Programming Languages Workshop
CS 67420
Lecture 8

Huji, Spring 2016

Dr. Arie Schlesinger

Prolog is a 'declarative' language

Clauses are statements about
what is true about a problem,
instead of
instructions how to accomplish the solution.

The Prolog system uses the clauses to
work out how to accomplish the solution by
searching through the space of possible solutions.

Prolog "logical variables"

Prolog uses "logical variables" :

That are diff from vars in other languages.

Can be used as *'holes'* in data structures which are gradually filled in as computation proceeds.

Unification

Unification is a built-in term-manipulation method that

-passes parameters,

-returns results,

-selects and constructs data structures.

`f(X,a(b,c)) and f(d,a(Z,c))…X/d and Z /b.`

`f(X,a(b,c)) and f(Z,a(Z,c))…X/b and Z/b.`

`f(c,a(b,c)) and f(Z,a(Z,c)) do not unify.`

No sugar please

The query `write_canonical(someTerm)` will display `someTerm` without infix operators

```
?- write_canonical(3+8*5).
+(3,*(8,5))
true.
```

Queries in the kb

KB lines starting with **:-** are "immediate" queries executed when the file is loaded.

Test code with assertions

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

% this is a test assertion as an inside query:

```
:- last([2,4,6],X), X=6.
```

Now from the interpreter:

```
?- last(Z,6).       % Z unbound var
Z = [6] .           % isn't it creative ?
```

trace

```
?- trace. % activate tracer
true.
[trace] 2 ?- last([],X).
Call: (7) last([], _G3355) ? creep
Fail: (7) last([], _G3355) ? creep
false.
```

Enter, "creeps" thru the trace, s "skips" calls:

```
[trace] 7 ?- last([5,4],X).
Call: (7) last([5, 4], _G3379) ? creep
Call: (8) last([4], _G3379) ? creep
Exit: (8) last([4], 4) ? creep
Exit: (7) last([5, 4], 4) ? creep
X = 4.
```

(to stop the detectives write "notrace." , "nodebug.")

=

```
?- X = 2+3. % no arithmetic is done here
X = 2+3
true

?- 1+2 = Y.
Y=1+2.
True

?- 2+3 = 5. % the same
false
```

“**=**” means: “try to unify two terms” : Left <-> Right

it is not an assignment,

nor it evaluates expressions

Arithmetic equality is not the same as unification

```
?- N = apple+pear.
N = apple+pear
true

?- apple+pear = N, N = +(apple,pear).
N=apple+pear
true
```

The plus (+) here is just creating compound terms

is

Use the operator "is" to evaluate arithmetic:

```
?- N is 2+3.
N = 5
true

?- N is apple + pear.
ERROR: …
```

The "is" activities:

(1) Eval the Right-hand expression (numbers)
(2) Unify the expr result with the Left side

is.. review

```
?- 5 is 2+3 .
true
?- 2+3 is 5. % no eval on left side..
false
```

The R side must be a ground term (no variables)

“is” does not force instantiation of variables

```
?- X is Y+1.
```

ERROR: is: Args are not sufficiently instantiated

`?- 5 is Y+3.` % Y is not forced to become 2

ERROR: is: Args are not sufficiently instantiated

Prolog uses depth-first search to answer queries

KB:

```
a(1).
a(2).
b(1).
b(2).
q(X,Y) :- a(X), b(Y).
```

Query

```
? q(E,F).
```

E = F, F = 1 ;

E = 1,

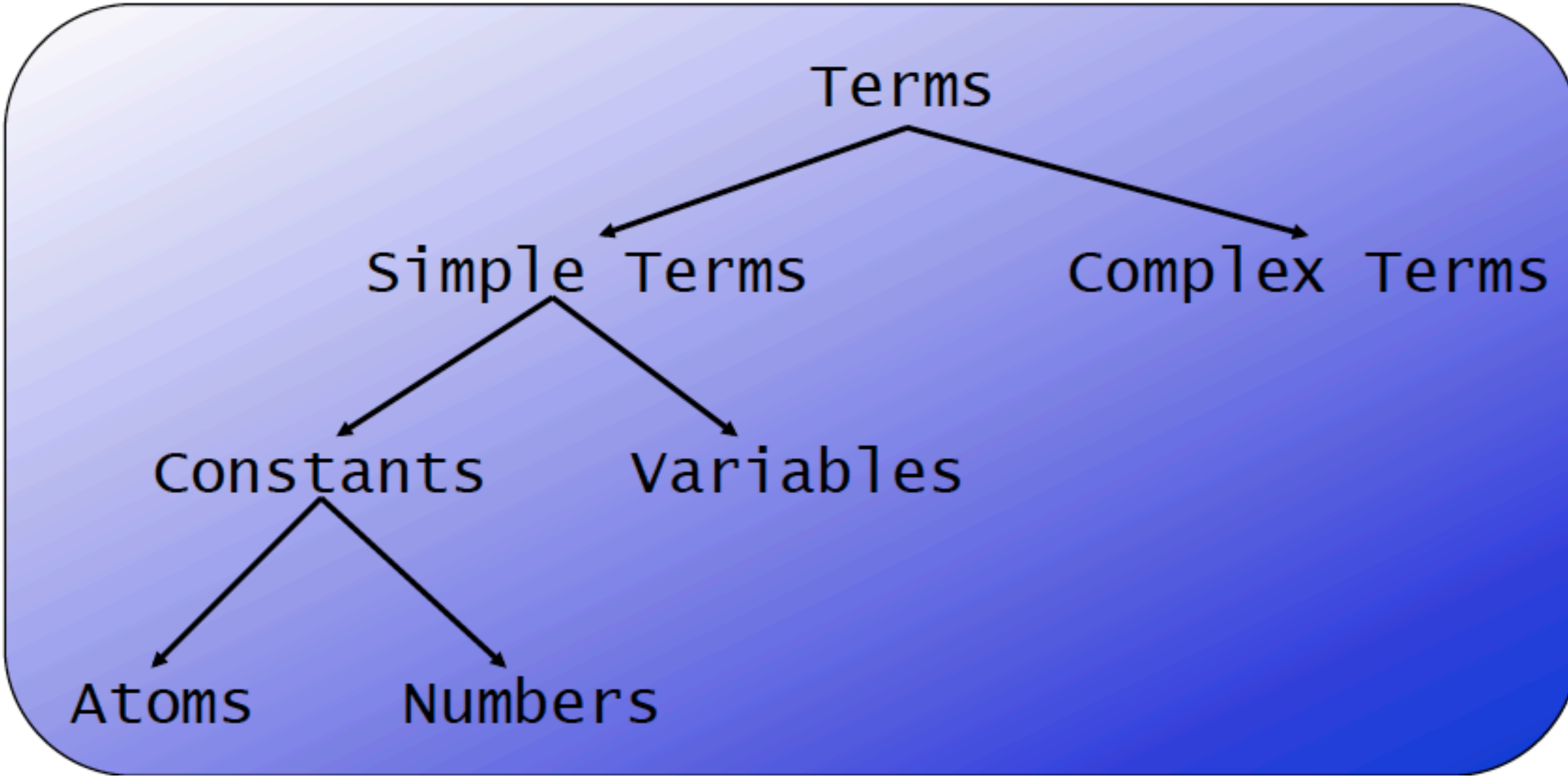
F = 2 ;

E = 2,

F = 1 ;

E = F, F = 2.

Type of terms



More detectives

Checking the type of a term

atom/1…Is the argument an atom?

integer/1… an integer?

float/1… a floating point number?

number/1… an integer or float?

atomic/1… a constant?

Constants: atoms, numbers

var/1… an uninstantiated variable?

nonvar/1… an instantiated variable or
another term that is not an
uninstantiated variable

Type checking: atom/1

```
?- atom(a).
true
?- atom(7). % numbers are not atoms
false
?- atom(X).
false
```

Type checking: atom/1

```
?- X=a, atom(X). % X is not a var anymore
X = a
true
?- atom(X), X=a.
false
```

Type checking: atomic/1

```
?- atomic(iosi).
true
?- atomic(5).
true
?- atomic(reads(ety,book)).
false
```

Type checking: var/1

```
?- var(iosi).
false
?- var(X).
true
?- X=2, var(X).% X got bounded !
false
```

Type checking: nonvar/1

```
?- nonvar(X).
false
?- nonvar(ety).
true
?- nonvar(23).
true

?- X=5,nonvar(X).
X = 5.

?- X=5,var(X).
false.
```

mapping

A relation between 2 data structures D1, D2 where each element of D2 is related to D1 by some evaluation on each element of D1

Mapping: The Full Map

Example:

sqList(Ln , Lsqn)

Ln..*list of numbers,* ***Lsqn***..*list of squares of numbers, of equal lengths*

```
sqList([], []).
sqList([A|X], [B|Y]) :- B is A * A,
                        sqList(X, Y).
```

example with compound terms

Map each list element (a number) to a term
s(A,B) where A is the number and B is its square.

```
sqterm(in,out)
For in: [1, 9, 15]
out is: [s(1,1), s(9, 81), s(15, 225)]
```

```
sqterm([], []).
sqterm([X|T], [s(X,Y)|L]) :-Y is X * X,
                             sqterm(T, L).
```

General scheme for full map

```
/* fullmap(In, Out) */

fullmap([], []).
fullmap([X|T], [Y|L]) :-
            transform(X,Y), fullmap(T, L).
```

a transformation table example

```
transform(a, 1).
transform(b, 2).
transform(c, 3).
transform(X, X).
```

a,b,c go to 1,2,3. Anything else goes to itself

```
?- fullmap([a, b, d], Z).
```

`Z = [1, 2, d]`

distinguishing between values

Sometimes the map needs to be sensitive to the input data:

```
Input: [1, 3, a, 5, d]
Output: [1, 9, a, 25, d]
a,d remain unchanged
```

```
sqInt([], []).
sqInt([X|T], [Y|L]) :-integer(X),
                      Y is X * X,
                      sqInt(T, L).
sqInt([X|T], [X|L]) :- sqInt(T,L).
```

A non sensitive variant

Using the infix binary compound term *, it is easy enough to give some "math-look" to the map:

```
Input: [1, 3, a, 5, d]
Output: [1, 9, a*a, 25, d*d]
```

```
sqInt([], []).
sqInt([X|T], [Y|L]) :-integer(X),
                      Y is X * X,
                      sqInt(T, L).
sqInt([X|T], [X*X|L]) :- sqInt(T,L).
```

Partial (conditional) Maps

Given an input list, partially map-it to an output list.

```
evens([], []).
evens([X|T],[X|L]):-0 is mod(X,2),evens(T,L).
evens([X|T], L):-1 is mod(X,2), evens(T, L).

?- evens([1, 2, 3, 4, 5, 6], Q),
```

Q = [2, 4, 6].

General Scheme for Partial Maps

```
partial([], []).
partial([X|T], [X|L]) :- include(X),
                         partial(T, L)
partial([X|T], L) :- partial(T, L).
```

For example:

```
include(X) :- X >= 0. % a condition
```

```
?- partial([-1, 0, 1, -2, 2], X),
```

X = [0, 1, 2].

Partial Maps

A program that 'censors' an input list, by making a new list in which certain prohibited words do not appear.

predicate prohibit(X) succeeds if X is a censored word.

For example,

```
prohibit(france).
prohibit(irland).
prohibit(turkey).
prohibit(hungary).
```

example

```
censor([], []).
censor([H|T], T1) :- prohibit(H),
                     censor(T, T1).

censor([H|T], [H|T1]) :- censor(T, T1).
```

Remove from a list

```
rem([H|T],H,T).  % rem 2nd arg from 1st arg
rem([H|T],X,[H|Y]) :- rem(T,X,Y).

?- rem([1,2,3],3,Res).
Res = [1, 2]
false.

?- rem([1,2,3],X,Res).% semicolon gets all options
X = 1,
Res = [2, 3]
X = 2,
Res = [1, 3]
X = 3,
Res = [1, 2]
false.
```

list bazar

First, last elements of a list

```
first([H|_],H).

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

What will `last([],X).` return ?

```
1- X = [], true
2- X = ???, true
3- pattern-match exception
4- false
```

Get all permutations of a list

```
perm([],[]).
perm(List,[H|T]):-rem(List,H,X),  perm(X,T).
```

Ex: order the list: [i,you,he,you,i], to be [i,i,you,you,he]

How2: generate solutions, and test them

(1) Generate a solution, Test it

(2) If not, backtrack to the next generated solution

```
namesRow(In,Out)  :-  perm(In,Out),
                      checkNamesRow(Out).
```

Removing Duplicates

```
setify([], []).
setify([X|T], L) :- member(X,T),
                    setify(T, L).
setify([X|T], [X|L]) :- setify(T, L).
```

Prefix רישא

```
prefix([],_).
prefix([H|T1],[H|T2]):-prefix(T1,T2).

?- prefix(L,[1,2,3,4]).
L = []
L = [1]
L = [1, 2]
L = [1, 2, 3]
L = [1, 2, 3, 4]
false.
```

suffix

```
suffix(S,S).
suffix([H|T],L):-suffix(T,L).
```

What are the L's such that [1,2,3,4] is their suffix?

```
?- suffix(L,[1,2,3,4]).
L = [1, 2, 3, 4]
L = [_G3281, 1, 2, 3, 4]
L = [_G3281, _G3284, 1, 2, 3, 4]
L = [_G3281, _G3284, _G3287, 1, 2, 3, 4]
L = [_G3281, _G3284, _G3287, _G3290, 1,2, 3, 4] .

?- suffix([1,2],L).
L = [1, 2]
L = [2]
L = [].
```

N-th element

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

nth_member(1,[M|_],M).
nth_member(N,[_|T],M):-N>1, N1 is N-1,
                       nth_member(N1,T,M).

?- nth_member(3,[a,b,c,d,e],Res).
Res = c .
```

append is the new salt

```
append([],L,L).
append([H|T],X,[H|Y]):-append(T,X,Y).
```

Define prefix, suffix with append:

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

sublist

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

?- sublist([6],[1,2]).
false.

?- sublist([2],[1,2]).
true .

?- sublist(Res,[1,2]).
Res = []
Res = [1]
Res = [1, 2]
Res = []
Res = [2]
Res = []
false.
```

delete

Args in Diff order than prev rem:

```
delete(X,[X|T],T).
delete(X,[Y|T],[Y|Z]):-delete(X,T,Z).

?- delete(2,[1,3,4,2,5],Res).
Res = [1, 3, 4, 5]
?- delete(2,X,Res).
X = [2|Res]
X = [_G1078, 2|_G1082],
Res = [_G1078|_G1082]
X = [_G1078, _G1084, 2|_G1088],
Res = [_G1078, _G1084|_G1088]
X = [_G1078, _G1084, _G1090, 2|_G1094],
Res = [_G1078, _G1084, _G1090|_G1094] .
```

delete examples

```
?- delete(X,Y,Res).
Y = [X|Res]
Y = [_G1093, X|_G1097],
Res = [_G1093|_G1097]
Y = [_G1093, _G1099, X|_G1103],
Res = [_G1093, _G1099|_G1103]
Y = [_G1093, _G1099, _G1105, X|_G1109],
Res = [_G1093, _G1099, _G1105|_G1109] .
```

append a list to a reverted list

```
rev_append([2,1],[3,4],[1,2,3,4]).

rev_append([],L,L).
rev_append([H|T],L,X):-rev_append(T,[H|L],X).

?- rev_append(X,[1,2,3],Res).
X = [],
Res = [1, 2, 3]
X = [_G2293],
Res = [_G2293, 1, 2, 3]
X = [_G2293, _G2299],
Res = [_G2299, _G2293, 1, 2, 3] .
```

delete2

-delete2 is defined using accumulator.

```
delete2(X,L,Z):-del2(X,L,[],Z).


del2(X,[X|T],Ac,Z):-rev_append(Ac,T,Z).
del2(X,[Y|T],Ac,Z):-del2(X,T,[Y|Ac],Z).
```

Ac acts a as a stack here.

The rev_append implements the FILO wars

?- delete2(X,[1,2,3],Res).

X = 1,

Res = [2, 3]

X = 2,

Res = [1, 3]

X = 3,

Res = [1, 2]

false.

revert a given list.

adding an elem to the end of list using append in each step is not effective:

```
revert([],[]).
revert([H|T],Rv):-revert(T,X),
                    append(X,[H],Rv).

?- revert([1,2],Res).
Res = [2, 1].

?- revert(X,[1,2]).
X = [2, 1]
…
```

revert2, uses an accum, is more efficient.

```
revert2(L,X):-rev_acc(L,[],X).
% reverted list is in Ac which contains part of list reverted until now

rev_acc([],Ac,Ac).
rev_acc([H|T],Ac,Rv):-rev_acc(T,[H|Ac],Rv).

?- revert2(X,Y).
X = Y, Y = []
X = Y, Y = [_G1477]
X = [_G1477, _G1483],
Y = [_G1483, _G1477] .

?- revert2([1,2],Y).
Y = [2, 1].

?- revert2(Y,[1,2]).
Y = [2, 1] .
```

naive sort

Naive sort is not very efficient algorithm. It generates all permutations and then it tests if the permutation is a sorted list.

```
naive_sort(List,Sorted):-
perm(List,Sorted),is_sorted(Sorted).
is_sorted([]).
is_sorted)[_]).
is_sorted([X,Y|T]):-X=<Y, is_sorted([Y|T]).
```

Naive sort uses the **generate and test** approach to solving problems which is usually utilized in case when everything else failed. However, sort is not such case.

insert sort

With an of accumulator:

```
insert_sort(List,Sorted):-
                    i_sort(List,[],Sorted).
i_sort([],Acc,Acc).
i_sort([H|T],Acc,Sorted):-
                     insert(H,Acc,NAcc),
                     i_sort(T,NAcc,Sorted).

insert(X,[Y|T],[Y|NT]):-X>Y, insert(X,T,NT).
insert(X,[Y|T],[X,Y|T]):- X=<Y.
insert(X,[],[X]).
```

merge sort

```
merge_sort([],[]).   % empty list is already sorted
merge_sort([X],[X]). % single element list is already sorted
merge_sort(List,Sorted):-  List=[_,_|_],
                           divide(List,L1,L2),
% list with at least two elements is divided into two parts
                           merge_sort(L1,Sorted1),
                           merge_sort(L2,Sorted2),
% then each part is sorted
                           merge(Sorted1,Sorted2,Sorted).
% and sorted parts are merged

merge([],L,L).
merge(L,[],L):-L\=[].
merge([X|T1],[Y|T2],[X|T]):-X=<Y,
                                merge(T1,[Y|T2],T).
merge([X|T1],[Y|T2],[Y|T]):-X>Y,merge([X|T1],T2,T).
```

Backtracking, cuts & negation

Backtracking is basically a form of searching.

Suppose Prolog is trying to satisfy a sequence of goals:

goal_1, goal_2.

When the Prolog interpreter finds a set of var bindings which allow *goal_1* to be satisfied,

-it commits itself to those bindings, and

-then seeks to satisfy *goal_2*.

Eventually one of two things happens:

(a) *goal_2* is satisfied, and finished with; or

(b) *goal_2* cannot be satisfied.

In either case, Prolog might backtrack:

It "un-commits" itself to the var bindings it made in satisfying *goal_1,* and goes looking for a *different* set of var bindings that allow *goal_1* to be satisfied.

Uses of Backtracking

If it finds a 2nd set of such bindings, it commits to them, and proceeds to try to satisfy *goal_2* again, with the new bindings.

- In case (a), the Prolog interpreter is looking for *extra* solutions,
- in case (b) it is *still* looking for the *first* solution.

So backtracking may serve:

- to find extra solutions to a problem, or
- to continue the search for a first solution, when a first set of assumptions (i.e. variable bindings) turns out not to lead to a solution.

Backtracking Choice-points:

Choice-Points:

In `member(X, [a, b, c])` , var `X` can have (match to) 3 choices: a,b,c

Choice-points are "provided" by any subgoals that can be satisfied with *more than one* match.

Backtracking: during goal execution Prolog *keeps track* of choice-points.

Why: if a certain path fails, it returns to the *last choice-point* and tries the next match.

Inefficiency ?

Backtracking is a characteristic feature of Prolog

But backtracking can lead to inefficiency:

– Prolog can waste time and memory exploring possibilities that lead nowhere

– It would be nice to have some control

select(El,List1,List2)

(another version of rem, delete …)

Succeeds if List2 is List1 less an occurrence of El in List1.

```
select(A, [A|B], B).
select(A, [B,C|D],[B|E]):-select(A, [C|D],E).
```

(all args are vars)

```
?-select(1,[1,2],[2]).
true.

?- select(1,[1,2],[2,3]).
false.
```

use of backtracking

```
perm([],[]).
perm(List, [Element | P]) :-
               select(Element, List, Rest),
               perm(Rest, P).
```

Given a list as the 1st arg position, `perm/2` generates all possible permutations of that list in the 2nd argument, by backtracking - if the user presses **;** after every solution

Example

```
?- permutation([1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false
```

Problems with Backtracking

- Asking for alternative solutions generates *wrong* answers for this predicate definition:

% choice-points

```
rem_dupl([], []).
rem_dupl([Head | Tail], Result) :-
                             member(Head, Tail),
                             rem_dupl(Tail, Result).

rem_dupl([Head |Tail], [Head | Result]) :-
                             rem_dupl(Tail, Result).
```

Example

```
?- rem_dupl([1, 2, 2, 3, 1], List).
List = [2, 3, 1] ;
List = [2, 2, 3, 1] ;
List = [1, 2, 3, 1] ;
List = [1, 2, 2, 3, 1] ;
```

using fail

- Look at the following predicate:

```
show(List) :- member(El, List), % choice-points
              write(El),
              nl,
              fail.
```

`fail/0` is a built-in predicate that *always fails*.

Use:

```
?- show([ship, plane, car]).
ship
plane
car
false
```

fail

- The fail causes Prolog to backtrack.
- The only choice-point is at `member(El, List).`
- In every backtracking-cycle a new element of List *is matched with the variable El.*
- At the end the query fails (as it intended to)

Stop the backtracking with operator cut

- to prevent Prolog from backtracking into certain choice-points, either because of bad alternatives, or for efficiency reasons.

- Can be done with op `cut`, (`!`).

- `!` prevents Prolog from backtracking into subgoals placed *before the cut inside the same rule body*

- cut always succeeds,

Correct Example with cut

- The correct program for removing duplicates from a list:

```
rem_dupl([], []).
rem_dupl([Head | Tail], Result) :-
                             member(Head, Tail), !,
                             rem_dupl(Tail, Result).

rem_dupl([Head| Tail], [Head | Result]) :-
                             rem_dupl(Tail, Result).
```

Smart list pro

combinations

Combination is an arbitrary subset of the set containing a given number of elements.

Order of elements is irrelevant.

```
comb(0,_,[]).
comb(N,[X|T],[X|Cmb]):-N>0,N1 is N-1,
                        comb(N1,T,Cmb).
comb(N,[_|T],Cmb):-N>0, comb(N,T,Cmb).
```

How comb works ?

Lets add some snooping devices:

```
comb(0,_,[]).
comb(N,[X|T],[X|Cmb]):-N>0,N1 is N-1,write(N1),write(' one '),comb(N1,T,Cmb).
comb(N,[_|T],Cmb):-N>0, write(N),write(' two '),comb(N,T,Cmb).
```

```
?- comb(3,[a,b,c,d],X).
2 one 1 one 0 one
X = [a, b, c] ;
1 two 0 one
X = [a, b, d] ;
1 two 2 two 1 one 0 one
X = [a, c, d] ;
1 two 2 two 1 one 2 two 3 two 2 one 1 one 0 one
X = [b, c, d] ;
1 two 2 two 1 one 2 two 3 two 2 one 1 one 2 two 3 two 2 one 3 two
false.
```

comb2

It is possible to program generator of combinations without the arithmetics.

comb2 assumes the list with N free variables as its second argument and it binds these variables.

Use :

```
?-comb2([1,2,3,4],[X,Y]).
```

Here to generate combinations with two elements.

```
comb2(_,[]).
comb2([X|T],[X|Comb]):-comb2(T,Comb).
comb2([_|T],[X|Comb]):-comb2(T,[X|Comb]).
```

combinations with repeated elements

This type of combination can contain an element more times. Thus, it is not a set but a *multi-set.*

```
comb_rep(0,_,[]).
comb_rep(N,[X|T],[X|RCmb]):-N>0, N1 is N-1,
                              comb_rep(N1,[X|T],RCmb).
comb_rep(N,[_|T],RCmb):-N>0,
                              comb_rep(N,T,RCmb).
```

variations

Variation is a subset with given number of elements.

The order of elements in variation is *significant*.

```
varia(0,_,[]).
varia(N,L,[H|Vr]):-N>0,N1 is N-1,
                   delete(H,L,Rest),
                   varia(N1,Rest,Vr).
```

variations with repeated elements

Again, this type of variation can contain repeated elements.

```
varia_rep(0,_,[]).
varia_rep(N,L,[H|RVaria]):-N>0, N1 is N-1,
                           delete(H,L,_),
                           varia_rep(N1,L,RVaria).
```