User's Manual for List Processing (lp.pro)

Introduction:

This manual will go over all the different methods that are associated with lp.pro, and give demonstrations and explanations for each. Lp.pro features prolog predicates, which the users is allowed to perform queries on. By using this guide it should give you more insight to who each method was developed and executed. Many of these programs feature recursion, where the tail becomes the input for the same method. This works because it will do the recursion until it comes to a terminal point (which needs to be explicitly stated); once the recursion makes it to the terminal statement, it will then go to the next possible statement that it can execute which is the previous statement from which the current statement was derived.

Table of Contents:

1. first ([H|_],H)
2. rest ([_|T],T)
3. last_element(List,Last)
4. write_list(H|T)
5. write_list_reversed([H|T])
6. size(List,L)
7. count(Element, [Element|Rest],Count)
8. element_of(X, List)
9. contains(List ,X)
10. nth(N, List ,E)
11. pick(List ,Item)
12. sum(List ,Sum)
13. make_list(Length,Element,[Element|Rest])
14. iota(N,IotaN)
15. add_first(X,List ,[X|L])
16. add_last(X,[H|T],[H|TX])
17. esrever(List, R) -- (reverse backwards)
18. join_lists([H|T1], L2, [H|T3])
19. product([Head|Tail], Product)
20. factorial(N,Nfactorial)
21. make_set([H|T],TS) / make_set([H|T],[H|TS])
22. replace(ListPosition,Object,[H|T1], [H|T2])
23. remove(First, [First|Rest],Rest).
24. take(List,Element,Rest)
25. split([[A,B]|T],[A|AA],[B|BB])
26. min_pair(N1,N2,N1)
27. max_pair(N1,N2,N1)
28. min([List],Min)
29. max([List],Max)
30. sort_inc(UnsortedList,SortedList)

31. sort_dec(UnsortedList,SortedList)
32. a_list(First_list,Sec_list,Assoc_list)
33. assoc(AList,Key,Value)

Entries

1) first

first([H | _], H).

Semantics/Syntax: First is a simple method to explicitly state how you get the head of the list. For a more applicable way of doing this, please see the entry for last_element. In this method, it asks for the head of the list by denoting it with square brackets and an H followed by a | and then an underscore. Outside the bracket is another H which can be any variable name you would like to call the head of the list.

H = variable to represent the head of the list.
_ = prolog doesn't care what value is there.

Demo:

?- first([2,4,5,7],First).
First = 2.

2) **rest**

rest ([_ | T], T).

Semantics/Syntax: Rest is a simple method to explicitly state how you get the tail of the list. For a more applicable way of doing this, please see the entry for last_element. In this method, it asks for the tail of the list by denoting it with square brackets with an underscore followed by a | and a T. Outside the bracket is another T which can be any variable name you would like to call the tail of the list.

T = variable to represent the tail of the list.
_ = prolog doesn't care what value is there.

Demo:

?- rest([2,4,5,7],Rest).
Rest = [4, 5, 7].

3) last_element

last_element([H|[]],H).
last_element([_|T], Result):- last_element(T,Result).

More explicitly…
%last_element(List,Last) :- rest(List,[]), first(List,Last).

%last_element(List,Last) :- rest(List,Rest), last_element(Rest,Last).

Semantics/Syntax: This is a more conventional way of getting the head and the tail of the list, but it also will tell you the last element of the list. This is done featuring recursion by entering tail into last_ element again.

Demo:
?- last_element([2,4,5,7],LastElement).
LastElement = 7 .

4) write_list

write_list([ ]).
write_list([H|T]) :- write(H), nl, write_list(T).

Semantics/Syntax:
Write is a method that will print out the list that you enter. This method uses the write method which just prints things to the standard output stream.

Demo:

?- write_list([4,5,8,9]).
4
5
8
9
true.

5) write_listrev

write_list_reversed([ ]).
write_list_reversed([H | T]) :- write_list_reversed(T), write(H), nl

Semantics/Syntax:
This method takes the list that you input and prints it out in reverse. The method uses recursion of the tail until it gets to one element and then prints it.

Demo:
?- write_list([1,2,3,4,5,6]).
1
2
3

4
5
6
true.

6) size(length)

```
size([],0).
size([_|T],L) :- size(T, K), L is (1 + K).
```

Semantics/Syntax:
This method has one input statement and one output statement. It's purpose is to tell you how long an input list is. The length is calculated by taking the size of the tail, chich is done recursively, and then it adds one.

Demo:

```
?- size([1,2,3,4,5,6],Size).
Size = 6.
```

7) count(Element,List,Count).

```
count(_, [ ], 0).
count(Element, [Element|Rest],Count):-
  count(Element,Rest,RestCount),
  Count is RestCount + 1.
count(Element,[_|Rest], Count) :-
  count(Element, Rest, Count).
```

Semantics/Syntax:
The goal of count is to count the number of user specified elements (via input) there are in a list. This method features three possibilities, one where there is not list, one where the head is the element, and one where the element is only in the tail. The recursion is used by deriving off of list Rest. and seeing how many the count is within that derivation.

Demo:

```
?- count(4,[3,4,5,4,9,4,6,7,8,4],Count).
Count = 4 .

?- count(5,[3,4,5,4,9,4,6,7,8,4],Count).
Count = 1 .
```

8) element_of(member).

```
element_of(X, [ X | _ ]).
```

```
element_of(X, [ _ | T]) :- element_of(X,T).
```

Semantics/Syntax:
Element_of is a method that looks to see if an element that was specified by input parameter X, is in the give list. The recursion is of the same element and is done based on the derivations of the tail.

Demo:
```
?- element_of(2,[4,5,8,3,2,7]).
true .

?- element_of(9,[4,5,8,3,2,7]).
false.
```

9) contains

```
contains([X|_],X).
contains([_|T],X) :- contains(T,X).
```

Semantics/Syntax:
contains is a method that looks to see if an element that was specified by input parameter X, is in the given list. The recursion is of the same element and is done based on the derivations of the tail.

Demo:
```
?- contains([4,5,9,7,3,1,6],2).
false.

?- contains([4,5,9,7,3,1,6],4).
true .
```

10) nth

```
nth(0,[H|_],H).
nth(N,[_|T],E) :- K is N-1, nth(K,T,E).
```

Semantics/Syntax:
Give an input of N and a list; N tells the program what place to look for the value of the element. Prolog counts with zero, so the fist element of the list is zero.

Demo:
```
?- nth(0,[5,8,7,3,2],Position).
```

Position = 5 .

?- nth(0,[5,8,7,3,2],ValueInThePosition).
ValueInThePosition = 5 .

?- nth(4,[5,8,7,3,2],ValueInThePosition).
ValueInThePosition = 2 .

?- nth(1,[5,8,7,3,2],ValueInThePosition).
ValueInThePosition = 8 .

11) pick

```
pick(L,Item) :-
  length(L, Length),
  random(0,Length,RN),
  nth(RN, L, Item).
```

Semantics/Syntax:
This method features nth, and randomly picks number for the placement of the list and then retrieves the value of the item. The range of the numbers is the length of the list.

Demo:
?- pick([1,2,3,4,5,6],Item).
Item = 2 .

12) sum

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

Semantics/Syntax:
Sum adds all of the elements of the list together. Recursion is done to get the sum of the tail and then it is added to the value of the head.

Demo:
?- sum([1,1,1,1,1,1,1,1,1],Sum).
Sum = 9.

?- sum([8,7,6,2,4,5,11,2],Sum).
Sum = 45.

1) make_list

```
make_list(0,_,[]).
make_list(Length, Element, [Element|Rest]) :-
  K is Length - 1,
  make_list(K, Element, Rest).
```

Semantics/Syntax:
Make_list takes the input of how many/ how long you want a list to be, the element that you want it to be consisted of, and what you want to call it. The purpose of making a list is to make a list of the users desired length.

Demo:

```
?- make_list(3,L,List).
List = [L, L, L] .
```

14) iota

```
iota(0,[]).
iota(N,IotaN) :-
  K is N - 1,
  iota(K, IotaK),
  add_last(N,IotaK,IotaN).
```

Semantics/Syntax:
Iota uses takes a number and an output variable/label. The number will then become the terminating set as the program is designed to count from one to the number that was inputted.

Demo:

```
?- iota(6,Iota).
Iota = [1, 2, 3, 4, 5, 6] .

?- iota(2,Iota).
Iota = [1, 2] .

?- iota(25,Iota).
Iota = [1, 2, 3, 4, 5, 6, 7, 8, 9|...] .
```

15) add_first

```
add_first(X,L,[X|L]).
```

Semantics/Syntax:

This method adds an Element to a specified list and then returns the new list with the added element. X is added to the front of the indicated list.

demo:
?- add_first(2,[3,4,5,6],List).
List = [2, 3, 4, 5, 6].

?- add_first(7,[3,4,5,6],List).
List = [7, 3, 4, 5, 6].

16) add_last

```
add_last(X,[],[X]).
add_last(X,[H|T],[H|TX]) :- add_last(X,T,TX).
```

Semantics/Syntax:
By analogy to add_first, this method adds an element to the end of the list.

demo:
?- add_last(2,[3,4,5,6],List).
List = [3, 4, 5, 6, 2] .

?- add_last(7,[3,4,5,6],List).
List = [3, 4, 5, 6, 7] .

17) esrever (reverse)

```
esrever([],[]).
esrever([H|T], R) :-
  esrever(T,Rev), add_last(H,Rev,R).
```

Semantics/Syntax:
Esrever is reverse backwards and will print a list backwards in the standard output stream. This method asks for a list and will give the reversed list back. It also uses add_last to help create the list.

Demo:
?- esrever([1,2,3,4,5,6],Reverse).
Reverse = [6, 5, 4, 3, 2, 1] .

?- esrever([5,4,7,3,1,9],Reverse).
Reverse = [9, 1, 3, 7, 4, 5] .

18) join_lists

```
join_lists([],L,L).
join_lists([H|T1], L2, [H|T3]) :- join_lists(T1,L2,T3).

join_lists(L1,L2,L3, Result) :-
  join_lists(L1,L2,L12), join_lists(L12,L3,Result).

join_lists(L1,L2,L3,L4,Result) :-
  join_lists(L1,L2,L3,L123), join_lists(L123, L4,Result).
```

Semantics/Syntax:
Join_lists can combine two to four lists. This program will combine the first two lists together and then add in the others sequentially after that.

Demo:

```
?- join_lists([1,2,3,4],[5,6,7,8],Join).
Join = [1, 2, 3, 4, 5, 6, 7, 8].

?- join_lists([1,2,3,4],[5,6,7,8],[1,5,9],Join).
Join = [1, 2, 3, 4, 5, 6, 7, 8, 1|...].
```

19) product

```
product([],1).
product([Head|Tail], Product) :-
  product(Tail,ProductOfTail),
  Product is Head * ProductOfTail.
```

Semantics/Syntax:
Product multiplies all of the elements in the list together; all of the elements need to be numeric in nature. The method uses recursion to get the last two numbers in the list and then multiplies them together. The product of those numbers will then be multiplied to the next number; it will continue till the list is complete.

Demo:

```
?- product([1,2,3],Product).
Product = 6.

?- product([1,2,30],Product).
Product = 60.

?- product([1,1,1,1,1],Product).
Product = 1.
```

20) factorial

```
factorial(N,Nfactorial) :-
  iota(N,IotaN),
  product(IotaN,Nfactorial).
```

Semantics/Syntax:
To get a factorial, you must multiply the number to all of the integers between that and one, inclusive. Therefore, this program makes good use of iota, which lists the numbers in a list, and product which multiplies the list that was made by iota.

Demo:

```
?- factorial(4,Factorial).
Factorial = 24 .

?- factorial(1,Factorial).
Factorial = 1 .

?- factorial(9,Factorial).
Factorial = 362880 .
```

21) make_set

```
make_set([],[]).
make_set([H|T],TS):-
  member(H,T),
  make_set(T, TS).
make_set([H|T],[H|TS]) :-
  make_set(T,TS).
```

Semantics/Syntax:
This method creates separate elements into a list. You can type in two lists, one as an actual list, and one as a variable, respectively. Make set will make you a set by creating a list.

Demo:

```
?- make_set([1,4],Set).
Set = [1, 4].

?- make_set([1,4,2],Set).
Set = [1, 4, 2].
```

```
?- make_set([1,4,2,7,9,2],Set).
Set = [1, 4, 7, 9, 2] .


?- make_set([1,4],Set).
Set = [1, 4].

?- make_set([1,4,2],Set).
Set = [1, 4, 2].

?- make_set([1,4,2,7,9,2],Set).
Set = [1, 4, 7, 9, 2] .
```

22) replace

```
replace(0,Object, [_|T],[Object|T]).
replace(ListPosition,Object,[H|T1], [H|T2]) :-
  K is ListPosition - 1,
  replace(K,Object,T1,T2).
```

Semantics/Syntax:
Replace will remove an element from the list and replace it with what you would like. To do this you need to know where the element you want to replace is (index from 0), and state what you will replace it with.

Demo:

```
?- replace(2,Object,[0,1,2,3,4],List).
List = [0, 1, Object, 3, 4] .

?- replace(0,replacingWith,[0,1,2,3,4],List).
List = [replacingWith, 1, 2, 3, 4]

?- replace(5,5,[0,1,2,3,4],List).
false.
```

23) remove

```
remove(_,[],[]).
remove(First, [First|Rest],Rest).
remove(Element,[First|Rest],[First|RestLessElement]) :-
  remove(Element,Rest,RestLessElement).
```

Semantics/Syntax:

This method removes the element that you want from the list. Looking at the code, there is no explicit way of stating where the deletion occurs, but the first remove statement that has parameters, just removes the head of the list. The second remove statement where we are removing something in the list. Will move through the tail of the list until if finds the element to be deleted at the head of the list, and then it will delete the element at the head of the list. The rest of the list will be the same minus the one element.

Demo:
?- remove(3,[1,2,3,4,5,6],List).
List = [1, 2, 4, 5, 6] .

?- remove(8,[1,8,6,7,3],List).
List = [1, 6, 7, 3] .

?- remove(0, [1,8,6,7,3], List).
List = [1, 8, 6, 7, 3] .

24) take

```
take(List,Element,Rest) :-
  pick(List,Element),
  remove(Element,List,Rest).
```

Semantics/Syntax:
The method take is very similar to the method remove, but this has an input of a list and two variable names, Element and Rest. The pick method that is carried out within the method take chooses a random element in the list, which then gets deleted by the next statement. Both the element that was deleted and the rest of the list gets printed to the standard output stream.

Demo:
?- take([3,7,9,2,6,1],Element, Rest).
Element = 1,
Rest = [3, 7, 9, 2, 6] .

?- take([3,7,9,2,6,1],Element, Rest).
Element = 9,
Rest = [3, 7, 2, 6, 1] .

?- take([3,7,9,2,6,1],Element, Rest).
Element = 6,
Rest = [3, 7, 9, 2, 1] .

?- take([3,7,9,2,6,1],Element, Rest).
Element = 9,
Rest = [3, 7, 2, 6, 1] .

?- take([3,7,9,2,6,1],Element, Rest).
Element = 1,
Rest = [3, 7, 9, 2, 6] .

25) split

```
split([],[],[]).
split([[A,B]|T],[A|AA],[B|BB]) :-
  split(T,AA,BB).
```

Semantics/Syntax:
Split will take a list of equal length and then separate the lists so that every other element from the head of the list will be one list, and every other element from the second element will be in the second list. This method only needs one input value as a list and two output values.

Demo:
?- split([[1,2],[3,4],[5,6]],L1,L2).
L1 = [1, 3, 5],
L2 = [2, 4, 6].

26) min_pair

```
min_pair(N1,N2,N1) :- N1 =< N2.
min_pair(N1,N2,N2) :- N2 =< N1.
```

Semantics/Syntax:
This method takes three numeric input values where the first to are the elements you are going to compare, and the third one is the one that you are asking prolog if it is the minimum pair out of the two elements that you presented.

Demo:
?- min_pair(7,6, Min).
Min = 6.

?- min_pair(7,8, Min).
Min = 7 .

27) max_pair

```
max_pair(N1,N2,N1) :- N1 >= N2.
max_pair(N1,N2,N2) :- N2 >= N1.
```

Semantics/Syntax:
This method takes three numeric input values where the first to are the elements you are going to compare, and the third one is the one that you are asking prolog if it is the maximum pair out of the two elements that you presented.

Demo:
```
?- min_pair(2,9, Min).
Min = 2 .

?- min_pair(7,6, Min).
Min = 6.

?- min_pair(7,8, Min).
Min = 7 .
```

28) min

```
min([H],H).
min(NumberList, MinNumber) :-
  NumberList = [H|T],
  min(T, MinTail),
  min_pair(H, MinTail, MinNumber).
```

Semantics/Syntax:
Min finds the minimum number in a list. This features the min_pair method by using min to recursively find the last two elements, and then finding the minimum pair of those two. Then the minimum pair of those two will move on and then be compared to the next element, until it gets to the head of the list.

Demo:
```
?- min([1,8,9,4,6,2],Minimum).
Minimum = 1 .

?- min([10,8,9,4,6,2],Minimum).
Minimum = 2 .
```

29) max

```
max([H],H).
max(NumberList, MaxNumber) :-
  NumberList = [H|T],
  max(T, MaxTail),
  max_pair(H, MaxTail, MaxNumber).
```

Semantics/Syntax:
Max finds the maximum number in a list. This features the max_pair method by using max to recursively find the last two elements, and then finding the maximum pair of those two. Then the maximum pair of those two will move on and then be compared to the next element, until it gets to the head of the list.

Demo:

```
?- max([1,8,9,4,6,2],Maximum).
Maximum = 9 .

?- max([10,8,9,4,6,2],Maximum).
Maximum = 10 .
```

30) sort_inc

```
sort_inc([],_).
sort_inc(UnsortedList,SortedList) :-
  max(UnsortedList,Max),
  remove(Max, UnsortedList,UnsortedList2),
  sort_inc(UnsortedList2, UnsortedList3),
  add_last(Max,UnsortedList3,SortedList).
```

Semantics/Syntax:
Sort_inc sorts a list in increasing order. It finds the maximum element of the list, deletes it and then put it as the last element in the new list. This is a recursion, so it will begin with the full list and then work its way up, as it keeps removing the maximums.

Demo:

```
?- sort_inc([5,7,3,1,9,0],IncreasingList).
IncreasingList = [0, 1, 3, 5, 7, 9] .

?- sort_inc([5,6,8,7,2,4,0],IncreasingList).
IncreasingList = [0, 2, 4, 5, 6, 7, 8] .
```

31) sort_dec

```
sort_dec([],_).
sort_dec(UnsortedList,SortedList) :-
  min(UnsortedList,Min),
  remove(Min, UnsortedList,UnsortedList2),
  sort_dec(UnsortedList2, UnsortedList3),
  add_last(Min,UnsortedList3,SortedList).
```

Semantics/Syntax:
Sort_dec sorts a list in decreasing order. It finds the minimum element of the list, deletes it and then put it as the first element in the new list. This is a recursion, so it will begin with the full list and then work its way up, as it keeps removing the minimums.

Demo:

```
?- sort_inc([5,7,3,1,9,0],IncreasingList).
IncreasingList = [0, 1, 3, 5, 7, 9] .

?- sort_inc([5,6,8,7,2,4,0],IncreasingList).
IncreasingList = [0, 2, 4, 5, 6, 7, 8] .
```

32) a_list

```
a_list([],[],_).
a_list(First_list,Sec_list,Assoc_list) :-
   First_list = [H1|T1],
   Sec_list = [H2|T2],
   a_list(T1,T2,Assoc_list2),
   add_first(pair(H1,H2),Assoc_list2,Assoc_list).
```

Semantics/Syntax:
A_list associates two lists of the same size together. The first element of the list A will be paired with the first element of list B; that trend will happen until the end of the list. This method will also print out a list of pair() elements with the two corresponding elements as the two parameters in pair().

Demo:

```
?- a_list( [one, two, three, four],[1,2,3,4], List).
List = [pair(one, 1), pair(two, 2), pair(three, 3), pair(four, 4)|_3708] .

?- a_list( [one, two, three, four],[uno, dos, tres, quatro], List).
List = [pair(one, uno), pair(two, dos), pair(three, tres), pair(four, quatro)|_3720] .
```

33) assoc

```
assoc(AList,Key,Value) :-
  AList = [H|_],
  H = pair(Key, Value).
assoc(AList, Key,Value) :-
  AList = [_|T],
  assoc(T,Key,Value).
```

Semantics/Syntax:
This program asks to find the corresponding value of the item based off of the key that the user provides. The first method of the association list is for starting off the association list as well as finishing it because the program is recursive and in this instance, it will go till the last pair in the list.

Demo:

```
?- assoc([pair(1, one),pair(2,two), pair(3,three)],2,Value).
Value = two .

?- assoc([pair(1, one),pair(2,two), pair(3,three)],1,Value).
Value = one .
```