Logic and Constraint Programming

PROLOG

Prof. Fabrizio Fornari

May 25, 2022

Declarative and Procedural meaning of Prolog programs

P :- Q, R.

Declarative readings: P is true if Q and R are ture.
From Q and R follows P.

Alternative procedural readings:

To solve problem P, *first* solve the subproblem Q and *then* the subproblem R.
To satisfy P, *first* satisy Q and *then* R.

The difference is that the latter also defines the *order* in which the goals are processed.

Declarative meaning of Prolog programs

Determines whether a given goal is true, and if so, for what values of variables it is true.

Given a program and a goal G, the declarative meaning says the following.

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if:

(1) there is a clause C in the pogram such that

(2) there is a clause instance I of C such that

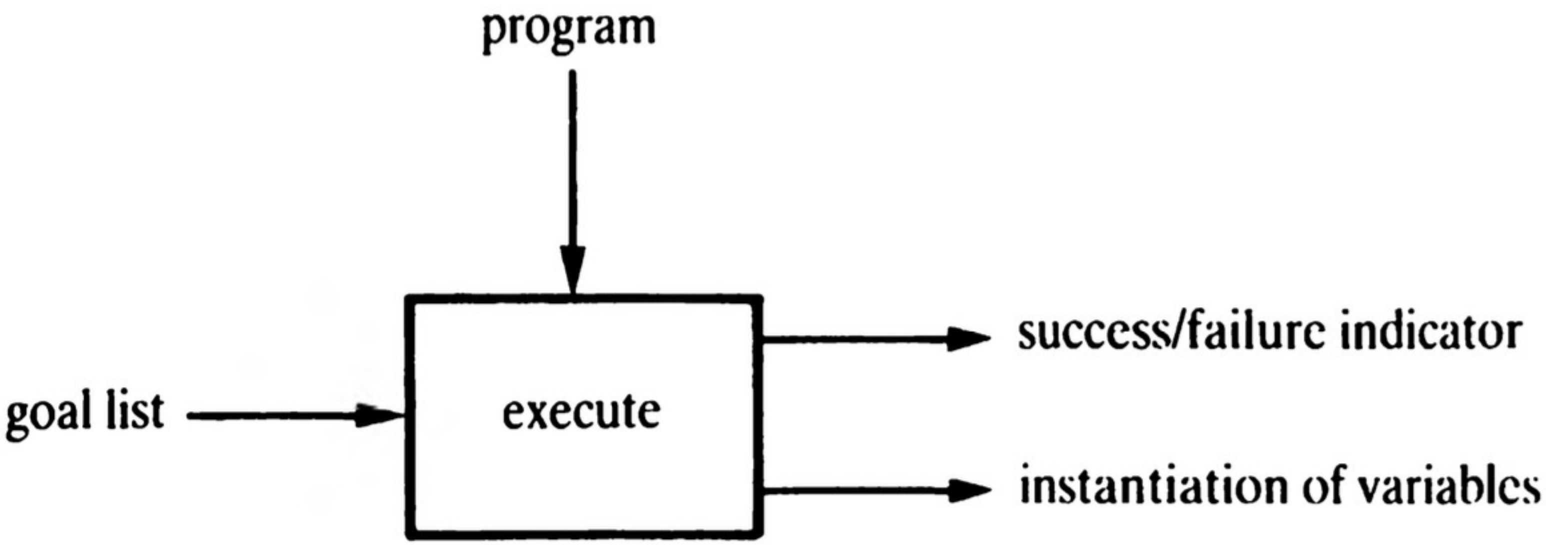
    (a) the head of I is identical to G, and

    (b) all the goals in the body of I are true.

Procedural meaning of Prolog programs

The procedural meaning specifies *how* Prolog answers questions.

To answer a questions means to try to satisfy a list of goals. Thus the porcedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program. To «execute goals» means: try to satisfy them.



Infinite loop

Write a program with the following clause.

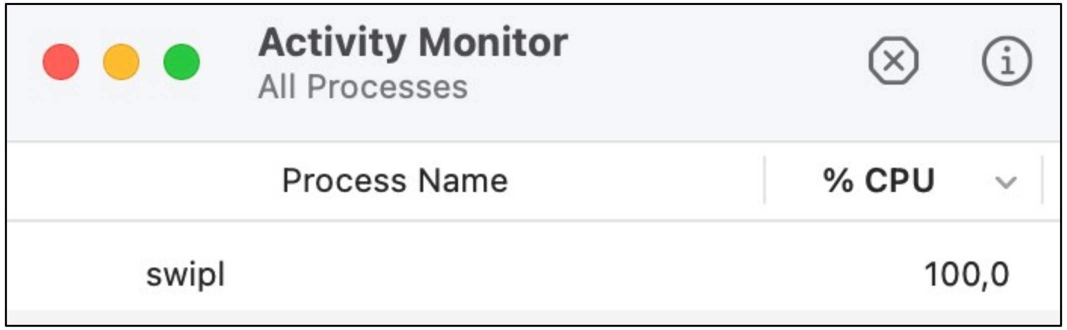
p :- p. “p is true if p is true” It is declaratively correct.

ask the following question.

?- p.

Which is the answer?

Open task manager or activity monitor



Infinite loop

Write a program with the following clause.

p :- p. “p is true if p is true” Is is procedurally useless

It results in an infinite loop

Infinite loops are not unusual in other programming languages.

A Prolog program may be declaratively correct, but at the same time be procedurally incorrect.

It may be not able to produce an answer to a question although the answer exists.

Prolog might choose a wrong path and the path could be infinite.

Order of clauses and goals

```
ancestor( X, Z) :-    % Rule a1: X is ancestor of Z
   parent( X, Z).

ancestor( X, Z) :-    % Rule a2: X is ancestor of Z
   parent(X, Y),
   ancestor( Y, Z).
```

Order of clauses and goals

```
ancestor( X, Z) :-    % Rule a1: X is ancestor of Z
   ancestor( Y, Z),
   parent(X, Y).

ancestor( X, Z) :-    % Rule a2: X is ancestor of Z
   parent( X, Z).
```

Which is the execution trace of **ancestor(tom, pat)**?

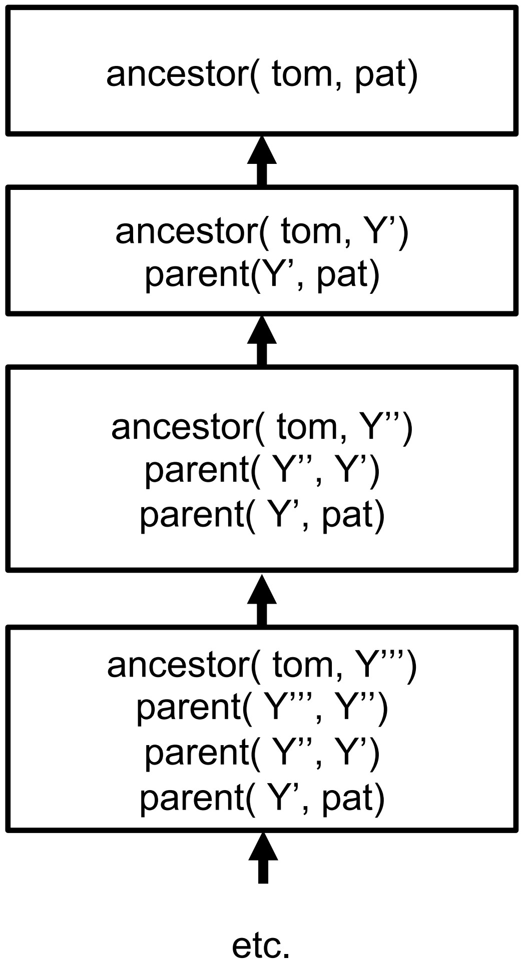
Try to graphically design the execution trace.

Order of clauses and goals

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

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

ancestor(tom, pat) execution trace.



Order of clauses and goals

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

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

What if we ask ancestor(tom, pat) to Prolog?

?- ancestor(tom, pat)

```
?- ancestor(tom, pat).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 48.4Mb, trail: 0Kb
ERROR:   Stack depth: 6,340,018, last-call: 0%, Choice points: 6,340,011
ERROR:   Probable infinite recursion (cycle):
ERROR:     [6,340,018] user:ancestor(_12694350, pat)
ERROR:     [6,340,017] user:ancestor(_12694370, pat)
   Exception: (6,340,017) ancestor(_12694282, pat) ?
```

A general heuristic in problem solving

It is usually best to try the simplest idea first.

1. the simpler idea is to check whether the two arguments of the **ancestor** relation satisfy the **parent** relation;

```
ancestor( X, Z) :-    % Rule a1: X is ancestor of Z
  parent( X, Z).
```

2. the more complicated idea is to find somebody "between" both people (somebody who is related to them by the **parent** and **ancestor** relations).

```
ancestor( X, Z) :-    % Rule a2: X is ancestor of Z
  parent(X, Y),
  ancestor( Y, Z).
```

Representation of lists

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

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

[ann, tennis, tom, skiing]

Remember: all structured objects in Prolog are trees. Lists are no exception to this.

Representation of lists

A list can be empty or non-empty.

Empty: **[ ]** .

Non-Empty:

1. the first items is called the *head* of the list;
2. the remaining part of the list is called the *tail*.

$$\textbf{[}\ \underbrace{\textbf{ann,}}_{\text{Head}}\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\ \textbf{]}$$

Representation of lists

$$\textbf{[}\ \underbrace{\textbf{ann,}}_{\text{Head}}\ \underbrace{\textbf{tennis, tom, skiing}}_{\text{Tail}}\ \textbf{]}$$

The tail has to be a list. The head and the tail are combined into a structure by the functor “.”:

.(Head, Tail)

Since **Tails** is in turn a list, it is either empty or it has its own head and tail

Representation of lists

[ann, tennis, tom, skiing]

Head Tail

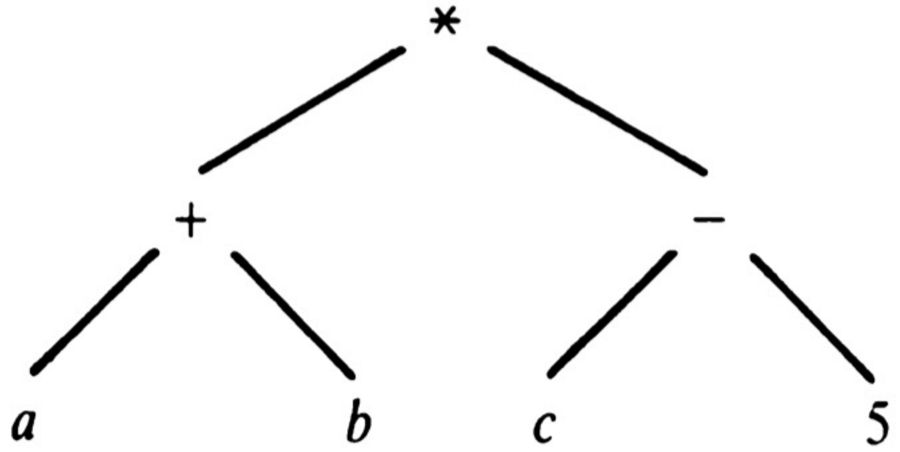
How Prolog interprets a List

.(Head, Tail)

Which is the graphical representation of the tree structure?

Remember the examples we made:

***(+(a, b),-(c,5))**



Representation of lists

[ann, tennis, tom, skiing]

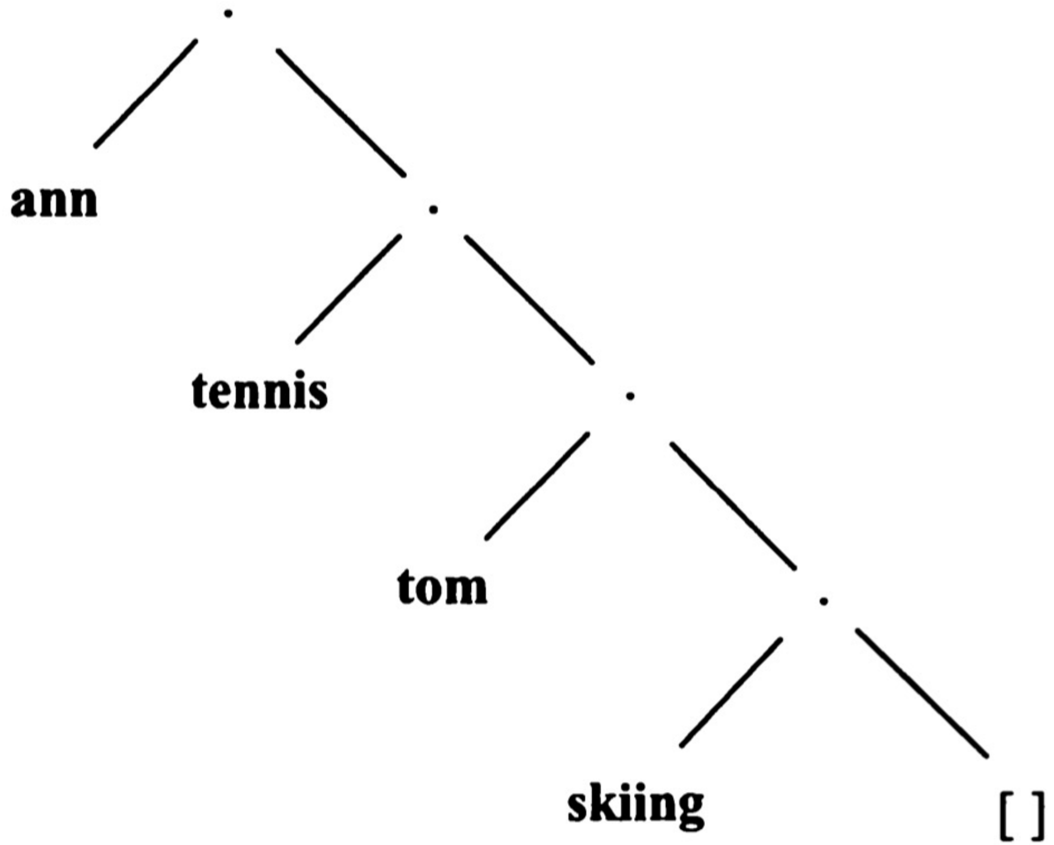
Head Tail

.(ann, .(tennis, .(tom, .(skiing, [ ]))))

Empty list

How Prolog interprets a List

.(Head, Tail)

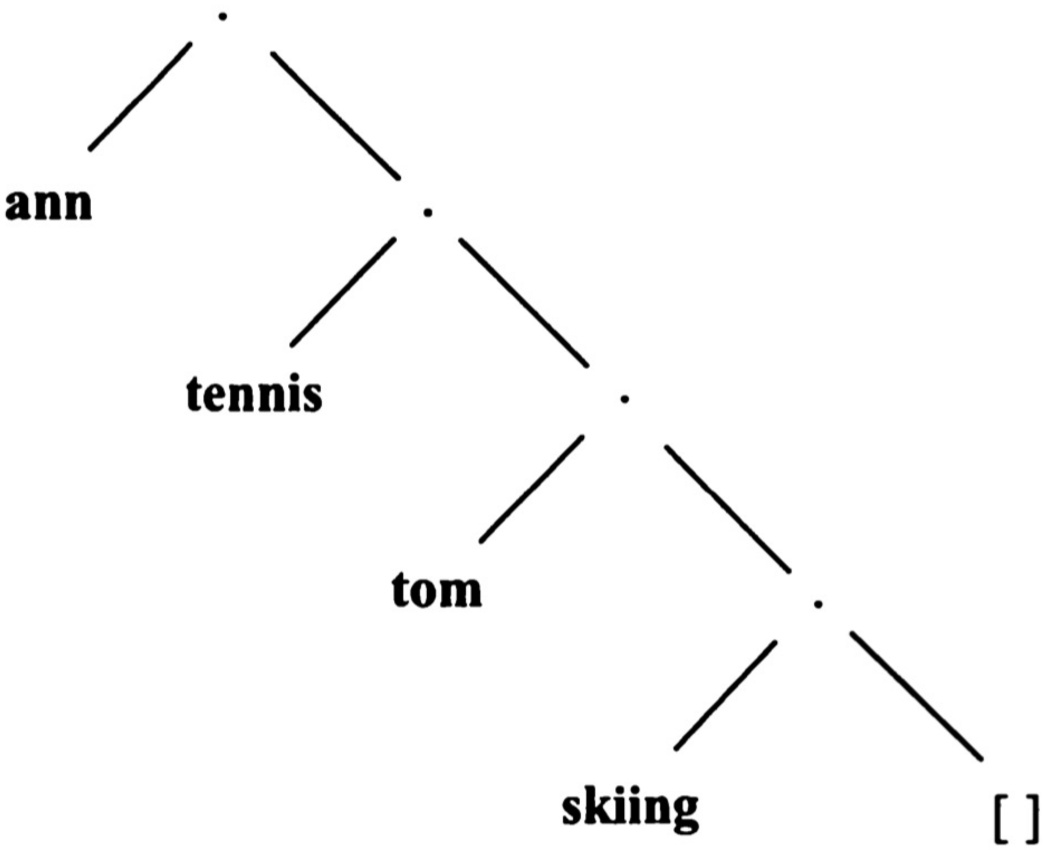


Representation of lists

Which one do you prefer?

[ann, tennis, tom, skiing]

.(ann, .(tennis, .(tom, .(skiing, [ ]))))



Operations on lists

Checking whether some object is an element of a list

Concatenation of two lists, obtaining a third one

Adding a new object to a list, or deleting an object from it.

Membership

Let us 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 if X occurs in L.

Some examples:

member(b, [a,b,c]) is true

member(b, [a,[b,c]]) is true

member([b,c], [a,[b,c]]) is true

Membership

The program for the membership relation can be based on the 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.

This can be written in two clauses:

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

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

Membership

member(X, [X | Tail]).

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

Let us aks whether b is in [a,b,c]

?- member(b, [a, b, c]). % Check whether b is in [a,b,c]

What if we ask the following? What do we obtain?

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

We can generate through backtracking all the members of a given list

Membership

We may also reverse the question: Find lists that contain a given item, e.g., “apple”

```
?- member( apple, L).

L = [ apple | _A];            % Any list that has "apple" as the head
L = [ _A, apple | _B];        %First item is anything, second is "apple"
L = [ _A, _B, apple | _C];
```

Membership

Find lists that contain **a**, **b**, and **c**:

?- member(a, L), member(b, L), member(c, L).

L = [a, b, c | _A];

L = [a, b, _A, c | _B];

L = [a, b, _A, _B, c | _C];

L = [a, b, _A, _B, _C, c | _D];

List with any length

Membership

Permutations of **a**, **b**, and **c**. % L is any list with exactly three elements

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

L = [a, b, c];

L = [a, c, b];

L = [b, a, c];

L = [c, a, b];

L = [b, c, a];

L = [c, b, a];

false.

Concatenation

For concatenating lists we will define the relation:

conc(L1, L2, L3)

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

conc([a,b], [c,d], [a,b,c,d])

is true, but

conc([a,b], [c,d], [a,b,a,c,d])

is false.

Concatenation

Let us define the concatenation relation.

1. If the first argument is the empty list then the second and the third arguments must be the same list (call it L); this is expressed by the following Prolog fact:

conc([ ], L, L).

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

[X | L1]

Concatenation

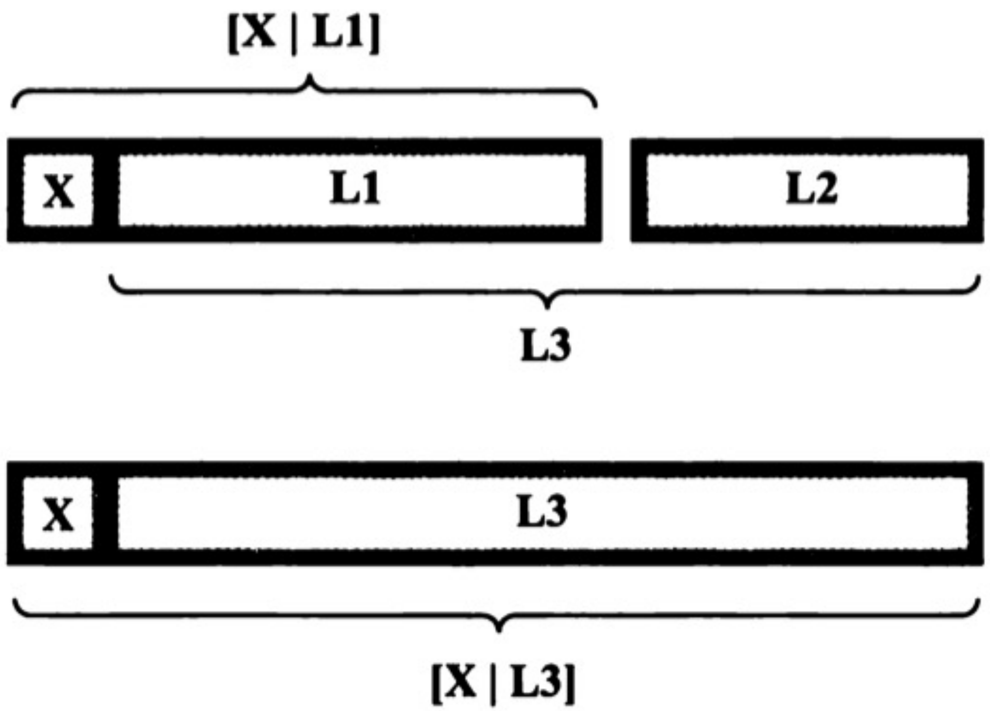
Concatenation of [X | L1] and some list L2. The result is the list [X | L3] where L3 is the concatenation of L1 and L2.

In Prolog:

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

?- 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]



Concatenation

In which way can we decompose the list [a,b,c] into two lists?

We can use **conc** in the inverse direction for *decomposing* a given list into two lists

?- conc(L1, L2, [a,b,c]).

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

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

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

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

false

Pattern in a List

We can also look for certain pattern in a list.

```
?- 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].

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

Month1 = apr
Month2 = jun
```

Pattern in a List

What are we requesting?

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

Pattern in a List

Delete from L1, everything that follows three successive occurrences of **z**

?- 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]

Concatenation

How can we improve the implementation of the membership relation?

We defined it in this way:

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

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

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

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

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

X is a member of list L if L can be decomposed into two lists so that the second one has X as its head.

```
member2( X, L) :-
  conc( _, [X |_], L).
```

member(apple, [peach,ananas,apple,mango,lemon]).

Execution Traces

We defined it in this way:

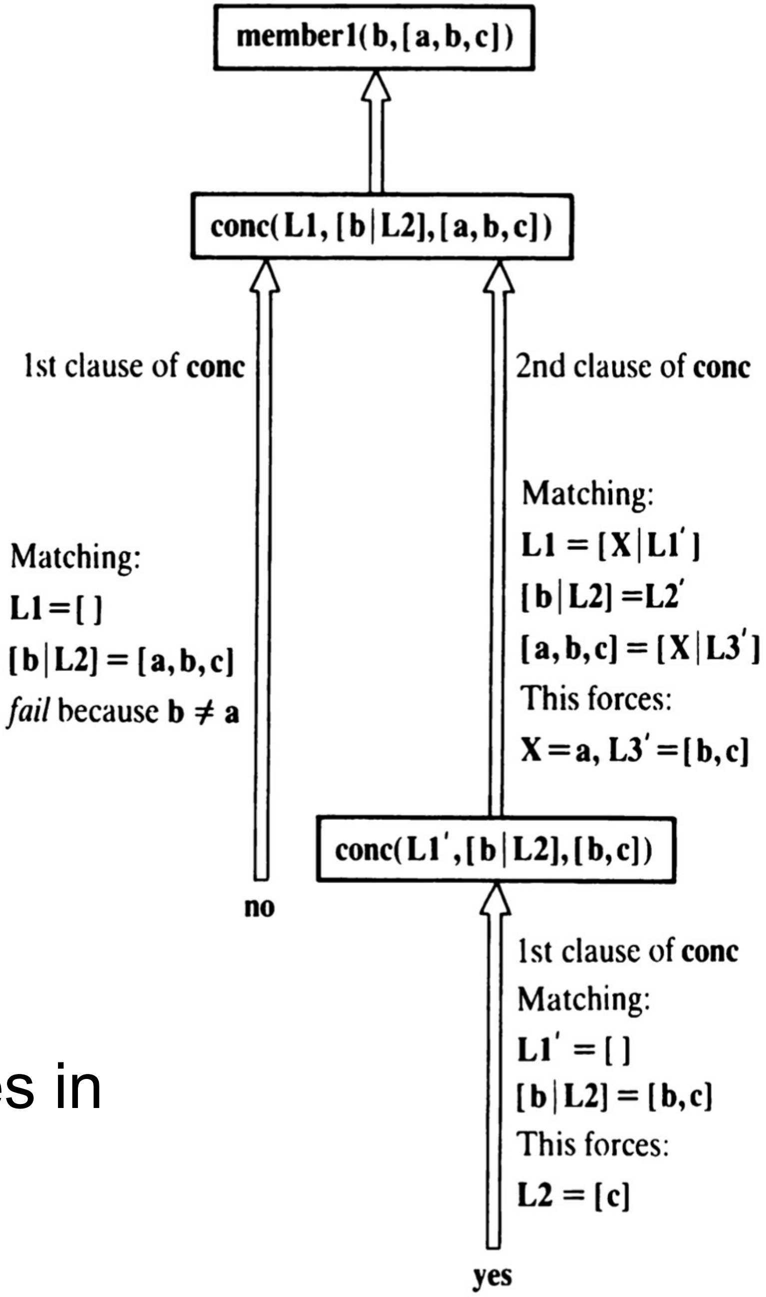
Which is the execution trace of member1(b, [a,b,c])?

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

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

Concatentation is provided as built-in predicates in Prolog systems under the name **append**.

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



Other Operations on Lists

Deleting an item, X, from a list, L.

del(X, L, L1)

where L1 is equal to L with the item X removed.

It can be defined similarly to the membership relation.

1. If X is the head of the list then the result after the 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).
```

Other Operations on Lists

```
del( X, [X | Tail], Tail).

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

?- del(a, [a,b,a,a], L).

L = [b, a, a] ;
L = [a, b, a] ;
L = [a, b, a] ;

Can we use **del** to insert **a** in the list **[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] ;

Other Operations on Lists

Generally inserting X at any place in some List giving BiggerList can be defined by the clause:

```
insert( X, List, BiggerList) :-
  del( X, BiggerList, List).
```

In **member1** we implemeneted the membership relation by using **conc**.

We can also use **del** to test for membership. The idea is simple: some X is a member of **List** if X can be deleted from **List**:

```
member2( X, List) :-
del( X, List, _).
```

Sublist

Let us define the sublist relation such that:

sublist([c,d,e], [a,b,c,d,e,f]) is true

sublist([c,e], [a,b,c,d,e,f]) is false

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.

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

What if we ask **sublist(S, [a,b,c])**?

Permutation

We can generate permutations of a list through backtracking using the **permutation** procedure, as in the following:

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

```
P = [a, b, c] ;
P = [a, c, b] ;
P = [b, a, c] ;
P = [b, c, a] ;
P = [c, a, b] ;
P = [c, b, a] ;
false.
```

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

List length

Let us count the elements in a list **List** and instantiate N to thein number.

1. If the list is empty then its length is 0
2. If the list is not empty then **List = [Head | Tail]**; then its length is equal to 1 plus the length of the tail **Tail**

```
length( [ ], 0).

length([_ | Tail], N) :-
   length( Tails, N1),
   N is 1 + N1.
```

```
?- length([ a,b, [ c,d], e], N).

N = 4
```

```
ERROR: /Users/fabriziofornari/Desktop/LCP - Fabrizio Fornari/SWI-Prolog/PROGRAMS/list8.pl:33:
ERROR:    No permission to modify static procedure `length/2'
ERROR:    Defined at /Applications/SWI-Prolog.app/Contents/swipl/boot/init.pl:4056
```

SWI-Prolog library

https://www.swi-prolog.org/pldoc/man?section=libpl