

PROLOG.  
Lists in PROLOG. Advanced Issues.  
List Ordering. Meta Predicates.

Antoni Ligęza

Katedra Automatyki, AGH w Krakowie

2011

- [1] Ulf Nilsson, Jan Maluszyński: **Logic, Programming and Prolog**, John Wiley & Sons Ltd., pdf, <http://www.ida.liu.se/~ulfni/lpp>
- [2] Dennis Merritt: **Adventure in Prolog**, Amzi, 2004  
<http://www.amzi.com/AdventureInProlog>
- [3] Quick Prolog:  
<http://www.dai.ed.ac.uk/groups/ssp/bookpages/quickprolog/quickprolog.html>
- [4] W. F. Clocksin, C. S. Mellish: **Prolog. Programowanie**. Helion, 2003
- [5] SWI-Prolog's home: <http://www.swi-prolog.org>
- [6] Learn Prolog Now!: <http://www.learnprolognow.org>
- [7] <http://home.agh.edu.pl/~ligeza/wiki/prolog>
- [8] <http://www.im.pwr.wroc.pl/~przemko/prolog>

## The idea of sorting by insertion

In order to sort a list:

- 1 check if the list is empty; an empty list is sorted,
- 2 if the list is not empty, then:
  - 1 take of the **Head**,
  - 2 sort the **Tail**,
  - 3 insert **Head** into an appropriate place of **Tail**.

## Insert sort

```
1 order([], []).
2 order([H|T], R):-
3     order(T, TR),
4     put(H, TR, R).
5
6 put(H, [], [H]):-!.
7 put(H, [X|Y], [H,X|Y]):-
8     H < X, !.
9 put(H, [X|Y], [X|Z]):-
10    put(H, Y, Z), !.
```

## Sorting by finding minimal/maximal element

```
1  min([X], X) :- !.
2  min([P|R], P) :- min(R, X), X > P, !.
3  min([P|R], X) :- min(R, X), X =< P.
4
5  max([X], X) :- !.
6  max([P|R], P) :- max(R, X), X =< P, !.
7  max([P|R], X) :- max(R, X), X > P.
8
9  min-sort([], []) :- !.
10 min-sort(L, [M|LS]) :-
11     min(L, M),
12     select(M, L, LM),
13     min-sort(LM, LS).
14
15 min-sort-iter(L, LS) :-
16     msi(L, [], LS).
17
18 msi([], LS, LS) :- !.
19 msi(L, LA, LS) :-
20     max(L, M),
21     select(M, L, LM), !,
22     msi(LM, [M|LA], LS).
```

## Idea of bubblesort

- 1 scan by pairs of elements from left to right,
- 2 if the order is wrong — correct; continue the scan,
- 3 if no correction takes place, the list is sorted.

## Bubblesort

```
1 busort(L,S):-  
2     swap(L,LS), !, busort(LS,S).  
3 busort(S,S).  
4  
5 swap([X,Y|T],[Y,X|T]):- X > Y.  
6 swap([Z|T],[Z|TT]):- swap(T,TT).
```

## Bubblesort 2a

```
1 busort2a(L,S):-  
2     append(F,[X,Y|T],L), X>Y, !, append(F,[Y,X|T],NL),  
3     busort2a(NL,S).  
4 busort2a(S,S).
```

## Mergesort: split, sort, merge

```
1 lse([],W1,W2,W1,W2,_):- !.
2 lse([H|T],L1,L2,W1,W2,l):- lse(T,[H|L1],L2,W1,W2,p), !.
3 lse([H|T],L1,L2,W1,W2,p):- lse(T,L1,[H|L2],W1,W2,l), !.
4
5 split([],[],[]).
6 split([H|T],[H|U],V):- split(T,V,U).
7
8 merge([H1|T1],[H2|T2],[H1|T2):-
9     H1 < H2, !,
10     merge(T1,[H2|T2],T).
11 merge([H1|T1],[H2|T2],[H2|T2):-
12     merge([H1|T1],T2,T),!.
13 merge(X,[],X):-!.
14 merge([],X,X).
15
16 mergesort([],[]):- !.
17 mergesort([H],[H]):- !.
18 mergesort(L,S):-
19     split(L,LL,LR),
20     mergesort(LL,SLL),
21     mergesort(LR,SLR),
22     merge(SLL,SLR,S).
```

## Idea of quicksort

- 1 select an arbitrary **threshold element**,
- 2 split the list into **smaller elements**, and **bigger elements**,
- 3 sort both the lists,
- 4 append the lists, including threshold element inside.

## Quicksort

```
1 qsort ([], []) .
2 qsort ([H|T], S) :-
3     split (H, T, L, R) ,
4     qsort (L, LS) , qsort (R, RS) ,
5     append (LS, [H|RS], S) .
6
7 split (_, [], [], []) .
8 split (H, [A|X], [A|Y], Z) :-
9     A =< H, !,
10    split (H, X, Y, Z) .
11 split (H, [A|X], Y, [A|Z]) :-
12    A > H, !,
13    split (H, X, Y, Z) .
```

## Example loop solutions

```
1 loop_infinite:-
2     repeat,
3     write('***loop: '),nl,
4     fail.
5
6 loop_infinite_read_write:-
7     repeat,
8     read(X), write('***loop: '), write(X),nl,
9     fail.
10
11 loop_find_fail:-
12     d(X),
13     write('***found: '),write(X),nl,
14     fail.
15 loop_find_fail.
16
17 loop_repeat_test:-
18     repeat,
19     d(X),
20     write('***found: '),write(X),nl,
21     read(Y), X = Y, write('***end: '),write(X),nl.
22 d(0). d(1). d(2). d(3). d(4). d(5). d(6). d(7). d(8). d(9).
```



## Example sort repeat-test

```

1  asort(L):-
2      retractall(list(_)),
3      assert(list(L)),
4      go.
5
6  go:-
7      repeat,
8      list(L),
9      write(L),nl,
10     noimprove(L), !.
11
12  noimprove(L):- sorted(L).
13  noimprove(L):-
14     append(P, [X,Y|K], L),
15     X > Y,
16     retract(list(L)),
17     append(P, [Y,X|K], N),
18     assert(list(N)),
19     fail.
20
21  sorted([_]).
22  sorted([X,Y|T]):- X=<Y,sorted([Y|T]).

```

## Example sort improve-fail

```
1  sort-improve-fail(L):-
2      retractall(list(_)),
3      assert(list(L)),
4      run.
5  run:-
6      improve,
7      fail.
8  run:- list(L), write(L),nl.
9
10 improve:-
11     list(L),
12     append(F, [X,Y|R],L),
13     X>Y,
14     retract(list(L)),
15     append(F, [Y,X|R],NL),
16     assert(list(NL)).
17 improve:-
18     list(L), \+sorted(L),
19     improve.
20
21 sorted([_]).
22 sorted([X,Y|T]):- X=<Y,sorted([Y|T]).
```

## Write a nested list with indent K

```
1 p(0):- write(''),!.
2 p(N):- write(' '),N1 is N-1, p(N1).
3
4 wl([H|T],D):- !, we(H,D), wl(T,D).
5 wl([],_).
6
7 we([H|T],D):- !, D2 is D+3, wl([H|T],D2).
8 we(H,D):- nl,p(D),write(H).
```

## Example: write a nested list with indent 3

```
1 ?- wl([1,[2,3],4,[5,[6,7]],8],3).
2
3     1
4       2
5         3
6     4
7       5
8         6
9           7
10      8
```

## example

```
1 :- dynamic fact/1.
2
3 find_list(L):-
4     collect([],L).
5 collect(L,W):-
6     retract(fact(X)),
7     !,
8     collect([X|L],W).
9 collect(W,W).
```

## example

```
1 ?- assert(fact(1)).
2 ?- assert(fact(2)).
3 ?- assert(fact(3)).
4
5 ?- find_list(L).
6 L = [3, 2, 1].
```

## example

```

1  :- dynamic(p/1).
2  :- dynamic(list/1).
3  :- assert(list([])).
4  p(1).
5  p(2).
6  p(3).
7  p(4).
8  p(5).
9
10 collect(_):-
11     p(X),
12     retract(list(T)),
13     assert(list([X|T])),
14     fail.
15 collect(L):- list(L).
16
17 % coollecti - iterative with retract
18 collecti(L):- collect_iter([],L).
19
20 collect_iter(T,L):- retract(p(X)), !,
21                    collect_iter([X|T],L).
22 collect_iter(L,L).

```

## example

```
1 translate([], []).
2 translate([H|T], [G|O]):-
3 dictionary(H,G),
4 translate(T,O).
5
6
7 dictionary(jest,is).
8 dictionary(to, this).
9 dictionary('chłopiec', 'a boy').
10 dictionary(dziewczynka, 'a girl').
```

## Extending Translation Capabilities

- 1 extend vocabulary,
- 2 one-to-many translation:
  - 1 item selection by context,
  - 2 item selection by frequency,
- 3 translations by patterns; longest first.

## findall/3

```
1 findall(+Template, :Goal, -Bag)
```

Creates a list of the instantiations Template gets successively on backtracking over Goal and unifies the result with Bag. Succeeds with an empty list if Goal has no solutions.

## bagof/3

```
1 bagof(+Template, :Goal, -Bag)
```

Unify Bag with the alternatives of Template, if Goal has free variables besides the one sharing with Template bagof will backtrack over the alternatives of these free variables, unifying Bag with the corresponding alternatives of Template. bagof/3 fails if Goal has no solutions.

## setof/3

```
1 setof(+Template, +Goal, -Set)
```

Equivalent to bagof/3, but sorts the result using sort/2 to get a sorted list of alternatives without duplicates.

## Calculating Average Salary

```
1  %%% A Database of facts
2
3  prac(adam,1400).
4  prac(bogdan,2300).
5  prac(cesiek,2700).
6  prac(damian,2400).
7  prac(eustachy,2600).
8
9  avg(AV):-
10     findall(P,prac(_,P),LP),
11     write(LP),nl,
12     sumlist(LP,SP),
13     length(LP,L),
14     AV is SP/L.
15
16  pracd(X,P):- prac(X,P),avg(AV), P > AV.
```



- ✘ `length(?List, ?Int)` – Int is the length of List,
- ✘ `sort(+List, -Sorted)` – list sorting; duplicates are eliminated,
- ✘ `msort(+List, -Sorted)` – list sorting; duplicates are not eliminated,
- ✘ `merge(+List1, +List2, -List3)` – merging two (sorted) lists; duplicates are left over,
- ✘ `merge_set(+Set1, +Set2, -Set3)` – merging two lists; duplicates are eliminated.

```
1  ?- length([2,3,4], X).
2  X = 3
3  ?- sort([4,6,3,1,3,6], X).
4  X = [1, 3, 4, 6]
5  ?- msort([4,6,3,1,3,6], X).
6  X = [1, 3, 3, 4, 6, 6]
7  ?- merge([1,3,4], [2,3], X).
8  X = [1, 2, 3, 3, 4]
9  ?- merge([1,3,4,2], [2,3], X).
10 X = [1, 2, 3, 3, 4, 2]
11 ?- merge_set([1,3,4], [2,3], X).
12 X = [1, 2, 3, 4]
```

- ✘ `maplist (:Pred, +List)` – Predykat `Pred` – application of a predicate to all list elements, until failure or end of the list,
- ✘ `maplist (:Pred, ?List1, ?List2)` – application of a predicate to all list elements, until failure or end of the list; results are put on `List2`,
- ✘ `maplist (:Pred, ?List1, ?List2, ?List3)` – application of a predicate to all pairs of elements of `List1` and `List2`, until failure or end of the list; results are put on `List3`, .

```
1 f(X, Y) :- Y is sin(X) + cos(X).
2 even(X) :- X mod 2 == 0.
3 %-----
4 ?- maplist(even, [1, 2, 3, 4]).
5 No
6
7 ?- maplist(even, [2, 4]).
8 Yes
9
10 ?- maplist(sqrt, [1, 2, 3], X).
11 X = [1.0, 1.41421, 1.73205]
12
13 ?- maplist(f, [1, 2, 3, 4], X).
14 X = [1.38177, 0.493151, -0.848872, -1.41045]
15
16 maplist(plus, [1, 2, 3], [1, 2, 3], X).
17 X = [2, 4, 6]
```

## Dlist: the idea

- ✘ a list can be represented as a **difference list**, i.e.

$$[a, b, c] = [a, b, c, d, e] - [d, e]$$

- ✘ hence, one can put:

$$[a, b, c] = [a, b, c|T] - T$$

where  $T$  can be unified with **any list**.

## Difference lists: efficient append through unification

```
1 :- op(200, xfy, -) .
2 % [a,b,c]-[]
3 % [a,b,c,d,e]-[d,e]
4 % [a,b,c,d,e|T]-[d,e|T]
5 % [a,b,c|T]-T
6
7 cappend(F-T, T-B, F-B) .
8
9 ?- cappend([a,b,c|T]-T, [d,e|S]-S, L) .
10 T = [d, e|S],
11 L = [a, b, c, d, e|S]-S.
```

# Selected Built-in List Predicates

```
append(?List1, ?List2, ?List3)
    Succeeds when List3 unifies with the concatenation of List1 and List2.
append(?ListOfLists, ?List)
    Concatenate a list of lists. Is true if Lists is a list of lists, and
    List is the concatenation of these lists.
member(?Elem, ?List)
    Succeeds when Elem can be unified with one of the members of List.
nextto(?X, ?Y, ?List)
    Succeeds when Y immediately follows X in List.
delete(+List1, ?Elem, ?List2)
    Delete all members of List1 that simultaneously unify with Elem
    and unify the result with List2.
select(?Elem, ?List, ?Rest)
    Select Elem from List leaving Rest. It behaves as member/2, returning
    the remaining elements in Rest.
nth0(?Index, ?List, ?Elem)
    Succeeds when the Index-th element of List unifies with Elem. Counting starts at 0.
nth1(?Index, ?List, ?Elem)
    Succeeds when the Index-th element of List unifies with Elem. Counting starts at 1.
last(?List, ?Elem)
    Succeeds if Elem unifies with the last element of List.
reverse(+List1, -List2)
    Reverse the order of the elements in List1 and unify the result with
    the elements of List2.
permutation(?List1, ?List2)
    Permutation is true when List1 is a permutation of List2.
flatten(+List1, -List2)
    Transform List1, possibly holding lists as elements into a 'flat' list
    by replacing each list with its elements (recursively). Example:

    ?- flatten([a, [b, [c, d], e]], X).
    X = [a, b, c, d, e]

sumlist(+List, -Sum)
    Unify Sum to the result of adding all elements in List.
max_list(+List, -Max)
    True if Max is the largest number in List. See also the function max/2.
min_list(+List, -Min)
    True if Min is the smallest number in List. See also the function min/2.
numlist(+Low, +High, -List)
```

The set predicates listed in this section work on ordinary unsorted lists. Note that this makes many of the operations order  $N^2$ . For larger sets consider the use of ordered sets as implemented by library `ordsets.pl`, running most these operations in order  $N$ . See section A.15.

`is_set(+Set)`

Succeeds if `Set` is a list (see `is_list/1`) without duplicates.

`list_to_set(+List, -Set)`

Unifies `Set` with a list holding the same elements as `List` in the same order.

If `list` contains duplicates, only the first is retained. See also `sort/2`. Example:

```
?- list_to_set([a,b,a], X)
```

```
X = [a,b]
```

`intersection(+Set1, +Set2, -Set3)`

Succeeds if `Set3` unifies with the intersection of `Set1` and `Set2`. `Set1` and `Set2` are lists without duplicates.

`subtract(+Set, +Delete, -Result)`

Delete all elements of set 'Delete' from 'Set' and unify the resulting set with 'Result'.

`union(+Set1, +Set2, -Set3)`

Succeeds if `Set3` unifies with the union of `Set1` and `Set2`. `Set1` and `Set2` are lists without duplicates. They may be the same list.

`subset(+Subset, +Set)`

Succeeds if all elements of `Subset` are elements of `Set` as well.

