

# Seminar 7 – Recapitulare

---

1. Se consideră problema ștergerii dintr-o listă a tuturor elementele care se repetă. De ex., pentru lista [1,2,3,2,1] rezultatul așteptat este lista [3].

Există mai multe variante de a rezolva problema, următoarea soluție conținând 3 predicate:

- Predicatul *exista*, care verifică dacă un element apare într-o listă sau nu.
- Predicatul *stergeelem*, care șterge toate aparițiile unui element dintr-o listă.
- Predicatul *rezolva*, care elimină elementele care se repetă.

Așadar, se dă codul următor:

```
% exista (L:lista, E: element)
% model de flux: (i,i), (i,o)
% L - lista in care verificam
% E - elementul cautat in lista L
exista([H|T],E) :- H=E. %a se observa lipsa tăieturii
exista([H|T],E) :- exista(T,E).

% stergeelem (L: lista, E: element, R: lista)
%model de flux: (i,i,o), (i,o,i)
% L - lista din care stergem
% E - elementul de sters
% R - lista rezultat
stergeelem([],X,[]).
stergeelem([H|T],X,R) :- H=X,stergeelem(T,X,R). %a se observa lipsa
tăieturii
stergeelem([H|T],X,[H|R]) :- stergeelem(T,X,R).

%rezolva (L: lista, R: lista)
%model de flux (i,o)
% L - lista de transformat
% R - lista rezultat
rezolva([],[]).
rezolva([H|T],S) :- exista(T,H),
                    stergeelem(T,H,R),
                    !, %a se observa amplasarea tăieturii
                    rezolva(R,S).
rezolva([H|T],[H|S]) :- not(exista(T,H)),
                        rezolva(T,S).
```

- Este corect codul de mai sus?
  - o Observăm predicatul *stergeelem*. Predicatul este nedeterminist (are mai multe soluții, dintre care doar una corectă), deși intenționăm un predicat determinist. De exemplu, pentru apelul `stergeelem([1,2,4,2,1,6], 2, X)`, rezultatele vor fi:
    - [1,4,1,6]

- [1,4,2,1,6]
    - [1,2,4,1,6]
    - [1,2,4,2,1,6]
  - Este, prin urmare, toată soluția greșită?
    - În mod interesant, nu. Predicatul principal furnizează soluția corectă și este determinist. De ce? Datorită tăieturii.
2. Se consideră problema generării tuturor combinațiilor luate a câte N, cu elementele unei liste date. Se dă următoarea implementare:

```
%comb(L: lista, N: integer, R:lista)
%model de flux (i,i,o)
%L - lista initiala
%N - numarul elementelor dorite in combinari
%R - un rezultat (o combinatie cu N elemente din lista L)
comb([E|_], 1, [E]).
comb([_|T], N, R):-
    comb(T, N, R).
comb([H|T],N, [H|R]):-
    N > 1,
    N1 is N - 1,
    comb(T, N1, R).
```

- Să presupunem că nu vrem toate combinațiile, ci doar cele în care elementele sunt în ordine crescătoare.
  - Evident, se poate face un alt predicat, care să verifice dacă o listă este crescătoare și un al 3-lea predicat care să genereze o combinație și să verifice dacă este crescătoare. Dar să presupunem că noi nu dorim un alt predicat, ci vrem să-l modificăm pe cel pe care îl avem.
    - Dacă dorim ca rezultatul să fie o listă crescătoare, modificarea trebuie efectuată acolo unde adăugăm un element în soluție, adică în a 3-a clauză. Înainte de a adăuga elementul H, trebuie să verificăm dacă respectă condiția, adică este mai mic decât primul element din soluție. Este în regulă să modificăm a 3-a clauză astfel?

```
comb([H|T], N, [H,H1|R]):-
    H < H1,
    N > 1,
    N1 is N - 1,
    comb(T, N1, [H1|R]).
```

- Dacă efectuăm această modificare vom avea eroare la  $H < H1$ , pentru că variabila H1 nu are valoare (este liberă). Parametrul 3 fiind de tip output, el nu are valoare în momentul apelului, ci trebuie să primească valoare în interiorul clauzei. Dacă analizăm varianta originală a predicatului, căutând unde primește parametrul R valoare, observăm că aceasta se întâmplă în ultima instrucțiune, în apelul recursiv. De aceea, dacă vrem să accesăm elemente din rezultat, trebuie să facem aceasta după apelul recursiv.

```

comb([H|T],N, [H,H1|R]):-
    N > 1,
    N1 is N - 1,
    comb(T, N1, [H1|R]),
    H < H1.

```

3. La teme de laborator apare problema următoare: să se șteargă 1-ul, al 3-lea, al 7-lea, etc. element dintr-o listă.

Să considerăm codul următor pentru a rezolva problema:

```

%elimina(list, pozC, pozR, listR)
%model de flux: (i, i, i, o)
elimina([], _, _, []).
elimina([_|T],PC, PR, R):-
    PC ::= PR, !,
    elimina(T, PC+1, PR*2+1, R).
elimina([H|T], PC, PR, [H|R]):-
    elimina(T, PC+1, PR, R).

```

E în regulă codul anterior, apelul inițial efectuându-se pentru  $\text{pozC} = \text{pozR} = 1$ ?

- Da, deoarece expresiile aritmetice, deși nu se calculează implicit, la transmiterea argumentelor pentru apelurile recursive, vor ajunge să fie comparate prin intermediul operatorului `::=`, care forțează evaluarea lor.