SEMINAR 4 – Backtracking in Prolog

"There and back again" - Hobbit

Backtracking – the facility of Prolog to determine all solutions for a problem.
In this case, we will have predicates with more than one solution => **non-deterministic predicates.**

Until now, all our predicates were mostly **deterministic predicates** – predicate with just **one** solution.
To collect all solutions of a predicate, we have the built-in predicate **findall** .
The built-in predicate* findall *collects all solutions of a predicate and puts them in a single list.

findall (ResPredicatePartial, PredicatePartial (InitialList, ResPredicatePartial), FinalResult).

Eg. If we already have a predicate for onesolution (L, ROS) , *L - initial list, ROS - resulted list and flow model (i, o),* then collecting all solution in a **main predicate** called allsolutions(L, RALL) , *L - initial list, RALL – resulted list and flow model (i, o),* we will use **findall** as follows:

allsolutions (L, RALL) :- findall (ROS, onesolution (L, ROS), RALL).

Problems

1. ***Given a list L, generate the list of all arrangements of K elements from the list that have product P and sum S. Ex. L=[1,2,3,10], K=2, P=30, S=13 then result R = [[3, 10], [10, 3]].***

```
% Produsul elementelor unei liste utilizand varaibila colectoare
% produs(L-list, C-colector var, P-produs rezultat)
% produs (i,i,o)

produs([], C, C).
produs([H | T], C, P) :-
    P1 is C * H,
    produs(T, P1, P).

% Suma elementelor unei liste utilizand variabila colectoare
% suma (L-list, SC-colector var, S-suma rezultat)
% suma (i,i,o)
suma([],SC,SC).
suma([H | T], SC, S):-
        SC1 is SC+H,
        suma(T,SC1,S).
```

```
% perm(l1l2..ln)= { []  , if n=0
%                 { insert(l1, perm(l2..ln) ) , otherwise

% insert(e, l1l2..ln) ={ e U l1l2..ln
%                       { l1 U instert(e, l2..ln)
```

```
%insert an element in a list
%insert(E -element, L -list, R - result list)
%insert(i,i,o)
insert(E, L, [E|L]).
insert(E, [H|T], [H|R]) :-
   insert(E, T, R).
```

```
%permutations of a linear list
%uses insert
%perm(L-list, R - result list)
%perm(i,o)
perm([], []).
perm([H|T], P) :-
   perm(T, R),
   insert(H, R, P).
```

```
%combinations of K elements from list L (with n elems, 1<=K<=N)
%comb(l1l2..ln, K)={ l1 , if k=1, n>=1
%                  { comb(l2..ln, K)  , k>=1
%                  { l1 U comb(l2..ln, K-1), k>1
```

```
% arr(l1l2..ln, K)={ l1 , if k=1
%                 { arr(l2..ln, K)  , if k>=1
%                 { insert(l1, arr(l2..ln, K-1))        , if k>1
```

```
%arrangements of K elements from a list L
%uses insert
%arr(L-list, K-number of elements, R -resulted list)
%arr(i,i,o)
arr([E|_], 1, [E]).
arr([_|T], K, R) :-
   arr(T, K, R).
arr([H|T], K, R1) :-
   K > 1,
   K1 is K-1,
   arr(T, K1, R),
   insert(H,R,R1).
```

```
%combinations of K elements from a list L
%comb(L-list, K-number of elements, R -resulted list)
%comb(i,i,o)
comb([E|_], 1, [E]).
comb([_|T], K, R) :-
   comb(T, K, R).
comb([H|T], K, [H|R]) :-
   K > 1,
   K1 is K-1,
   comb(T, K1, R).
```

```
%arrangements by creating permutations of combinations
lazyarr(L,K,R):-
    comb(L,K,A),
    perm(A,R).
```

```
?- findall(R,insert(11, [1,2,3], R), RL).
RL = [[11, 1, 2, 3], [1, 11, 2, 3], [1, 2, 11, 3], [1, 2, 3, 11]]
```

```
?- findall(S, perm([1,2,3], S), RL).
RL = [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

```
?- findall(S, comb([1,2,3,4],3, S), RL).
RL = [[2, 3, 4], [1, 3, 4], [1, 2, 3], [1, 2, 4]]
```

```
?- findall(X, arr([1,2,3,4],2, X), R).
R = [[3, 4], [4, 3], [2, 3], [3, 2], [2, 4], [4, 2], [1, 2], [2, 1], [1, 3], [3, 1], [1, 4], [4, 1]]
```

```
% Determinarea unei solutii pentru problema noastra
% onesol (L-lista, K-nr elem, P-valoare produs, R-lista rez)
% onesol (i,i,i,o)
onesol(L, K, P, S, RL) :-
    arr(L, K, RL),
    produs(RL, 1, P),
    suma(RL,0,S).

% Determinarea tuturor solutilor intr-o lista rezultat
% prin utilizarea predicatului predefinit FINDALL
% allsols (L-list, K-nr elem, P-produs, S-suma, R-list rez)
% allsols (i,i,i,i,o)
allsols(L, K, P, S, R) :-
        findall(RL, onesol(L, K, P, S, RL), R).
```

2. ***We are given a sequence $a_1...a_n$ composed of distinct integers numbers. We have to display all the subsequences which have a valley aspect. For example, for the list [5, 3, 4, 2, 7, 11, 1, 8, 6], some of the solutions would be [5, 4, 3, 11], [3, 2, 1, 4, 5, 7, 8], [11, 6, 1, 3, 4, 5], etc.***

- A very inefficient solution is to solve the problem using 2 predicates: one to generate all possible sub-sequences and one to check if a subsequence is a valley or not (*** see at the end of the document). This solution is inefficient because there are many possible candidates and most of them are not valleys. For the list above, there are 986328 possible candidates, but „only" 8828 are correct solutions.

Solution 1:

```
% subsets of a list
% subs(l1l2..ln)= { [], if n=0
%                 { l1 U subs(l2..ln) , n>0
%                 { subs(l2..ln)

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

```
% 1 decreasing, 0 increasing
% valley(l1l2..ln, flag)={ valley(l2..ln, 1) , if l1>l2, flag=1
%                        { valley(l2..ln, 0) , if l1<l2, and (flag=0 or flag=1)
%                        { true , n=1, flag=0

%mainValley(l1l2..ln)=valley(l1l2..ln, 1), if n>=3, l1>l2
%valley(L-list, F-flag)
%floe(i,i)
valley([_],0).
valley([H1,H2|T],1):-
    H1>H2,
    valley([H2|T],1).
valley([H1,H2|T], _):-
    H1<H2,
    valley([H2|T],0).

mainV(L):-
    L=[H1,H2|_],
    H1>H2,
    valley(L,1).

onesol(L,SP):-
    subs(L,SS),
    perm(SS,SP),
    mainV(SP).

allsols(L,R):-
    findall(X, onesol(L,X),R).
```

Solution 2a:

- If we have at a given point the subsequence [5, 3, 11] (which is a valley), there are no other elements we could add to the end of this subsequence, so we should not generate them at all.
- In order to have a more efficient solution, we will combine the generation and the verification steps. We will use a candidate list (a collector variable) in which we will keep the solution built until now. We will add elements one-by-one in this list, but only those elements that can lead us to a correct solution.
- To generate a valley, we need a decreasing sequence of elements, followed by an increasing sequence of elements. When we add an element to the candidate solution we need to know whether we are on the decreasing or increasing side (this will decide if an element can be added or not). This is why we are going to use an extra parameter (let's call it Direction), which will tell us the current direction:

- 0 – for the decreasing part
- 1 – for the increasing part

- The candidate list is in fact a collector variable. In general, we do not like to use collector variables when the result has to be a list, because we need that ***addToEnd*** function, to add the elements to the end of the list. In our case we can avoid implementing the ***addToEnd*** function if we add the elements to the beginning of the collector variable. But this means that we will generate the list in the reverse order. For example:
  - [7]
  - [6, 7]
  - [3, 6, 7]
  - [4, 3, 6, 7]
  - [5, 4, 3, 6, 7]

- When we want to add a new element to the candidate list, we have to consider two things: the direction and the first element of the candidate list (which is the last element that was added in the collector variable). This leads to 4 possible cases:
  - Direction 1 and element to be added less than the first element of the collector variable (ex. 8 to be added in front of [9, 11]) => add the element and keep the Direction 1.
  - Direction 1 and element to be added greater than the first element of the collector variable (ex. 8 to be added in from of [7, 11]) => add the element and make the Direction 0.
  - Direction 0 and element to be added greater than the first element of the collector variable (ex. 8 to be added in front of [7, 5, 6]) => add the element and keep Direction 0.
  - Direction 0 and element to be added less than the first element of the collector variable (ex. 8 to be added in front of [9, 5, 6]) => impossible

- When is the candidate list a solution? To have a valid valley, we need a decreasing and an increasing sequence. First we generate the increasing sequence, so whenever we are at the decreasing sequence part (meaning the direction is 0) we have a solution. But, in the same time, it is possible that we can also extend this solution to generate other solutions.
- How do we generate the elements to be added in the candidate solution? ***We need a predicate to return us, one-by-one, all the elements from the original list. It is a non-deterministic predicate***

$$element(l_1 l_2 \ldots l_n) = \begin{cases} l_1 \\ element(l_2 \ldots l_n) \end{cases}$$

```
% element(L:list, E: element)
% flow model: (i,o), (i,i)
% L – initial list
```

```
% E - an element from the list
element([H|_], H).
element([_|T], E):-
   element(T, E).
```

We are not allowed to add the same element twice in the candidate solution. How can we check this? We can use the *element* predicate for this as well, but with a different flow model (i,i)

Another solution would be to make the *element* predicate return an element and the list without that element. How would we change it to do this?

$$element(l_1 l_2 \dots l_n) = \begin{cases} (l_1, l_2 \dots l_n) \\ (e, l_1 \cup L), if\ (e, L) = element(l_2 \dots l_n) \end{cases}$$

```
%element2(L:List, E: element, R: list)
%flow model (i, o, o)
%L - initial list
%E - an element from the list
%R - the initial list without the element E
element2([H|T], H, T).
element2([H|T], E, [H|Rez]):-
    element2(T, E, Rez).
```

Now we can write the predicate to generate the solutions. It will have the following parameters:

- ***L – the input list***
- ***C – the collector variable***
- ***D – the current direction***
- ***R – result (only in Prolog).***

$$generate(L, c_1 \dots c_n, D) = \begin{cases} c_1 \dots c_n, if\ D = 0 \\ generate(Rest, E\ U\ c_1 .. c_n, 0), if\ (E, Rest) = element\ 2(L), E > c_1 \wedge D = 0 \\ generate(Rest, E \cup c_1 \dots c_n, 1), if\ (E, Rest) = element\ 2(L), E < c_1 \wedge D = 1 \\ generate(Rest, E \cup c_1 \dots c_n, 0), if\ (E, Rest) = element\ 2(L), E > c_1 \wedge D = 1 \end{cases}$$

```
generate(L:list, C:list, Direction:integer, R:list)

      %flow model (i,i,i,o)
      % L – initial list
      % C – candidate solution
      % Direction – 0 for the decreasing part 1 for the increasing part
      % R - result
```

```
generate(_, Cand, 0, Cand).
generate(L, [H|Cand], 0, R):-
      element2(L, E, Rest),
      E  > H,
      generate(Rest, [E,H|Cand], 0, R).
generate(L, [H|Cand], 1, R):-
      element2(L, E, Rest),
      E < H,
      generate(Rest, [E,H|Cand], 1, R).
generate(L, [H|Cand], 1, R):-
      element2(L, E, Rest),
      E > H,
      generate(Rest, [E,H|Cand], 0, R).
```

We need a predicate to call *generate*, let's call it *start*. Since the three recursive clauses require the division of the candidate solution into first element and rest of the list, we cannot start with an empty list. We will need to generate an element (using the *element2* predicate) and to call *generate* having as candidate list the list made of that element.

If we implement and run *start*, we will see that it returns incorrect solutions as well. It will generate lists which have only the decreasing part. In order to avoid this problem, we will actually generate the first two elements of the candidate list, make sure that they are in the correct order (increasing) and only after that will we call *generate*.

$start(L) = generate(Rest, [E_1, E_2], 1), if(E_1, R1) = element2(L), (E¿¿2, Rest) = element2(R1), E_1 < E_2¿$

```
%start(L:list, Rez: list)
%flow model (i,o) (i,i)
%L – initial list
%Rez – one solution
start(L, Rez):-
      element2(L, E1, Rest),
      element2(Rest, E2, Rest2),
      E1 < E2,
      generate(Rest2, [E1,E2], 1, Rez).
```

The *start* predicate generates the solutions one after the other, and for seeing the next solution we have to press ; after every solution. If we want to have a list with all the solutions, we can use the *findall* predicate from Prolog.

$startAll(l_1 l_2 \ldots l_n) = start(l_1 l_2 \ldots l_n)$

```
%startAll(L:list, Rez: list)
%flow model (i,o) (i,i)
%L – initial list
%Rez – list with all the solutions
startAll(L, Rez):-
      findall(R, start(L, R), Rez).
```

Solution 2b:

The final solution using the same notation as the first one with valley (1 decreasing, 0 increasing) like the one solved during the seminar.

```
% element2(l1l2..ln)={ (l1 , l2..ln)
%                     { (e, l1 U Rest), if  element2(l2..ln)= (e, Rest)
%flow i, o, o
element2([H|T], H, T).
element2([H|T],E, [H|Rest]):-
    element2(T,E, Rest).


% 1 for decreasing, 0 for increasing
%generate(l1l2..ln, c1c2..cm, Flag)={ c1c2..cm, if flag=1
%           { generate(Rest, e U c1c2..cm,0) , if flag=0, e<c1, element2(L)=(e, Rest)
%           { generate(Rest, e U c1c2..cm,1) , if (flag=0 or 1), e>c1, element2(L)=(e, Rest)

%start(l1l2..ln)= generate(Rest2, [e1,e2],0) , if  element2(L)=(e1, Rest),
%                                                  element2(Rest)=(e2, Rest2), e1<e2


%generate(L-list, Cand -list, Flag -number, R-list)
generate(_, Cand, 1, Cand).
generate(L, [C1|Cand], 0, R):-
    element2(L, E, Rest),
    E<C1,
    generate(Rest, [E,C1|Cand], 0, R).
generate(L, [C1|Cand], _, R):-
    element2(L, E, Rest),
    E>C1,
    generate(Rest, [E,C1|Cand] ,1, R).

start(L, R):-
    element2(L, E1, Rest),
    element2(Rest, E2, Rest2),
    E1<E2,
    generate(Rest2, [E1, E2], 0, R).

allSol(L,R):-
    findall(X, start(L,X), R).
```

Solution 3:

What if we wanted to have in the solution the elements in the same order in which we have them in the initial list? How should we modify our solution to do this?

When we add an element to the candidate solution, the next elements that will be added in the candidate solution will be placed in front of it. And if we want to keep the original order of element, we need to make sure that the next element to be added will be generated only out of the elements that are in front of the current element in the list. So if element2([5, 3, 4, 2, 7, 11, 1, 8, 6]) returns the element 7, we need to make sure that the next element will be generated only from the list [5, 3, 4, 2]. To do this, we will modify the element2 predicate, to return an element and the list of element before it.

$$element3(l_1 \ldots l_n) = \begin{cases} (l_1, \emptyset) \\ (E, l_1 \cup Rest), if\ (E, Rest) = element3(l_2 \ldots l_n) \end{cases}$$

```
%element3(L:list, E:element, R:list)
%flow model (i,o,o), (i,i,i)
%L - initial list
%E - an element from the list L
%R - the list of elements until element E
element3([E|_], E, []).
element3([H|T], E, [H|R]):-
      element3(T,E,R).
```

Solution 4:

*** The inefficient solution: generate subsets (n! where n length of list), permutate them (s solutions where s is the length of the subset), check the ones with valley aspect (**different method**). Write a program to compute the time difference in running the two solutions and compare the results. **We can skip perm in findsol if we want to keep the order or elements in subsets from the initial list.**

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

ins(E, [], [E]).
ins(E, [H|T], [E,H|T]).
ins(E,[H|T], [H|Tr]):-
    ins(E,T,Tr).

perm([],[]).
perm([H|T], R):-
    perm(T,RT),
    ins(H,RT,R).
```

```
increase([H1,H2]):- !, H1<H2.
increase([H1,H2|T]):-
        H1 < H2,
        increase([H2|T]).

decrease([H1,H2|T]):-
    H1>H2, !,
    decrease([H2|T]).
decrease(L):-
    increase(L).

valley(L):-
    L=[H1,H2|T],
    H1 > H2,
    decrease([H2|T]).
```

```
findsol(L, SP):-
    subs(L,S),
    perm(S,SP),
    valley(SP).


main(L,X):-
    get_time(T1),
    findall(R, findsol(L,R),X),
    get_time(T2),
    T is (T2-T1),
    write("Seconds:"),
    write(T).
```

3. ***The problem of the 3 houses (from the course).***
    a. ***The Englishman lives in the first house on the left.***
    b. ***In the house on the right to the house where the wolf lives, the owner smokes Luck Strike.***
    c. ***The Spanish smokes Kent.***
    d. ***The Russian has a horse.***

Who smokes LM? Who owns the dog?

The problem has 2 solutions:

1	English	Spanish	Russian
	Dog	Wolf	Horse
	LM	Kent	Lucky Strike
2	English	Russian	Spanish
	Wolf	Horse	Dog
	LM	Lucky Strike	Kent

Problem codified with triplets (N, A, T) for nationality, animal and tobacco, where:

- N ∈ { eng, spa, rus}
- A ∈ {dog, wolf, horse}
- T ∈ {LM, LuckyStrike, Kent}

We will use the following predicates:

- Solve(N,A,T) (o,o,o) to generate a solution to the problem
- Candidate(N,A,T)(o,o,o) to generate all candidates for a solution
- Constraints(N,A<T)(i,i,i) to verify if a candidate satisfies the restrictions
- Perm(L, L1) (i,o) that generates permutations of list L

```
% ins (i,i, o)
ins(E, L, [E|L]).
ins(E, [H|L], [ H|T]) :-
    ins(E,L,T).

% perm (i,o)
perm([], []).
perm([H|T], L):-
    perm(T, P),
    ins(H, P, L).

%(o,o,o)
```

```
candidate(N, A, T) :-
	perm([eng, spa, rus], N),
	perm([dog, wolf, horse], A),
	perm([lm, kent, ls],T).

% (i,i,i)
constraints(N, A, T) :-
	aux(N, A, T, eng ,_ , _, 1),
	aux(N, A, T, _, wolf, _, Nr),
	toright(Nr, M),
	aux(N, A, T, _, _, ls, M),
	aux(N, A, T, spa, _, kent, _),
	aux(N, A, T, rus, horse, _, _).

% right - (i,o)
toright(I,J) :- J is I+1.

% aux (i,i,i,o,o,o,o)
% houses, numbered 1,2,3 from left to right
aux([N1 ,_ ,_], [A1, _, _], [T1, _, _], N1, A1, T1, 1).
aux([_, N2, _], [_, A2, _], [_, T2,_], N2, A2, T2, 2).
aux([_, _, N3], [_, _, A3], [_, _, T3], N3, A3, T3, 3).

% (o,o,o)
solve(N, A, T) :-
	candidate(N, A, T),
	constraints(N, A, T).
```

Optional Homework:

a. Implement a Prolog program to find solutions for the 5 house version zebre puzzles: https://www.brainzilla.com/logic/zebra/

b. Same for Sudoku (you might still find magazines that will give you a prize if you find the solution for a hard one :D) https://www.brainzilla.com/logic/sudoku/

Example implementation for Inspiring Women Puzzle: https://github.com/alinacalin/Scripts-Portfolio/blob/main/Prolog/InspiringWomenPuzzle.pl