SEMINAR 2 – Lists in Prolog

"Try not. Do or do not. There is no try." – Yoda, Star Wars

Problems

1. ***Compute number of occurrences of an element in a list.***

$$nrOccurrences(l_1 l_2 \ldots l_n, e) = \begin{cases} 0, if\ n = 0 \\ 1 + nrOccurences(l_2 l_3 \ldots l_n, e), if\ l_1 = e \\ nrOccurences(l_2 l_3 \ldots l_n, e), otherwise \end{cases}$$

```
%nOcc(L-list of elements, E-element, N-number of occurences)
%flow model (i i i), (i i o)
nOcc([],_,0).
nOcc([H|T],E,N):-
        H=E,
        nOcc(T,E,N1),
        N is N1+1.
nOcc([H|T],E,N):-
        H\=E,
        nOcc(T,E,N).

%nOcc([2,3,4,2],2,N).
```

- There is another way of solving this problem (true for almost every problem), using an **accumulator (collector variable)**, which is an extra parameter representing the result of the function.

$$nrOccurrencesC(l_1 l_2 \ldots l_n, e, res) = \begin{cases} res, if\ n = 0 \\ nrOccurencesC(l_2 l_3 \ldots l_n, e, res + 1), if\ l_1 = e \\ nrOccurencesC(l_2 l_3 \ldots l_n, e, res), otherwise \end{cases}$$

- Will the collector variable be an input or an output parameter? We need it to start from the value 0. Since it is a parameter that we initialize with a specific value, it is an input parameter. *nCV* is the implementation for the *nrOccurencesC* predicate.

```
%nCV(L-initial list, E-element, C-collector variable, R-resulted list)
%flow model: nCV(i i i i); nCV(i i i o)
nCV([],_,C,C).
nCV([H|T],E,C,R):-
        H=E,
        C1 is C+1,
        nCV(T,E,C1,R).
nCV([H|T],E,C,R):-
        H\=E,
        nCV(T,E,C,R).
%nCV([2,3,4,5,2,2,1],2,0,R).
```

Trace execution for each of the two versions: first goes deep in recursion, then computes the result as it returns from recursion; second version goes deep in recursion and returns the result there. Use commands **trace.** (and **notrace.** when you finish**)** in Prolog to run queries in trace mode.

2. ***Write a predicate to remove from a list all the elements that appear only once. For example: for [1,2,1,4,1,3,4] the result will be [1,1,4,1,4].***

- How do we determine if an element appears only once? We need a predicate to count the number of occurrences of an element in a list.
- In order to have the correct answer, nrOccurrences has to be called for the initial list, instead of the list from which we kept eliminating elements during the recursive call.

$$remove(l_1 l_2 \dots l_n, L_1, L_2 \dots L_m) = \begin{cases} \emptyset, if\ n = 0 \\ remove(l_2 \dots l_n, L_1 L_2 \dots L_m), if\ \text{nrOccurrences}(L_1 L_2 \dots L_n, l_1) = 1 \\ l_1 \cup\ remove(l_2 \dots l_n, L_1 L_2 \dots L_m), otherwise \end{cases}$$

- We need another function to initialize the copy of the original list

$$removeMain(l_1 l_2 \dots l_n) = \ remove(l_1, l_2 \dots l_n, l_1 l_2 \dots l_n)$$

```
%we use the predicate nOcc from the previous problem
nOcc([],_,0).
nOcc([H|T],E,N):-
        H=E,
        nOcc(T,E,N1),
        N is N1+1.
nOcc([H|T],E,N):-
        H\=E,
        nOcc(T,E,N).


% el(L-initial list, LC-copy of the initial list, R-resulted list)
% flow model: (i i i), (i i o).
el([],_,[]).
el([H|T],LC,R):-
        nOcc(LC,H,N),
        N=1,
        el(T,LC,R).
el([H|T],LC,[H|R]):-
        nOcc(LC,H,N),
        N>1,
        el(T,LC,R).

%main
elM(L,R):-el(L,L,R).
```

- We can use a **collector variable** to rewrite our remove predicate as well. There is one problem with this version, however: our collector variable is going to be a list, in which we will add elements one by one, after checking whether the element has to be added or not (depending on the number of occurrences). If we put the elements to the beginning of the list (where we can easily add elements), our result will be reversed. In order to have the elements in the right order, we need to put every element to the end of the collector variable, but for this we need another function: *addToEnd.*

- Let's assume that we already have this predicate, with the following header and flow model:
    - *addToEnd(List, Elem, Result), (i, i, o)*

- For counting the occurrences we can use nrOccurrences or nrOccurrencesC. We will use nrOccurrencesC

$$removeC(l_1 l_2 \dots L_n, L_1, L_2 \dots L_m, C_1 C_2 \dots C_k) = \begin{cases} C_1 C_2 \dots C_k, if\ n = 0 \\ removeC2(l_2 \dots l_n, L_1 L_2 \dots L_m, C_1 C_2 \dots C_k), if\ \text{nrOccurrencesC}(L_1 L_2 \dots L_m, l1, 0) = 1 \\ removeC2(l_2 \dots l_n, L_1 L_2 \dots L_m, addToEnd(C_1 C_2 \dots C_k, l1)), otherwise \end{cases}$$

```
% removeC(L:list, LO:list, Col:list, R:list)
% flow model: (i,i,i,o) or (i,i,i,i)
% L - list from which we remove elements that occur only once
% LO - copy of the initial list, to count occurrences
% Col - collector variable
% R - resulting list

removeC([], _, Col, Col).
removeC([H|T], LO, Col, R):-
    nrOccurrencesC(LO, H, 0, S),
    S = 1,
    removeC(T, LO, Col, R).
removeC([H|T], LO, Col, R):-
    nrOccurrencesC(LO, H, 0, S),
    S > 1,
    addToEnd(Col, H, Col1),
    removeC(T, LO, Col1, R).
```

- The LO list has to be initialized with the original list and the collector variable has to be initialized with the empty list. So we need another function.
-

$$removeCMain(l_1 l_2 \dots l_n) = removeC(l_1 l_2 \dots l_n, l_1 l_2 \dots l_n, \emptyset)$$

```
% removeCMain(L:list, R:list)
% flow model: (i,o), (i,i)
% L - list from which we remove elements that occur only once
% R - resulting list

eliminaCMain(L, R):-removeC(L, L, [], R).
```

3. *Given a list of numbers, remove all the increasing sequences. Ex. remove([1,2,4,6,5,7,8,2,1]) => [2, 1]*

- For this problem, it is not enough to just check if the first 2 elements are in increasing order and if they are so, remove them. If we do like this, in case of sequences with an odd number of elements, we will always be left with one element that is not removed because we do not know which element to compare it to.

$$removeInc(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & , n = 0 \\ l_1 & , n = 1 \\ \emptyset & , n = 2 \text{ și } l_1 < l_2 \\ removeInc(l_2 \dots l_n) & , l_1 < l_2 < l_3 \\ removeInc(l_3 \dots l_n) & , l_1 < l_2 \geq l_3 \\ l_1 \cup removeInc\ (l_2 \dots l_n) & , otherwise \end{cases}$$

```prolog
% removeInc(L:list, R:list)
% flow model: (i,o) or (i,i)
% L - list from which we remove the increasing sequences
% R - resulting list

removeInc([], []).
removeInc([H], [H]).
removeInc([H1,H2], []):- H1 < H2.
removeInc([H1,H2,H3|T], R):-
    H1 < H2,
    H2 < H3,
    removeInc([H2,H3|T], R).
removeInc([H1,H2,H3|T], R):-
    H1 < H2,
    H2 >= H3,
    removeInc([H3|T], R).
removeInc([H1,H2|T], [H1|R]):-
    H1 >= H2,
    removeInc([H2|T], R).
```

4. *Given a linear list, compute the greatest common divisor of all numbers.*

We will use additional predicate gcd(a,b) for two numbers, and apply it for the list.

$$listGCD(l_1 l_2 \dots l_n) = \begin{cases} l1, & if\ n = 1 \\ gcd(l1, listGCD(l_2 \dots l_n), & otherwise \end{cases}$$

```
%gcd(M-first number,N-second number,R-result)
%gcd(i,i,o)
gcd(M, N, R) :-
    M =:= N,
    R is M.
gcd(M, N, R) :-
    N>M,
    Y is N-M,
    gcd(M, Y, R).
gcd(M, N, R) :-
    N<M,
    Y is M-N,
    gcd(Y, N, R).
```

```
%listGCD(L-list,R-res, integer)
%listGCD(i,o)-flow model
listGCD([E], E).
listGCD([H|T], R) :-
    listGCD(T, R1),
    gcd(H, R1, R).
```

Write the math model for *gcd*.

$$gcd(a,b,res) = \begin{cases} res = a & , a = b \\ \gcd(a, b-a, res) & , b > a \\ \gcd(a-b, b, res & , a > b \end{cases}$$

5. Compute the reverse of a number using a collector variable.

Math model:

$$inv\_no(n,nc,r) = \begin{cases} r = nc & , \quad n = 0 \\ inv\_no(n \ div \ 10, \quad nc * 10 + n \bmod 10, \quad r) & , n > 0 \end{cases}$$

$$inv\_noMain(n,r) = inv\_no(n, \quad 0, \quad r)$$

```
%inv_no(N - integer, NC - collector, R -result, integer)
%flow model (i,i,o)  (i,i,i)

inv_no(0, NC, NC).
inv_no(N, NC, R):-
    N>0,
    Digit is N mod 10,
    NewNC is NC*10+Digit,
    NewN is N div 10,
    inv_no(NewN, NewNC, R).

inv_noMain(N,R):-
    inv_no(N,0,R).
```

Tail recursion: optimization, reuses stack frame. Requirements:

- Deterministic predicate
- Recursive call is last

Compute the sum of elements in a list.

$$suma(l_1 l_2 \dots l_n) = \begin{cases} 0, & if\ n = 0 \\ l1 + suma(l_2 \dots l_n), & otherwise \end{cases}$$

$$suma(l_1 l_2 \dots l_n, c) = \begin{cases} c, & if\ n = 0 \\ suma(l_2 \dots l_n, l_1 + c), & otherwise \end{cases}$$

$$mainsuma(l_1 l_2 \dots l_n) = \begin{cases} suma(l_1 l_2 \dots l_n, & 0) \end{cases}$$

Implementation (Prolog):	
V1	V2

V1:

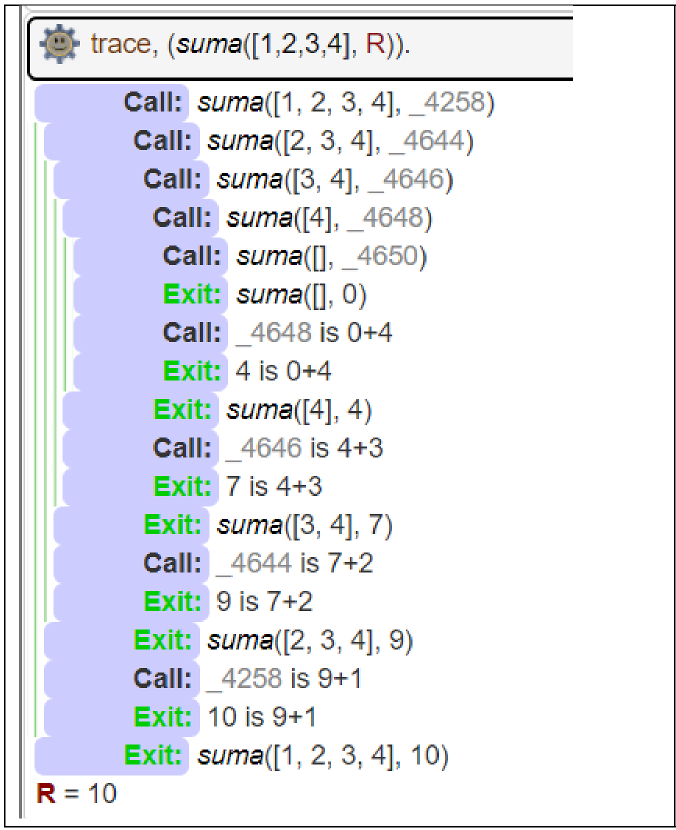
```
%For a list of elements, compute the sum.
%sum(L-list, S-result, integer)
%flow model: sum(i,o)

suma([],0).
suma([H|T],S):-
        suma(T,ST),
        S is H+ST.
```

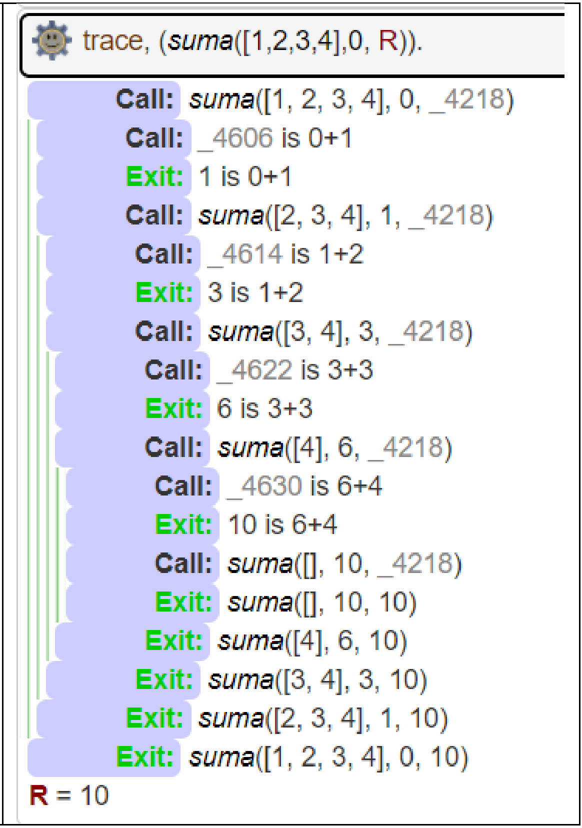
V2:

```
suma([], Col,Col).
suma([H|T], Col,R):-
    Col1 is Col+H,
    suma(T, Col1,R).

%suma([1,2,3,4],0, R).
```



```
trace, (suma([1,2,3,4], R)).
     Call: suma([1, 2, 3, 4], _4258)
      Call: suma([2, 3, 4], _4644)
       Call: suma([3, 4], _4646)
        Call: suma([4], _4648)
         Call: suma([], _4650)
         Exit: suma([], 0)
         Call: _4648 is 0+4
         Exit: 4 is 0+4
        Exit: suma([4], 4)
        Call: _4646 is 4+3
        Exit: 7 is 4+3
       Exit: suma([3, 4], 7)
       Call: _4644 is 7+2
       Exit: 9 is 7+2
      Exit: suma([2, 3, 4], 9)
      Call: _4258 is 9+1
      Exit: 10 is 9+1
     Exit: suma([1, 2, 3, 4], 10)
R = 10
```



```
trace, (suma([1,2,3,4],0, R)).
     Call: suma([1, 2, 3, 4], 0, _4218)
      Call: _4606 is 0+1
      Exit: 1 is 0+1
      Call: suma([2, 3, 4], 1, _4218)
       Call: _4614 is 1+2
       Exit: 3 is 1+2
       Call: suma([3, 4], 3, _4218)
        Call: _4622 is 3+3
        Exit: 6 is 3+3
        Call: suma([4], 6, _4218)
         Call: _4630 is 6+4
         Exit: 10 is 6+4
         Call: suma([], 10, _4218)
         Exit: suma([], 10, 10)
        Exit: suma([4], 6, 10)
       Exit: suma([3, 4], 3, 10)
      Exit: suma([2, 3, 4], 1, 10)
     Exit: suma([1, 2, 3, 4], 0, 10)
R = 10
```