PROLOG Programming: List

The list is a simple data structure that is widely used in non-numeric programming. List consists of any number of items, for example, red, green, blue, white, dark. It will be represented as, [red, green, blue, white, dark]. The list of elements will be enclosed with **square brackets**.

A list can be either **empty** or **non-empty**. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list consists of two things as given below −

- The first item, called the **head** of the list;
- The remaining part of the list, called the **tail**.

Suppose we have a list like: [red, green, blue, white, dark]. Here the head is red and tail is [green, blue, white, dark]. So the tail is another list.

Now, let us consider we have a list, L = [a, b, c]. If we write Tail = [b, c] then we can also write the list L as L = [a | Tail]. Here the vertical bar (|) separates the head and tail parts.

So the following list representations are also valid −

- [a, b, c] = [H | T]: H = a; T = [b, c]
- [a, b, c] = [X, Y | Z]: X = a, Y = b, Z = [c]
- [a, b, c] = [X, Y, Z | Tail]: X= a, Y = b, Z = c, Tail = []

For these properties we can define the list as −

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.

Length Calculation

This is used to find the length of list L. We will define one predicate to do this task. Suppose the predicate name is **list_length(L, N)**. This takes L and N as input argument. This will count the elements in a list L and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases −

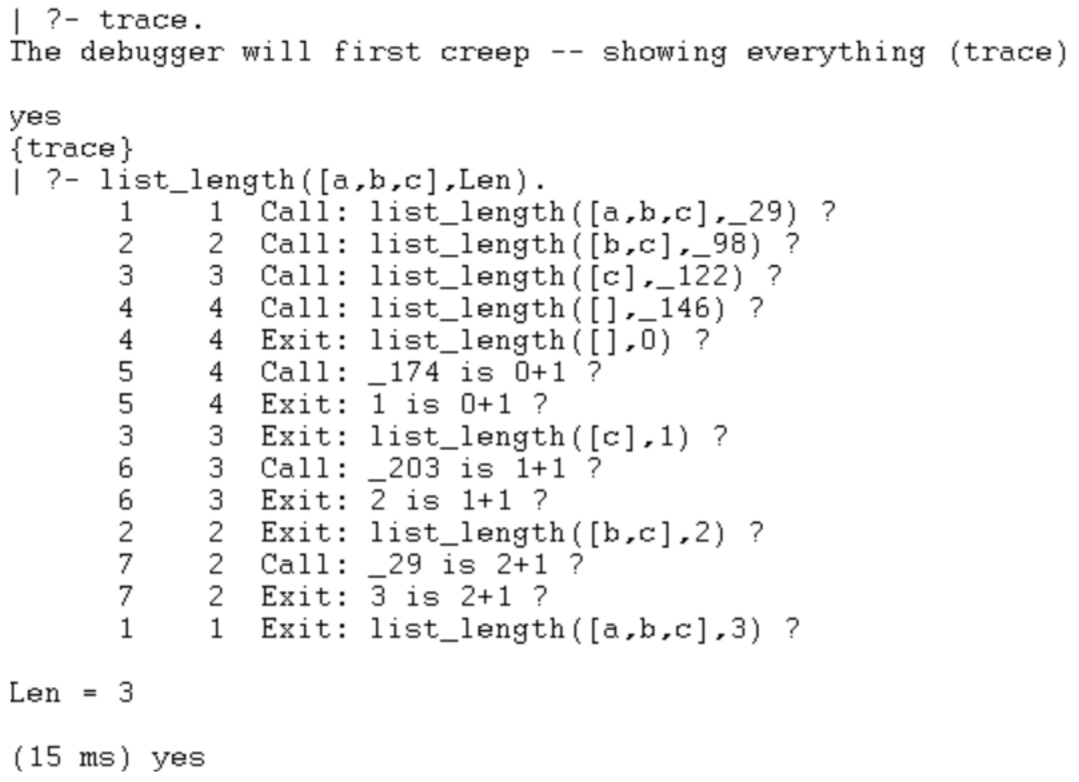
- If list is empty, then length is 0.
- If the list is not empty, then L = [Head|Tail], then its length is 1 + length of Tail.

Program

```
list_length([],0).
list_length([H|TAIL],N) :- list_length(TAIL,N1), N is N1 + 1.
```

Output

```
| ?- list_length([a,b,c,d,e,f,g,h,i,j],Len).

Len = 10

yes
| ?- list_length([],Len).

Len = 0

yes
| ?- list_length([[a,b],[c,d],[e,f]],Len).

Len = 3

yes
```



```
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- list_length([a,b,c],Len).
      1    1  Call: list_length([a,b,c],_29) ?
      2    2  Call: list_length([b,c],_98) ?
      3    3  Call: list_length([c],_122) ?
      4    4  Call: list_length([],_146) ?
      4    4  Exit: list_length([],0) ?
      5    4  Call: _174 is 0+1 ?
      5    4  Exit: 1 is 0+1 ?
      3    3  Exit: list_length([c],1) ?
      6    3  Call: _203 is 1+1 ?
      6    3  Exit: 2 is 1+1 ?
      2    2  Exit: list_length([b,c],2) ?
      7    2  Call: _29 is 2+1 ?
      7    2  Exit: 3 is 2+1 ?
      1    1  Exit: list_length([a,b,c],3) ?

Len = 3

(15 ms) yes
```

Max Item Operation

This operation is used to find the maximum element from a list. We will define a predicate, **list_max_elem(List, Max)**, then this will find Max element from the list and return.

- If there is only one element, then it will be the max element.
- interpret the list as [X|Tail]. Now recursively find max of Tail and store it into MaxRest, and find maximum of X and MaxRest, to determine max element of the list.

Program

```
max_of_two(X,Y,X) :- X >= Y.
max_of_two(X,Y,Y) :- X < Y.
list_max_elem([X],X).
list_max_elem([X|Rest],Max) :-
   list_max_elem(Rest,MaxRest),
   max_of_two(X,MaxRest,Max).
```

Output

```
| ?- list_max_elem([8,5,3,4,7,9,6,1],Max).

Max = 9 ?

yes
| ?- list_max_elem([5,12,69,112,48,4],Max).

Max = 112 ?

Yes
```

List Sum Operation

In this example, we will define a clause, list_sum(List, Sum), this will return the sum of the elements of the list.

- If the list is empty, then sum will be 0.
- Represent list as [Head|Tail], find sum of tail recursively and store them into SumTemp, then set Sum = Head + SumTemp.

Program

```
list_sum([],0).
list_sum([Head|Tail], Sum) :-
   list_sum(Tail,SumTemp),
   Sum is Head + SumTemp.
```

Output

```
| ?- list_sum([5,12,69,112,48,4],Sum).

Sum = 250

yes
| ?- list_sum([8,5,3,4,7,9,6,1],Sum).

Sum = 43
```

```
yes
```

Membership Operation

During this operation, we can check whether a member X is present in list L or not? So how to check this? Well, we have to define one predicate to do so. Suppose the predicate name is **list_member(X,L)**. The goal of this predicate is to check whether X is present in L or not.

To design this predicate, we can follow these observations. X is a member of L if either −

- X is head of L, or
- X is a member of the tail of L

Program

```
list_member(X,[X|T]).
list_member(X,[H|TAIL]) :- list_member(X,TAIL).
```

Output

```
yes
| ?- list_member(b,[a,b,c]).

true ?

yes
| ?- list_member(b,[a,[b,c]]).

no
| ?- list_member([b,c],[a,[b,c]]).

true ?

yes
| ?- list_member(d,[a,b,c]).

no
| ?- list_member(d,[a,b,c]).
```

Concatenation

Concatenation of two lists means adding the list items of the second list after the first one. So if two lists are [a,b,c] and [1,2], then the final list will be [a,b,c,1,2]. So to do this task we will create one predicate called list_concat(), that will take first list L1, second list L2, and the L3 as resultant list. There are two observations here.

- If the first list is empty, and second list is L, then the resultant list will be L.
- If the first list is not empty, then write this as [Head|Tail], concatenate Tail with L2 recursively, and store into new list in the form, [Head|New List].

Program

```
list_concat([],L,L).
list_concat([X1|L1],L2,[X1|L3]) :- list_concat(L1,L2,L3).
```

Output

```
yes
| ?- list_concat([1,2],[a,b,c],NewList).

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

yes
| ?- list_concat([],[a,b,c],NewList).

NewList = [a,b,c]

yes
| ?- list_concat([[1,2,3],[p,q,r]],[a,b,c],NewList).

NewList = [[1,2,3],[p,q,r],a,b,c]

yes
```

Delete from List

Suppose we have a list L and an element X, we have to delete X from L. So there are three cases −

- If X is the only element, then after deleting it, it will return empty list.
- If X is head of L, the resultant list will be the Tail part.
- If X is present in the Tail part, then delete from there recursively.

Program

```
list_delete(X, [X], []).
list_delete(X,[X|L1], L1).
list_delete(X, [Y|L2], [Y|L1]) :- list_delete(X,L2,L1).
```

Output

```
yes
| ?- list_delete(a,[a,e,i,o,u],NewList).

NewList = [e,i,o,u] ?

yes
| ?- list_delete(a,[a],NewList).

NewList = [] ?

yes
```

Append into List

Appending two lists means adding two lists together, or adding one list as an item. Now if the item is present in the list, then the append function will not work. So we will create one predicate namely, list_append(L1, L2, L3). The following are some observations −

- Let A is an element, L1 is a list, the output will be L1 also, when L1 has A already.
- Otherwise, new list will be L2 = [A|L1].

Program

```
list_member(X,[X|Tail]).
list_member(X,[H|TAIL]) :- list_member(X,TAIL).

list_append(A,T,T) :- list_member(A,T),!.
list_append(A,T,[A|T]).
```

In this case, we have used (!) symbol, that is known as cut. So, when the first line is executed successfully, then we cut it, so it will not execute the next operation.

Output

```
| ?- list_append(a,[e,i,o,u],NewList).

NewList = [a,e,i,o,u]

yes
```

```
| ?- list_append(e,[e,i,o,u],NewList).

NewList = [e,i,o,u]

yes
| ?- list_append([a,b],[e,i,o,u],NewList).

NewList = [[a,b],e,i,o,u]

yes
```

Union Operation

Let us define a clause called list_union(L1,L2,L3), So this will take L1 and L2, and perform Union on them, and store the result into L3. As you know if two lists have the same element twice, then after union, there will be only one. So we need another helper clause to check the membership.

Program

```
list_union([],Z,Z).
list_union([X|Y],Z,W) :- list_member(X,Z), list_union(Y,Z,W).
list_union([X|Y],Z,[X|W]) :- \+(list_member(X,Z)),
list_union(Y,Z,W).
```

Note − In the program, we have used (\+) operator, this operator is used for **NOT**.

Output

```
| ?- list_union([a,b,c,d,e],[a,e,i,o,u],L3).

L3 = [b,c,d,a,e,i,o,u] ?

(16 ms) yes

| ?- list_union([a,b,c,d,e],[1,2],L3).

L3 = [a,b,c,d,e,1,2]

Yes
```

Intersection Operation

Let us define a clause called list_intersection(L1,L2,L3), So this will take L1 and L2, and perform Intersection operation, and store the result into L3. Intersection will return those elements that are present in both lists. So L1 = [a,b,c,d,e], L2 = [a,e,i,o,u], then L3 = [a,e]. Here, we will use the list_member() clause to check if one element is present in a list or not.

Program

```
list_member(X,[X|_]).
list_member(X,[_|TAIL]) :- list_member(X,TAIL).

list_intersect([X|Y],Z,[X|W]) :-
   list_member(X,Z), list_intersect(Y,Z,W).
list_intersect([X|Y],Z,W) :-
   \+ (list_member(X,Z)), list_intersect(Y,Z,W).
list_intersect([],Z,[]).
```

Output

```
| ?- list_intersect([a,b,c,d,e],[a,e,i,o,u],L3).

L3 = [a,e] ?

yes
| ?- list_intersect([a,b,c,d,e],[],L3).

L3 = []
yes
```