UCS1524 – Logic Programming

List in Prolog

Session Objectives

- Understanding list concept in Prolog

Session Outcomes

- At the end of this session, participants will be able to
  - Develop Prolog programs using list and list operations

Agenda

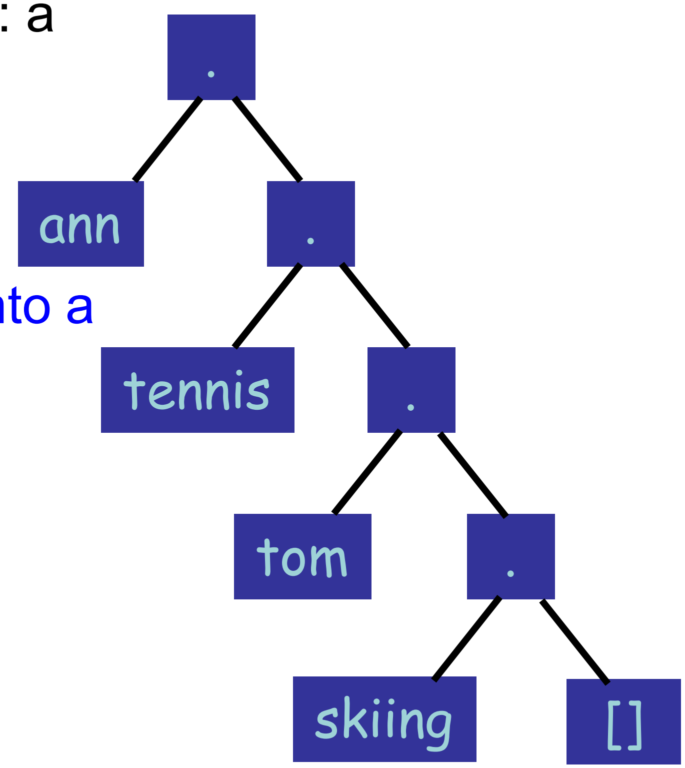
- Representation of list
- Some operations on lists

Representation of list

- A list is a sequence of any number of items.
- For example:
  - [ann, tennis, tom, skiing]
- A list is either empty or non-empty.
  - Empty: []
  - Non-empty:
    - The first term, called the head of the list
    - The remaining part of the list, called the tail
    - Example: [ann, tennis, tom, skiing]
      - Head: ann
      - Tail: [tennis, tom, skiing]

Representation of list

- In general,
  - the head can be anything (for example: a tree or a variable)
  - the tail has to be a list



- The head and the tail are then combined into a structure by a special functor

  .(head, Tail)

  - For example:

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

    L = [ann, tennis, tom, skiing].

    are the same in Prolog.

Representation of list

```
| ?- List1 = [a,b,c],
     List2 = .(a, .(b, .(c,[]))).

List1 = [a,b,c]
List2 = [a,b,c]

yes

| ?- Hobbies1 = .(tennis, .(music, [])),
     Hobbies2 = [skiing, food],
     L = [ann, Hobbies1, tom, Hobbies2].

Hobbies1 = [tennis,music]
Hobbies2 = [skiing,food]
L = [ann,[tennis,music],tom,[skiing,food]]

yes
```

```
| ?- L= [a|Tail].

L = [a|Tail]

yes

| ?- [a|Z] = .(X, .(Y, [])).

X = a
Z = [Y]

yes

| ?- [a|[b]] = .(X, .(Y, [])).

X = a
Y = b

yes
```

Representation of list

- Summarize:
  - A list is a data structure that is either empty or consists of two parts: a head and a tail.
  - The tail itself has to be a list.
  - List are handled in Prolog as a special case of binary trees.

  - Prolog accept lists written as:
    - [Item1, Item2,…]
    - [Head | Tail]
    - [Item1, Item2, …| Other]

Some operations on lists

- The most common operations on lists are:
  - Checking whether some object is an element of a list, which corresponds to checking for the set membership;
  - Concatenation of two lists, obtaining a third list, which may correspond to the union of sets;
  - Adding a new object to a list, or deleting some object form it.

Membership

- The membership relation:

  member(X, L)

  where X is an object and L is list.

  - The goal **member(X, L)** is true if X occurs in L.

  - For example:

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

    member(b, [a, [b, c]]) is not true

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

Membership

- 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.

member1(X, [X| Tail]).
member1(X, [Head| Tail]) :- member1(X, Tail).

```
| ?- member1(3, [1,2,3,4]).
     1    1  Call: member1(3, [1,2,3,4]) ?
     2    2  Call: member1(3, [2,3,4]) ?
     3    3  Call: member1(3, [3,4]) ?
     3    3  Exit: member1(3, [3,4]) ?
     2    2  Exit: member1(3, [2,3,4]) ?
     1    1  Exit: member1(3, [1,2,3,4]) ?
true ?
Yes
```

Concatenation

- The concatenation relation:

  conc(L1, L2, L3)

  here 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, a, c, d]) is not true

Concatenation

- Two case of concatenation relation:

  (1) If the first argument is the empty list then the second and the third arguments must be the same list.

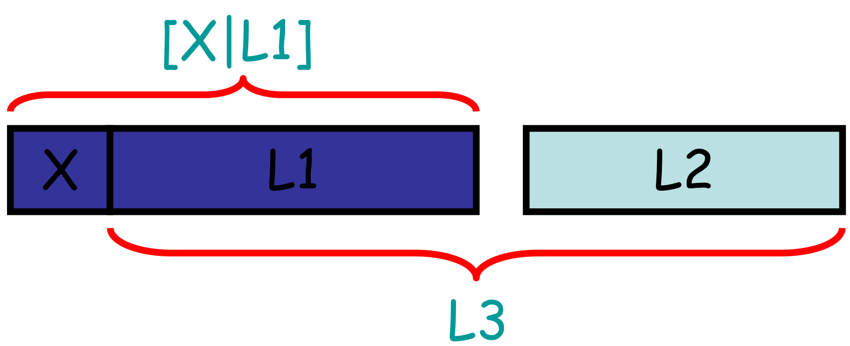
  conc([], L, L).

  (2) If the first argument is an 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.

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



Concatenation

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

```
| ?- conc( [a, b], [c, d], A).
     1    1  Call: conc([a,b],[c,d],_31) ?
     2    2  Call: conc([b],[c,d],_64) ?
     3    3  Call: conc([],[c,d],_91) ?
     3    3  Exit: conc([],[c,d],[c,d]) ?
     2    2  Exit: conc([b],[c,d],[b,c,d]) ?
     1    1  Exit: conc([a,b],[c,d],[a,b,c,d]) ?
A = [a,b,c,d]
yes
```

Concatenation

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

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

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

```
| ?- 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 = [] ? ;

no
```

Concatenation

```
| ?- conc( Before, [may| After], [jan, feb, mar, apr, may, jun, jul, aug, sep, oct,
     nov, dec]).
After = [jun,jul,aug,sep,oct,nov,dec]
Before = [jan,feb,mar,apr] ? ;
no

| ?- conc( _, [Month1,may, Month2|_], [jan, feb, mar, apr, may, jun, jul, aug, sep,
     oct, nov, dec]).
Month1 = apr
Month2 = jun ? ;
No

| ?- 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] ? ;
no
```

Concatenation

- Define the membership relation:

  member2(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).

```
| ?- member2(3, [1,2,3,4]).
      1    1  Call: member2(3,[1,2,3,4]) ?
      2    2  Call: conc(_58,[3|_57],[1,2,3,4]) ?
      3    3  Call: conc(_85,[3|_57],[2,3,4]) ?
      4    4  Call: conc(_112,[3|_57],[3,4]) ?
      4    4  Exit: conc([],[3,4],[3,4]) ?
      3    3  Exit: conc([2],[3,4],[2,3,4]) ?
      2    2  Exit: conc([1,2],[3,4],[1,2,3,4]) ?
      1    1  Exit: member2(3,[1,2,3,4]) ?
true ?
(15 ms) yes
```

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

  – Compare to the member relation defined on 3.2.1:

  member1(X, [X| Tail]).
  member1(X, [Head| Tail]) :- member1(X, Tail).

Adding an item

- To add an item to a list, it is easiest to put the new item in front of the list so that it become the new head.
- If X is the new item and the list to which X is added is L then the resulting list is simply:

  [X|L].
- So we actually need no procedure for adding a new element in front of the list.
- If we want to define such a procedure:

  add(X, L,[X|L]).

```
| ?- add(4, [1,2,3],Y).
     1    1  Call: add(4,[1,2,3],_29) ?
     1    1  Exit: add(4,[1,2,3],[4,1,2,3]) ?
Y = [4,1,2,3]
Yes
```

Deleting an item

- Deleting an item X form a list L can be programmed as a relation:

  del(X, L, L1)

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

- Two cases of delete 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).
```

Deleting an item

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

| ?- del(4, [1,2,3,4,5,6],Y).
     1    1  Call: del(4,[1,2,3,4,5,6],_35) ?
     2    2  Call: del(4,[2,3,4,5,6],_68) ?
     3    3  Call: del(4,[3,4,5,6],_95) ?
     4    4  Call: del(4,[4,5,6],_122) ?
     4    4  Exit: del(4,[4,5,6],[5,6]) ?
     3    3  Exit: del(4,[3,4,5,6],[3,5,6]) ?
     2    2  Exit: del(4,[2,3,4,5,6],[2,3,5,6]) ?
     1    1  Exit: del(4,[1,2,3,4,5,6],[1,2,3,5,6]) ?
Y = [1,2,3,5,6] ?
(31 ms) yes
```

Deleting an item

- Like **member, del** is also non-deterministic.

  ```
  | ?- del(a,[a,b,a,a],L).
  L = [b,a,a] ? ;
  L = [a,b,a] ? ;
  L = [a,b,a] ? ;
  (47 ms) no
  ```

- **del** can also be used in the inverse direction, to add an item to a list by inserting the new item anywhere in the list.

  ```
  | ?- 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] ? ;
  (16 ms) no
  ```

Deleting an item

- Two applications:
  - Inserting X at any place in some list **List** giving **BiggerList** can be defined:

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

  - Use **del** to test for membership:

```
member3( X, List) :- del( X, List, _).
```

Sublist

- The sublist relation:
  - This relation has two arguments, a list L and a list S such that S occurs within L as its sublist.

    For example:

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

    sublist([c, e], [a, b, c, d, e, f]) is not true

  - S is a sublist of L if

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

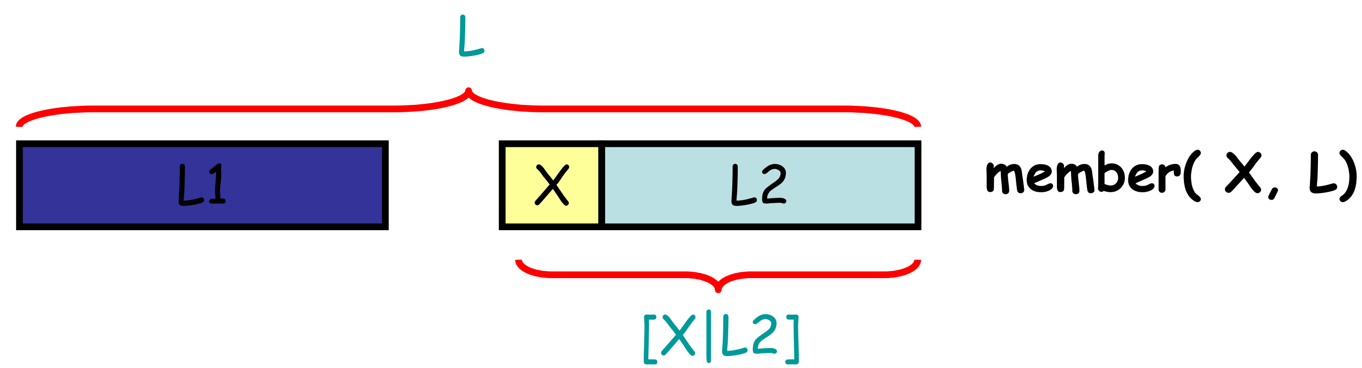
    (2) L2 can be decomposed into two lists, S and some L3.

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

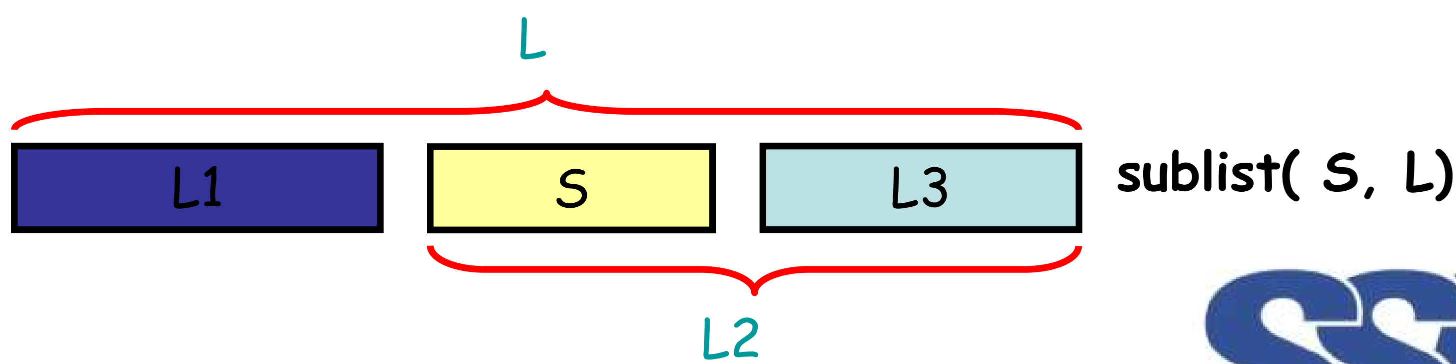
Sublist

- Compare to **member** relation:

member2(X, L):- conc(L1,[X|L2],L).



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



Sublist

- An example:

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

S = [a,b,c] ? ;
S = [b,c] ? ;
S = [c] ? ;
S = [] ? ;
S = [b] ? ;
S = [a,c] ? ;
S = [a] ? ;
S = [a,b] ? ;
(31 ms) no
```

The power set of [a, b, c]

Exercise:
Please show L1, L2 and L3 in each case.

Permutations

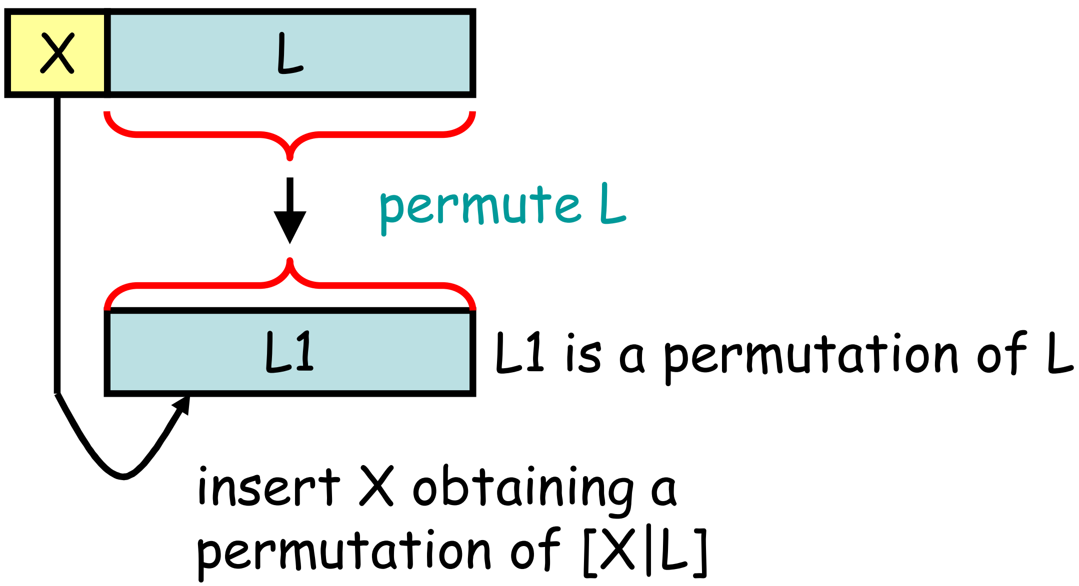
- Permutation examples:

```
| ?- 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] ? ;
(31 ms) no
```

```
| ?- permutation(P, [1,2,3]).

P = [1,2,3] ? ;
Stack overflow…
```

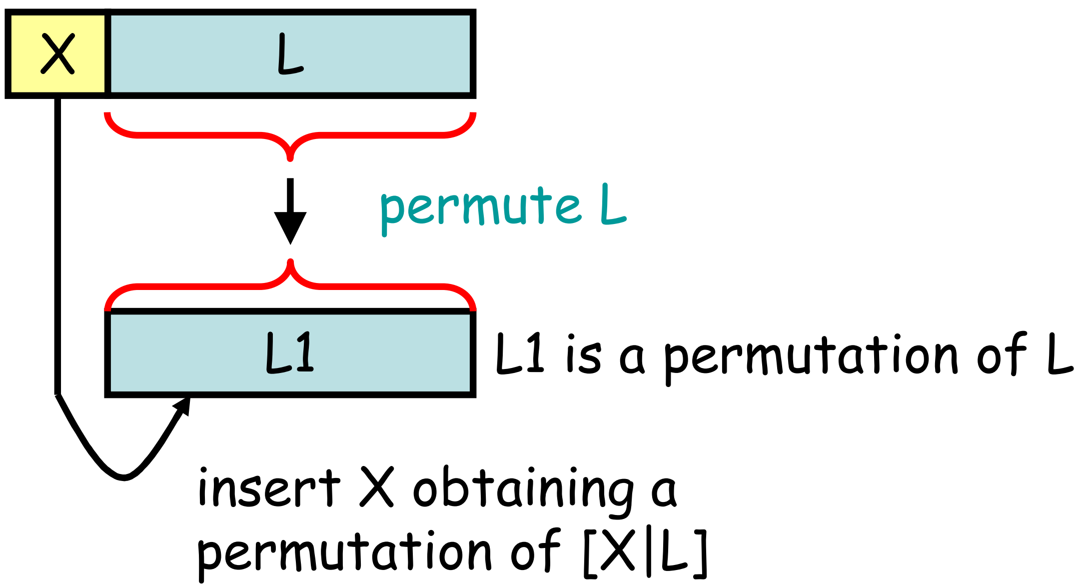
Permutations

- Two cases of permutation relation:
  - If the first list is empty then the second list must also be empty.
  - If the first list is not empty then it has the form [X|L], and a permutation of such a list can be constructed as shown in Fig. 3.15: first permute L obtaining L1 and then insert X at any position into L1.



Permutations

```
permutation1([],[]).
permutation1([ X| L], P):-
        permutation1( L, L1), insert( X, L1, P).

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



Permutations

- Another definition of permutation relation:

```
permutation2([],[]).
permutation2(L, [ X| P]):-
        del( X, L, L1), permutation2(L1, P).
```

  – To delete an element X from the first list, permute the rest of it obtaining a list P, and add X in front of P.

Permutations

- Examples:

```
| ?- permutation2([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] ? ;
no
```

```
| ?- permutation( L, [a, b, c]).
```

(1) Apply **permutation1**: The program will instantiate L successfully to all six permutations, and then get into an infinite loop.

(2) Apply **permutation2**: The program will find only the first permutation and then get into an infinite loop.

Summary

- Representation of list
- Some operations on lists
  - Membership
  - Concatenation
  - Add
  - Delete
  - Sublist
  - Permutation

Check your understanding

- Write a goal, using **conc**, to delete the last three elements from a list L producing another list L1.

- Write a goal to delete the first three elements and the last three elements from a list L producing list L2.

Check your understanding

- Define the relation

  last1(Item, List)

  so that **Item** is the last element of a list **List**.

  Write two versions:

  - Using the **conc** relation
  - Without **conc**

- Redefine the concatenation relation:

  conc1(L1, L2, L3)

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

Check your understanding

- Define a relation **add_end(X, L, L1)** to add an item X to the end of list L.

  For example,

  add_end(a, [b,c,d], L1) → L1 = [b, c, d, a].

```
add_end(I, L, L1) :- conc(L, [I], L1).
```

Check your understanding

- Define a relation **del_all(X, L, L1)** to remove all items X (if any) from list L.

  For example,

  del_all(a, [a,b,a,c,d,a], L1) → L1 = [b, c, d].

Check your understanding

- Define the relation

  reverse1(List, ReversedList) that reverses lists.

  For example,

  reverse1([a, b, c, d],L). ➔ L= [d, c, b, a]

```
reverse1([], [] ).
reverse1( [First | Rest], Reversed) :-
reverse1( Rest, ReversedRest), conc(ReversedRest, [First], Reversed).
```

Check your understanding

- Define the predicate **palindrome(List).**
  - A list is a palindrome if it reads the same in the forward and in the backward direction.
  - For example, [m,a,d,a,m] ➔ true.