).

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

  L1=[]
  L2=[a,b,c];

  L1=[a]
  L2=[b,c];

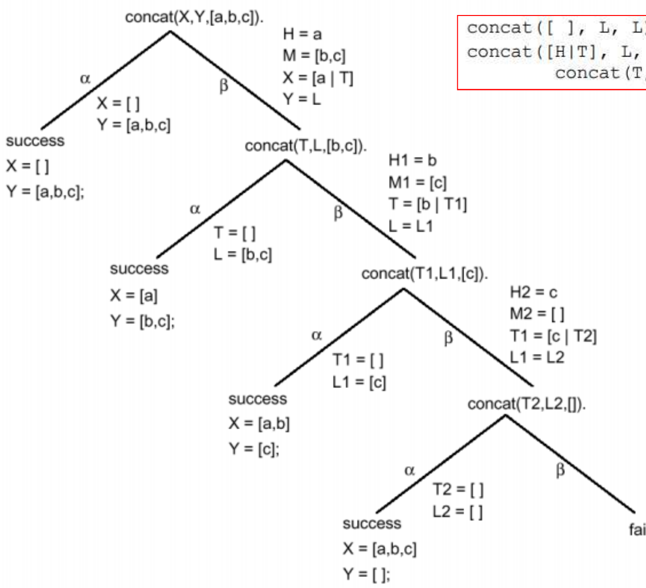
  L1=[a,b]
  L2=[c];

  L1=[a,b,c]
  L2=[];
  false
- It is possible to decompose the list **[a,b,c]** in <u>four ways</u>, all of which were found by our program through backtracking.

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

```
concat([ ], L, L).          % clause a
concat([H|T], L, [H|M]) :-
       concat(T, L, M). % clause b
```



List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

- We can use **conc** to look for a certain pattern in a list.
- **Example:** we can find the months "before" and "after" a given month:

  ?- conc(Before, [may | After],
  [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec]).

  Before = [jan,feb,mar,apr]
  After = [jun,jul,aug,sep,oct,nov,dec]

- Using **conc,** we can find the immediate month before and the immediate month after May by asking:

  ?- conc(_, [Month1,may,Month2 | _],
  [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec]).

  Month1 = apr
  Month2 = jun

33

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).

- Observe that we can use **conc** to <u>delete</u> from some list, L1, everything that follows three successive occurrences of z in L1 together with the three z's.
- **Example:**

  ?- **L1 = [a,b,z,z,c,z,z,z,d,e],**
  conc(L2,[z,z,z | _], L1).

  L1 = [a,b,z,z,c,z,z,z,d,e],
  L2 = [a,b,z,z,c]

34

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

- Note that using **conc** we can redefine the **membership relation** by the clause:

```
member1(X,L) :- conc(L1, [X | L2], L).
```

- % X is a member of list L, if L can be decomposed into two list % and the second one has X as its **head**
- The **member1** clause can be written using **anonymous** variables as:

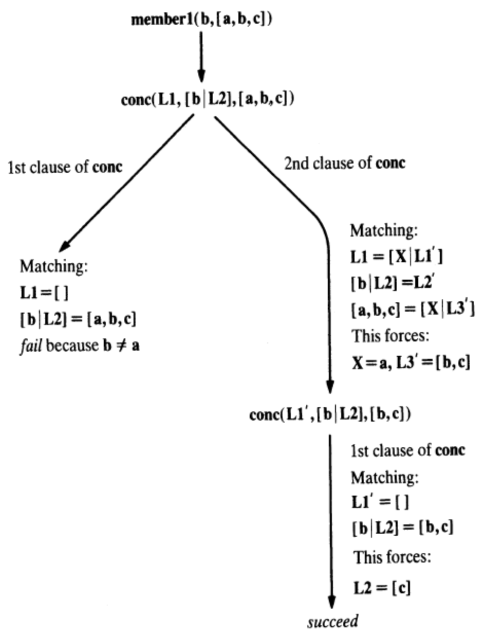
```
member1(X,L) :- conc(_, [X | _], L).
```

35

List processing in PROLOG

- Procedure **member1** finds an item in a given list.

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :-
        conc(L1,L2,L3).

member1(X,L) :-
        conc(L1, [X | L2], L).
```



List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

- Define a predicate **rotate(X,Y)** where both X and Y are represented by lists, and Y is formed by rotating X to the left by one element.
- **Solution:**
  - Take the first element off the first list (H) and append it after the tail (i.e. at the end) in the solution (R)

```
rotate([H|T],R) :- conc(T,[H],R).
```

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1,L2,L3).
```

37

List processing in PROLOG

Some operations on Lists. Concatenation (cont…)

- **Exercises (Homework):**

  (1) Write a query, using **conc**, to **delete** the last three elements from a list **L** producing another list **L1**.
  Hint: **L** *is the concatenation of* **L1** *and a three-element list.*

  (2) Define the predicate
  last(Item, List)
  so that **Item** is the last element of a list **List** using the **conc** relation/predicate.

38

List processing in PROLOG

Some operations on Lists. Adding an item to a list

- To **add** an item to a list the easiest way is to put item in front of the list so that it becomes the new **head**.
- If X is the new item and the list is L then the result is:

  [X | L]
- Using **PROLOG** we can define the **add** procedure as the fact:

  add(X, L, [X | L]).
- **Example:**

  ?- add(a,[1,2,3], L).
  L = [a,1,2,3]

39

List processing in PROLOG

Some operations on Lists. Deleting an item

- **Deleting** an item, X, from a list, L, can be programmed as a relation:

  del(X, L, L1)

  where L1 is equal to the list with the item X removed.
- The **del** relation can be defined similarly to the membership relation. We have, again, two cases:

  (1) If X is the **head** of the list then the result, after deletion, is the tail of the list.

  (2) If X is in the **tail** then it is deleted from there.

  del(X, [X | Tail], Tail).
  del(X, [Y | Tail], [Y | Tail1]) :- del(X, Tail, Tail1).

40

List processing in PROLOG

Some operations on Lists. Deleting an item (cont…)

```
del(X, [X | Tail], Tail).
del(X, [Y | Tail], [ Y | Tail1]) :- del(X, Tail, Tail1).
```

- Note that
  - If there are several occurrences of X in the list then **del** will be able to delete anyone of then by backtracking.
  - each alternative execution of del will only delete one occurrence of X, leaving the other untouched.
- Example

```
?- del(a,[a,b,a,a],L).
L = [b,a,a];
L = [a,b,a];
L = [a,b,a];
false
```

- Note: **del** will fail if the list does not contain the item to be deleted.

41

List processing in PROLOG

Some operations on Lists. Deleting an item (cont…)

```
del(X, [X | Tail], Tail).
del(X, [Y | Tail], [Y | Tail1]) :- del(X, Tail, Tail1).
```

- Observe that **del** can also used to **add** an item to a list by inserting the new item anywhere in the list.
- **Example:**
  - If we want to insert **a** at any place in the list **[1,2,3]** then then we can do this by asking the query:
  - What's **L** such that after deleting **a** from **L** we obtain **[1,2,3]?**

```
?- del(a,L,[1,2,3]).
L = [a,1,2,3];
L = [1,a,2,3];
L = [1,2,a,3];
L = [1,2,3,a];
false
```

42

List processing in PROLOG

Sublist

- Let us now consider the **sublist** relation.
- This relation has two arguments:
  - a list **L** and
  - a list **S** such that occurs in **L** as its sublist
- So

  sublist([c,d,e],[a,b,c,d,e,f]) is **True** and

  sublist([c,e],[a,b,c,d,e,f]) is **False**
- The **sublist relation** can be formulated as:
  - **S** is a **sublist** of **L** if:

    (1) **L** can be decomposed into two lists, **L1** and **L2**, and

    (2) **L2** can be decomposed into two lists, **S** and some **L3**
- In **PROLOG** as:

  sublist(S,L) :- conc(L1,L2,L), conc(S,L3,L2).

List processing in PROLOG

Sublist (cont...)

sublist(S,L) :- conc(L1,L2,L), conc(S,L3,L2).

- Observe that **sublist** relation can also be used, for example, to find all sublists of a given list:

  ?- sublist(S,[a,b,c]).

  S = [];
  S = [a];
  S = [a,b]
  S = [a,b,c]
  S = [b];
  …
  how many?

List processing in PROLOG

Permutation

- We will define the **permutation relation** with two lists such that one is the permutation of the other.
- **Example:**

  ?- permutation ([a,b,c],P).

  P = [a,c,b]

  …
- The intention is to **generate permutations** of a list through **backtracking** using the permutation procedure.
- The procedure for **permutation** can be based on the consideration of two cases, depending on the first list:

  (1) If the first list is empty then the second list must also be empty.

  (2) If the first list is not empty then it has the form [X|L], and the permutation can be constructed as: first permuted L obtaining L1, and the insert X at any position into L1.

List processing in PROLOG

Permutation (cont…)

- …

  (1) f the first list is empty then the second list must also be empty.

  (2) If the first list is not empty then it has the form [X|L], and the permutation can be constructed as:

  first permuted L obtaining L1, and

  insert X at any position into L1.
- The **PROLOG** clauses for the permutation relation are:

```
permutation([], []).
permutation([X|L],P) :-
        permutation (L,L1),
        insert(X,L1,P).
```

List processing in PROLOG

Permutation (cont…)

```
permutation([], []).
permutation([X | L],P) :-
        permutation (L,L1),
        insert(X,L1,P).
```

- **Example:**

?- permutation([red,blue,green],P).

P = [red,blue,green];
P = [red,green,blue];
P = [blue,red,green];
P = [blue,green,red];
P = [green,red,blue];
P =[green,blue,red];
false

47