List utility predicates

member (X , L)

◊ Item X is a member of the list L.

Reduce the list – second rule –
until first in list – first rule.
or empty – no rule so fail –

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

◊ Note the use of the anonymous variable _

» **We do not care about the value of the rest in the first rule, nor the value of first in the second rule**

» **Typically use it when it is the only instance of that variable in the rule**

append (L1, L2 , R)

◊ R is the result of appending list L2 to the end of list L1.

append ([ ] , L , L).

– **Appending to nil yields the original list.**

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

> **Simultaneous recursive descent on L1 & L3 first of the left list is the first of the result.**

Pattern

L1 = a b c **L2 = 2 3 4 5** **L3 = a b c 2 3 4 5**

= [a | [b, c]] **= [a | [b, c, 2, 3, 4, 5]]**

append (L1 , L2 , R) – 2

◊ Queries – ask for results in all combinations. Not like Java or C where functions are programmed for only one query

append ([1 , 2 , 3] , [a , b , c] , R).

> **What is the result of appending L1 and L2?**

append (L1 , [a , b , c] , [1 , 2 , 3 , a , b , c]).

> **What L1 gives [1 , 2 , 3 , a , b , c] when appended with [a , b , c] ?**

append ([1 , 2 , 3] , L2 , [1 , 2 , 3 , a , b , c]).

> **What L2 gives [1 , 2 , 3 , a , b , c] when appended to [1 , 2 , 3] ?**

append (L1 , L2 , R) – 3

append (L1 , L2 , [1 , 2 , 3 , a , b , c]).

> **What L1 and L2 gives [1 , 2 , 3 , a , b , c] when L2 is appended to L1?**

append (L1 , L2 , R).

> **What L1 and L2 give R? Infinite number of answers**

append (Before , [Middle | After] , List).

> **If middle is defined we can get the before and after**
>
> **append (Before , [4 | After] , [1,2,3,4,5,6,7]).**

Last predicate defined using append

◊ Define the predicate **Last (Item , List)** that asserts Item is the last element of the list **List**.

```
Last ( Item , List ) :- append ( _ , [ Item ] , List ) .
```

Shift predicate using append

◊ Define the predicate **shift (List , Shifted)** that asserts Shifted is the **List** rotated by one element to the left.

```
shift ( [ Head | Tail ] , Shifted ) :-
        append ( Tail , [ Head ] , Shifted ) .
```

Reverse predicate using append

- Define the predicate **reverse (List , ReversedList)** that asserts **ReversedList** is the **List** in reverse order.

```
reverse ( [ ] , [ ] ) .
reverse ( [Head | Tail ] , Reversed ) :-
      reverse ( Tail , ReversedTail ) ,
      append ( ReversedTail , [ Head ] , Reversed ) .
```

Trace – append (P, [a] , [1 , 2 , 3 , a])

- Variables are renamed every time a rule is used for matching

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

- Try to match rule 1

  P = []   [a] = L_1   [1,2,3,a] = L_1

  From query = From rule

- 1 – Fail, try to match rule 2

  P = [X_2 | L1_2]   [a] = L2_2   [1,2,3,a] = [X_2 | L3_2]

  » **Succeed with X_2 = 1   L2_2 = [a]   L3_2 = [2,3,a]**

Trace – append (P, [a] , [1 , 2 , 3 , a]) – 2

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

- ◊ Try to match rule 1 **append(L1_2, [a], [2,3,a])**
  L1_2 = [] [a] = L_3 [2,3,a] = L_3
- ◊ 2 – Fail, try to match rule 2
  L1_2 = [X_4 | L1_4] [a] = L2_4 [2,3,a] = [X_4 | L3_4]
  - » **Succeed with X_4 = 2 L2_4 = [a] L3_4 = [3,a]**
- ◊ Try to match rule 1 **append(L1_4, [a], [3,a])**
  L1_4 = [] [a] = L_5 [3,a] = L_5

Trace – append (P, [a] , [1 , 2 , 3 , a]) – 3

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

- ◊ 3 – Fail, try to match rule 2
  L1_4 = [X_6 | L1_6]   [a] = L2_6   [3,a] = [X_6 | L3_6]
  - » **Succeed with   X_6 = 3   L2_6 = [a]   L3_6 = [a]**
- ◊ Try to match rule 1   **append(L1_6, [a], [a])**
  L1_6 = []   [a] = L_7   [a] = L_7
- ◊ Succeed, recursion stops, backtrack and substitute values

Trace – append (P, [a] , [1 , 2 , 3 , a]) – 4

- In step 3
  - **L1 _4 = [3 | []] = [3]**
- In step 2 we had
  - **L1_2 = [X_4 | L1_4] L2_4 = [a] [2,3,a] = [X_4 | L3_4]**
  - » **Succeed with X_4 = 2 L2_4 = [a] L3_4 = [3,a]**
  - » **and from Step 3 L1_4 = [3]**
  - » **Thus L1_2 = [2, 3]**
- In step 1 we had
  - **P = [X_2 | L1_2] [a] = L2_2 [a,1,2,3] = [X_2 | L3_2]**
  - » **Succeed with X_2 = 1 L2_2 = [a] L3_2 = [2,3,a]**
  - » **and from Step 2 L1_2 = [2, 3]**
  - » **Thus P = [1, 2, 3]**

delete (X , L , R)

◊ R is the result of deleting item X from the list L.

Remove if first in the list.

```
delete ( X , [ X | R ] , R ).
```

If not the first then remove from the next smaller sublist.

```
delete ( X , [ Y | L ] , [ Y | R ] )  :-  delete ( X , L , R )
```

The SWI Prolog built-in predicate delete does not work as the above definition. Arguments are in a different order and have different meaning.

prefix (P , L)

◊ **P** is a prefix of the list **L**. It can be defined using append as follows.

prefix (P , L) :- append (P , _ , L).

> **P is a prefix of L if something, including nil, can be suffixed to P to form L.**

prefix (P , L) – 2

◊ We can define prefix in terms of itself as follows.

```
List    PPPPPPXXXXX   ==>   XXXXX
Prefix  YYYYYY         -    Empty
        ^^^^^^
```

Check equality until Prefix is exhausted.

◊ The base case is having the empty list as the prefix.

prefix ([ ] , _).

◊ The recursive case is having the first items on the prefix and the list being the same and the reduced prefix and list satisfy the prefix property.

prefix ([A | B] , [A | C]) :- prefix (B , C).

suffix (S , L)

◊ **S** is a suffix of the list **L**.   It can be defined using append as follows.

```
suffix ( S , L )  :-  append ( _ , S , L ).
```

> **S is a suffix of L if something, including nil, can be prefixed to S to form L.**

suffix (S , L) – 2

◊ We can define suffix in terms of itself as follows.

```
List     PPPPPPXXXXX   ==>   XXXXX
Suffix         YYYYY         YYYYY
         ^^^^^^
```

Reduce the prefix part of the List.

◊ In the base case the suffix is the list.

suffix (L , L).

◊ The recursive case is to reduce the size of the prefix of the list.

suffix (S , [_ | L]) :- suffix (S , L).

sublist (S , L)

◊ S is a sublist of L can be defined using append as follows.

```
sublist ( S , L )  :-  append ( _ , S , Lt ) ,
                       append ( Lt , _ , L ).
```

> **S is a sublist of L if something, including nil, can be prefixed to S to form the list Lt**

> **And something, including nil, can be suffixed to Lt to form L.**

◊ In other words, **S** is a sublist of **L** if there exists a prefix **P** to **S** and a suffix **T** to **S** such that **L = P || S || T**

> **where || means concatenate.**

sublist(S,L)

◊ We can define sublist in terms of itself and prefix as follows.

```
List      PPPPSSSSSXXXXXX   ==>   SSSSSXXXXXX
Sublist       YYYYY               YYYYY
        ^^^^
```

Reduce the prefix part of the List.

◊ In the base case the sublist is the prefix of the list.

sublist (S , L) :- prefix (S , L).

◊ The recursive case is to reduce the size of the prefix of the list.

sublist (S , [_ | L]) :- sublist (S , L).

removeAllTop (Item, List, Result)

◊ Asserts that **Result** is **List** with all occurrences of **Item** removed from the top level of **List** .

```
removeAllTop (_, [], []).

removeAllTop (Item, [Item | Lt], R) :-
        removeAllTop (Item, Lt, R).

removeAllTop (Item, [H | Lt], [H | Rt]) :-
        Item \= H ,
        removeAllTop (Item, Lt, Rt).
```

removeAll (Item, List, Result)

◊ Asserts that **Result** is **List** with all occurrences of **Item** removed from all levels of **List** .

```
removeAll (_, [], []).

removeAll (Item, [Item | Lt], R) :-
        removeAll (Item, Lt, R).

removeAll (Item, [H | Lt], [H | Rt]) :-
        Item \= H , H \= [_|_] ,
        removeAll (Item, Lt, Rt).

removeAll (Item, [Lh | Lt], [Rh | Rt]) :-
        Item \= Lh , Lh = [_|_] ,
        removeAll (Item, Lh, Rh)
        removeAll (Item, Lt, Rt).
```