List Processing in Prolog User's Guide

By Will Schell

Table of Content:

Introduction

This is a user's guide to list processing in prolog programming. We will discuss the methods and the actual commands that should be used and how to program them. We will first display and show all of the prolog functions and each has a reference to another page so that you can learn about each part.

1. List Processing Prolog Code

All will be referenced in section 2. Use corresponding numbers after the '1.' to find the correct list process.

1.01

```
%%% writelist

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

1.02

```
%%% member

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

1.03

```
%%% size

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

1.04

```
%%% item

item(X,[H|_],H) :- X=0.
item(X,[_|T],Y) :-
  X > 0,
  Z is X - 1,
  item(Z,T,Y).
```

1.05

```
%%% append

append([],L,L).
append([H|T1],L2,[H|T3]) :- append(T1,L2,T3).
append(L1,L2,L3,Result) :-
  append(L1,L2,L12),append(L12,L3,Result).
append(L1,L2,L3,L4,Result) :-
  append(L1,L2,L3,L123),append(L123,L4,Result).
```

1.06

```
%%% last

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

1.07

```
%%% remove

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

1.08

```
%%% replace

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

1.09

```
%%% makelist

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

1.10

```
%%% reverse

reverse([],[]).
reverse([H|T],R) :-
  reverse(T,Rev),lastput(H,Rev,R).
```

1.11

```
%%% lastput

lastput(E,[],[E]).
lastput(E,[H|T],[H|L]) :- lastput(E,T,L).
```

1.12

```
%%% pick

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

1.13

```
%%% take

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

1.14

```
%%% iota

iota(0,[]).
iota(N,IotaN) :-
  NM1 is N - 1,
  iota(NM1,IotaNM1),
  lastput(N,IotaNM1,IotaN).
```

1.15

```
%%% sum

sum([],0).
sum([H|T],Sum) :-
  sum(T, SumT),
  Sum is H + SumT.
```

1.16

```
%%% min

min([H], H).
min([H | T], H) :-
      min(T, L),
      H =< L, !.
min([_ | T], A) :-
      min(T, A).
```

1.17

```
%%% max

max([H], H).
max([H | T], H) :-
      max(T, L),
      H >= L, !.
max([_ | T], A) :-
      max(T, A).
```

1.18

```
%%% sort_inc

insert(H,T,[H|T]).
sort_inc([],[]).
sort_inc(Unordered,Ordered) :-
  min(Unordered,Minimum),
  remove(Minimum,Unordered,Rest),
  sort(Rest,Number),
  insert(Minimum,Number,Ordered).
```

1.19

```
%%% sort_des

sort_dec([],[]).
sort_dec(A,B) :-
  sort_inc(A,C),
  reverse(C,B).
```

1.20

```
%%% alist

alist([],[],[]).
alist([H1|T1],[H2|T2], Alist) :-
  insert([pair(H1,H2)],Nlist,Alist),
  alist(T1,T2,Nlist).
```

1.21

```
%%% assoc

assoc([pair(K,V)|_],K,V).
assoc([_|T],K,V) :-
  assoc(T,K,V).
```

1.22

```
%%% flatten

flatten([],[]).
flatten([H|T],L) :-
  atom(H),
  flatten(T,Tflattened),
  append([H],Tflattened,L).
flatten([H|T],L) :-
  flatten(H,FlatHead),
  flatten(T,FlatTail),
  append(FlatHead,FlatTail,L).
```

2. List Processing Code w/ descriptions

To see these in full action, check out section 3. There will be corresponding numbers like in this section after the '3.'

2.01

```
%%% writelist

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

writelist :: this takes one argument, a list, and will write the list out for the user. This will print it out on multiple lines. This is a recursive function to go back into the write list using the tail of the list after the head of the list has been printed out. **See 3.01 for the demo.**

2.02

```
%%% member

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

member :: this function will tell us whether or not an element is in a list. This takes two arguments, the first being the thing that you want to be matched to in the list, and the second argument being the list itself. **See 3.02 for the demo.**

2.03

```
%%% size

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

size :: the size function takes in two arguments and will tell you the size of the list. The first being the list, the second argument with then be what you want to be printed out with the final number. This meaning if you call the second argument 'Size' then you will get the output of:
Size = #;
This also has a recursive function so that it takes in the count of how many times it recursively calls the size function. That is what will be outputted. **See 3.03 for the demo.**

2.04

```
%%% item

item(X,[H|_],H) :- X=0.
item(X,[_|T],Y) :-
  X > 0,
  Z is X - 1,
  item(Z,T,Y).
```

item :: item is used for the index of a certain element in the list. It takes three arguments. The first is the index. This can be represented by an integer zero or higher. Zero will return the first element of the list. The second argument is the list you want to index. The last argument is the variable you want to be displayed, just like in **size(2.03)**. This is a recursive function to search through the list starting at the head and going through the list. If the list runs out before the index you want to see appears it will return false. **See 3.04 for the demo.**

2.05

```
%%% append

append([],L,L).
append([H|T1],L2,[H|T3]) :- append(T1,L2,T3).
append(L1,L2,L3,Result) :-
  append(L1,L2,L12),append(L12,L3,Result).
append(L1,L2,L3,L4,Result) :-
  append(L1,L2,L3,L123),append(L123,L4,Result).
```

append :: this function can take up to five arguments. The last argument must be a variable that you store the newly created list in. What append will do is take 2-4 lists and combine them together into 1 list that will be displayed in your variable in the last argument. What this does is goes in through the amount of lists that you have, breaks it down into smaller functions to combine the lists, then combines the bigger lists until they are completely combined into one list. **See 3.05 for the demo.**

2.06

```
%%% last

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

last :: this function will return the last item in a list. It takes two arguments, the first being the list and the second being the variable you want to call the list in the output. What this does is recursively goes through the list until the tail of the list is empty, or [], then it will print out the head of the list, or the last element. **See 3.06 for the demo.**

2.07

```
%%% remove

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

remove :: this function will remove an element from the list if it exists. It takes three arguments, first the element you want removed from the list, second the list, third the variable you want displayed with your list in the output. What this does is it will check the first element to see if the is the element to be removed, if not it will recursively go through the list until it finds the first instance of the element to be removed. Once it is removed then the new list will be displayed. **See 3.07 for the demo.**

2.08

```
%%% replace

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

replace :: This function is used to replace an element in a list. It takes in four arguments. First the index of where you want to replace, this is starting at index 0 not 1. The second is the item you want to input into that index. Third is the original list. And finally, the last is going to be a variable to be displayed in the output with the newly created list. This function will recursively go through the list, if the head of

the list is the index we want then we will replace it. If it isn't we will reduce the index by 1 and move onto the rest of the list. **See 3.08 for the demo.**

2.09

```
%%% makelist

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

makelist :: this function is will make a list for you. It takes in three arguments. The first will be the length of the list that you are creating, the second will be the element you want in the list, the third will be the variable to be displayed with the newly created list. What this does is it will insert the element you want into the list however many times until the length has been achieved. **See 3.09 for the demo.**

2.10

```
%%% reverse

reverse([],[]).
reverse([H|T],R) :-
  reverse(T,Rev),lastput(H,Rev,R).
```

reverse :: What this will do is reverse the order of the list you have. This takes two arguments. The first is the list you want to be reversed, the second is the variable name you will give to be output with the reversed list. This will recursively go into the list and then we will call the **lastput(1.11, 2.11)** function to reverse the list. We go through the entire list before we start inserting the values. **See 3.10 for the demo.**

2.11

```
%%% lastput

lastput(E,[],[E]).
lastput(E,[H|T],[H|L]) :- lastput(E,T,L).
```

lastput :: this was mentioned above in **reverse(2.10)**. This function will take three arguments. The first will be what you want to be put into the list, the second will be the list, and the third will be a variable displayed with the newly created list at the output. We recursively call lastput until we get an

empty list in the tail, then we will put the new element at the end of the list. **See 3.11 for the demo.**

2.12

```
%%% pick

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

pick :: this function allows us to randomly select an element in the list. It takes in two arguments to do this. The first is the list, the second is the variable displayed at the output with the randomly selected element. We first will get the length of the list, then we will use that to randomly select an index from 0 to the length of the list. Then it calls the **item** function with the argument of the random number, the list and the variable from the second argument. This then will display the result. **See 3.12 for the demo.**

2.13

```
%%% take

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

take :: this function will randomly remove an item from the list. We have three arguments, the list, an element and a variable to be displayed with the new list. This calls the **pick** function and this will randomly select an element from the list, then we pass that element to the remove function and it will remove the element from the list and display the rest of the list. **See 3.13 for the demo.**

2.14

```
%%% iota

iota(0,[]).
iota(N,IotaN) :-
  NM1 is N - 1,
  iota(NM1,IotaNM1),
  lastput(N,IotaNM1,IotaN).
```

iota :: this function will take two arguments. The first will be an integer. The second will be a variable to be displayed

with the newly created list. What this does is it recursively calls **iota**. We also will subtract the integer until the integer is zero. Once this is accomplished, the lastput function is called and writes the list in order from 1 to the integer that was inputted. This is stored in the variable and outputted. **See 3.14 for the demo.**

2.15

```
%%% sum

sum([],0).
sum([H|T],Sum) :-
  sum(T, SumT),
  Sum is H + SumT.
```

sum :: the sum function is used to get the sum of all the integers in a list. It takes two arguments, the first is the list and the second is the variable that the sum is stored in and displayed with. We recursively call up sum giving it the tail of the list and temporary variable, this then will add up all the numbers once the tail is empty, and then displays the sum. **See 3.15 for the demo.**

2.16

```
%%% min

min([H], H).
min([H | T], H) :-
      min(T, L),
      H =< L, !.
min([_ | T], A) :-
      min(T, A).
```

min :: this takes in 2 parameters, a list and the variable to store the min and display it. **min** is recursively called searching each head element and storing the smallest integer value into a temporary variable to compare it to other values in the list. This will then reach the end of the list and display the smallest value stored in the variable. **See 3.16 for the demo.**

2.17

```
%%% max

max([H], H).
max([H | T], H) :-
      max(T, L),
      H >= L, !.
max([_ | T], A) :-
      max(T, A).
```

max :: this takes in 2 parameters, a list and the variable to store the max and display it. **max** is recursively called searching each head element and storing the largest integer value into a temporary variable to compare it to other values in the list. This will then reach the end of the list and display the largest value stored in the variable. **See 3.17 for the demo.**

2.18

```
%%% sort_inc

insert(H,T,[H|T]).
sort_inc([],[]).
sort_inc(Unordered,Ordered) :-
  min(Unordered,Minimum),
  remove(Minimum,Unordered,Rest),
  sort(Rest,Number),
  insert(Minimum,Number,Ordered).
```

sort inc :: this function is used to sort the list in an increasing order. It takes 2 parameters, a list and then a variable for a new list. This will find the minimum value of the list and then insert it into a new list. This uses the sort function and the created insert function above to make this happen. The output will then be a sorted list from the lowest value to the highest value. **See 3.18 for the demo.**

2.19

```
%%% sort_des

sort_dec([],[]).
sort_dec(A,B) :-
  sort_inc(A,C),
  reverse(C,B).
```

sort dec :: this function takes two parameter to sort the list from highest to lowest. The first is the list you give it, the

second is the variable for the new list to be put into. What this does is it will pass the variables onto the **sort_inc(2.19)** and that will sort it in order lowest to highest. Then we will call the reverse function to reverse the list so that it is highest to lowest. Then this will be displayed. **See 3.19 for the demo.**

2.20

```
%%% alist

alist([],[],[]).
alist([H1|T1],[H2|T2], Alist) :-
  insert([pair(H1,H2)],Nlist,Alist),
  alist(T1,T2,Nlist).
```

alist :: this function is used to create pairs of values between 2 lists and put them into a new list. It takes three parameters. First a list, second another list and thirdly a variable for the new list. This will take the first element of each list and put them together in a **pair(E1,E2)** that will be inserted into a new list. It does this for all elements of each list. **See 3.20 for the demo.**

2.21

```
%%% assoc

assoc([pair(K,V)|_],K,V).
assoc([_|T],K,V) :-
  assoc(T,K,V).
```

assoc :: this function will use the values created from **alist** that will then be put into the first argument, then the second argument is a key and the third argument is the value for that key. The value and key come from the **pair(key,value)**. This then will recursively search through the list of pairs to find a match for either the key or value. You can search a list using the key to find a value or a value to find a key, or to see if a pair of a certain key and value you can type in both to get a true or false. **See 3.21 for the demo.**

2.22

```
%%% flatten

flatten([],[]).
flatten([H|T],L) :-
  atom(H),
  flatten(T,Tflattened),
```

```
  append([H],Tflattened,L).
flatten([H|T],L) :-
  flatten(H,FlatHead),
  flatten(T,FlatTail),
  append(FlatHead,FlatTail,L).
```

flatten :: flatten does this with two arguments. Takes a list of lists and makes one list out of it. Decomposes each list in sequential order. **See 3.22 for the demo.**

3. List Processing Demos

```
...@...:~/Desktop/COG366$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.0-rc2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- consult('a11/lp.pro').
true.
```

3.01

```
?- writelist([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p]).
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
true.
```

3.02

```
?- member(d,[a,b,c,d,e]).
true .
```

3.03

```
?- size([a,b,c,d,e,f,g,h,i,j,k,l],Size).
Size = 12.
```

3.04

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

3.05

```
?- append([we,hello,tooth],[will,goodbye,123],[3.4,woah,ok],Result).
Result = [we, hello, tooth, will, goodbye, 123, 3.4, woah, ok].
```

3.06

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

3.07

```
?- remove(s,[a,r,b,s,t,e,s,g],Result).
Result = [a, r, b, t, e, s, g] .
```

3.08

```
?- replace(0,here,[nothere,here,here,here],Result).
Result = [here, here, here, here] .
```

3.09

```
?- makelist(4,nyr,List).
List = [nyr, nyr, nyr, nyr] .
```

3.10

```
?- reverse([this,is,not,today,but,tomorrow],RevList).
RevList = [tomorrow, but, today, not, is, this] .
```

3.11

```
?- lastput(last,[first,second,third,fourth,fifth],List).
List = [first, second, third, fourth, fifth, last] .
```

3.12

```
?- pick([a,b,c,d,e,f],Result).
Result = c .

?- pick([a,b,c,d,e,f],Result).
Result = d .

?- pick([a,b,c,d,e,f],Result).
Result = e .
```

3.13

```
?- take([a,b,c,d,e,f],a,List).
List = [b, c, d, e, f] .
```

3.14

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

3.15

```
?- sum([1,4,3,5,6,3],Sum).
Sum = 22.
```

3.16

```
?- min([12,34,23,5,23,55,6,123,3,7],Min).
Min = 3.
```

3.17

```
?- max([12,34,23,5,23,55,6,123,3,7],Max).
Max = 123.
```

3.18

```
?- sort_inc([5,3,7,2,5,4,6,1,3,2,5],Result).
Result = [1, 2, 3, 4, 5, 6, 7] .
```

3.19

```
?- sort_dec([4,6,2,3,1,7,9],Result).
Result = [9, 7, 6, 4, 3, 2, 1] .
```

3.20

```
?- alist([1,2,3],[z,y,x],Result).
Result = [pair(1, z), pair(2, y), pair(3, x)].
```

3.21

```
?- assoc([pair(not,today),pair(hello,goodbye),pair(well,ok)],K,V).
K = not,
V = today ;
K = hello,
V = goodbye ;
K = well,
V = ok ;
false.

?- assoc([pair(not,today),pair(hello,goodbye),pair(well,ok)],not,V).
V = today .

?-
assoc([pair(not,today),pair(hello,goodbye),pair(well,ok)],not,today).
true .
```

Conclusion

These are very useful and successful list processing functions in prolog. These rules and the way they run help us come to conclusions and solve problems through Prolog programming. As these functions are listed above, there are still other out there that can help with other tasks as well. These ones are a great place to start to understand the way lists processing works and the functionality of them. The other important concept from this is recursion. This is used in most of these list processing functions and it is a tremendous tool to have to have a lot of time.