

Operații pe liste

1. Obiective

Lucrarea de față vă familiarizează cu operațiile ce se pot realiza pe liste: adăugarea, ștergerea, căutarea și înlocuirea de elemente. Aceste predicate sunt deja implementate în Prolog, din acest motiv implementarea noastră lor va conține și "1" în denumire.

2. Considerații teoretice

2.1. Reprezentarea unei liste

Lista vidă este reprezentată prin simbolul `[]`.

O listă care conține cel puțin un element poate fi reprezentată prin șablonul `[H|T]`, unde `H` este capul listei și poate fi de orice tip, iar `T` este coada listei și trebuie să fie la rândul ei, o listă.

Observație. Șablonul `[H|T]` nu permite unificarea cu o listă vidă. Încercați întrebarea următoare: `?- [H|T]=[]`. Veți vedea că rezultatul returnat este **false**.

Exemple:

?- `L=[1,2,3]`.

?- `[1,2,3]=[1|[2|[3]]]`.

?- `L=[a,b,c]`.

?- `L=[a, [b, [c]]]`.

?- `L=[a, [b], [[c]]]`.

Observație. Șablonul `[H|T]` poate face **segregare** sau **concatenare** în funcție de caz. Regula este următoarea. Când `H` și `T` sunt neinstantiate șablonul `[H|T]` face segregare, pe când dacă sunt instantiate face concatenare. Încercați întrebarea următoare:

?- `[H|T]=[1,2,3], X=3, L=[X|T]`.

2.2. Predicatul „member”

Predicatul *member(X,L)* verifică dacă un elementul *X* este inclus lista *L*. Recurența matematică poate fi formulată în felul următor:

$$x \in L \Leftrightarrow x = \text{primul}(L) \vee x \in \text{coada}(L)$$

În Prolog se va scrie:

```
member1(X, [H|T]) :- H=X.  
member1(X, [H|T]) :- member1(X, T).
```

Simplificarea predicatului:

- Prima clauză a predicatului *member1/2* poate fi simplificată prin înlocuirea lui *H* din capul clauzei cu *X*.
- În a doua clauză, variabila *H* nu este folosită și atunci o putem înlocui cu *_* (simbolul pentru o variabilă anonimă).
- Aceeași înlocuire o putem face și pentru *T* din prima clauză.

Astfel predicatul *member1/2* devine:

```
member1(X, [X|_]).  
member1(X, [_|T]) :- member1(X, T).
```

Exemplu de urmărire a execuției (cu *trace*):

[trace] 3 ?- member1(3,[1,2,3,4]).

Call: (7) member1(3, [1,2,3,4]) ?

% se unifică cu clauza 2

Call: (8) member1(3, [2,3,4]) ?

% apelul recursiv din clauza 2

Call: (9) member1(3, [3, 4]) ?

% se unifică cu clauza 1

Exit: (9) member1(3, [3, 4])

% succes și iese din apelul recursiv

Exit: (8) member1(3, [2, 3, 4]) ?

Exit: (7) member1(3, [1, 2, 3, 4]) ?

true ;

% repetăm întrebarea

Redo: (9) member1(3, [3, 4]) ?

% ultima unificare (cea cu clauza 1) se

% va relua și se va unifica cu clauza 2

Call: (10) member1(3, [4]) ?

% se unifică cu clauza 2

Call: (11) member1(3, []) ?

% apelul recursiv din clauza 2

Fail: (11) member1(3, []) ?

% eșuează (nu există clauză care să

% conțină în al doilea argument

% lista vidă)

Fail: (10) member1(3, [4]) ?

Fail: (9) member1(3, [3, 4]) ?

Fail: (8) member1(3, [2, 3, 4]) ?

Fail: (7) member1(3, [1, 2, 3, 4]) ?
false.

Urmăriți execuția la:

?- member1(a,[a, b, c, a]).

?- X=a, member1(X, [a, b, c, a]).

?- member1(a, [1,2,3]).

Exemplu de comportament nedeterminist la predicatul *member*:

?- member1(X, [1,2,3]).

X = 1 ; **% la repetarea întrebării se alege următoarea soluție posibilă**

X = 2 ;

X = 3 ;

false. **% nu mai exista alte soluții**

?- member1(1, L).

L = [1|_] ;

% prima soluție este o listă care începe cu 1, continuată de orice

L = [_, 1|_] ;

% a doua soluție este o listă care are 1 pe poziția 2, continuată de orice

L = [_, _, 1|_] ;

% etc.

2.3. Predicatul „append”

Predicatul *append(L1, L2, R)* realizează concatenarea listelor *L1* și *L2* și pune rezultatul în parametrul *R*. Recurența matematică poate fi formulată în felul următor:

$$L_1 \oplus L_2 = R \Leftrightarrow \text{primul}(R) = \text{primul}(L_1) \wedge \text{coada}(R) = \text{coada}(L_1) \oplus L_2$$

În cazul în care *L1* este o listă vidă atunci *R* va fi egal cu *L2*. În Prolog, se va scrie ca:

append1([], L2, R) :- R=L2.

append1([H|T], L2, R) :- append1(T, L2, CoadaR), R=[H|CoadaR].

Putem simplifica predicatul prin înlocuirea argumentului *R* din capul celor 2 clauze cu ultima unificare.

append1([], L2, L2).

append1([H|T], L2, [H|CoadaR]) :- append1(T, L2, CoadaR).

Observație. Ce fel de recursivitate avem aici? Înainte sau înapoi? Explicați.

Observație. Nu uitați că aceste două variante (explicită și simplificată) sunt *echivalente*. Chiar dacă nu pare, ambele folosesc același tip de recursivitate și funcționează în același fel. Schimbările aduse în cele două clauze ale predicatului sunt identice, în loc să specificăm explicit unificarea, se face implicit.

Exemplu de urmărire a execuției (cu trace):

[trace] 6 ?- append1([a,b],[c,d],R).

Call: (7) append1([a, b], [c, d], _G1680) ? **% se unifică cu clauza 2 -> apoi apel recursiv**

Call: (8) append1([b], [c, d], _G1762) ? **% se unifică cu clauza 2 -> apoi apel recursiv**

Call: (9) append1([], [c, d], _G1765) ? **% se unifică cu clauza 1**

Exit: (9) append1([], [c, d], [c, d]) ? **% succes și iese din apelul recursiv**

Exit: (8) append1([b], [c, d], [b, c, d]) ?

Exit: (7) append1([a, b], [c, d], [a, b, c, d]) ?

R = [a, b, c, d]. **% nu mai exista o alta soluție**

Observație. Urmăriți ce se întâmplă. Prima listă este traversată până când ajunge să fie vidă, moment în care (la condiția de oprire) rezultatul este instanțiat cu a doua listă (R=[c, d]). Prin revenirea din apelurile recursive succesive, prima listă este concatenată la începutul rezultatului, întâi devine R==[b,c,d] și apoi R=[a,b,c,d]. Prin urmare, este recursivitate **înapoi**. Ce se întâmplă dacă schimbăm recursivitatea înapoi cu una înainte?

Exemplu de comportament nedeterminist la predicatul *append*:

?- append1(L1, L2, [1,2,3]).

L1 = [],

L2 = [1, 2, 3] ; **% prima soluție**

L1 = [1],

L2 = [2, 3] ; **% a doua soluție**

L1 = [1, 2],

L2 = [3] ; **% a treia soluție**

L1 = [1, 2, 3],

L2 = [] ; **% nu mai există alte soluții**

false.

În termeni simpliști, punem Prolog-ului următoarea întrebare, “Care două liste concatenate rezultă în [1,2,3]?” și ne returnează toate soluțiile posibile. Astfel, ideea clasică de ce reprezintă un input și un output nu este aplicabilă. Prolog consideră ca fiind output, *variabilele*, oricare argumente ar fi acestea.

Urmăriți execuția la:

?- append1([1, [2]], [3|[4, 5]], R).

?- append1(T, L, [1, 2, 3, 4, 5]).

?- append1(_, [X|_], [1, 2, 3, 4, 5]).

Vă amintiți observația despre șablonul [H|T] și inabilitatea de a se unifica cu lista vidă []? Inversați ordinea clauzelor predicatului *append* și urmăriți execuția întrebărilor de mai sus. Ce diferențe apar în comportamentul predicatului?

2.4. Predicatul „delete”

Predicatul *delete(X, L, R)* șterge prima apariție a elementul X din lista L și pune rezultatul în R. Dacă X nu există în L atunci R va fi egal cu L. Recurența matematică poate fi formulată în felul următor:

$$R = L - \{x\} = \begin{cases} \{\}, & L = \{\} \\ coada(L), & x = primul(L) \\ \{primul(L)\} \oplus (coada(L) - \{x\}), & \text{altfel} \end{cases}$$

În Prolog se va scrie:

```
delete1(X, [X|T], T).  
% șterge prima apariție și se oprește  
delete1(X, [H|T], [H|R]) :- delete1(X, T, R).  
% altfel iterează peste elementele listei  
delete1(_, [], []).  
% dacă a ajuns la lista vidă înseamnă că elementul nu a fost găsit  
% și putem returna lista vidă
```

Observație1. Observați că predicatul *delete/3* are de fapt 2 condiții de oprire (prima și a treia clauză). Prima clauză se unifică când elementul a fost găsit și șterge elementul, pe când a treia clauză se unifică doar dacă elementul nu apare în listă sau se dă ca listă de intrare lista vidă.

Observație2. Aici avem recursivitate înapoi. **Al doilea argument** (lista de intrare), în a treia clauză, se unifică cu lista vidă și astfel se **oprește** parcurgerea listei, pe când **al treilea argument** - **instanțiază** rezultatul.

Urmărește execuția la:

?- delete1(3, [1, 2, 3, 4], R).

?- X=3, delete1(X, [3, 4, 3, 2, 1, 3], R).

?- delete1(3, [1, 2, 4], R).

?- delete1(X, [1, 2, 4], R).

Observație. Ce se întâmplă cu aceste întrebări dacă scoatem a treia clauză a predicatului?

2.5. Predicatul „delete_all”

Predicatul *delete_all*(X, L, R) va șterge toate aparițiile lui X din lista L și va pune rezultatul în R. Recurența matematică poate fi formulată în felul următor:

$$R = L - \{x\} = \begin{cases} \{\}, & L = \{\} \\ coada(L) - \{x\}, & x = primul(L) \\ \{primul(L)\} \oplus (coada(L) - \{x\}), & altfel \end{cases}$$

Predicatul *delete_all* diferă față de predicatul *delete* doar la prima clauză.

```
delete_all(X, [X|T], R) :- delete_all(X, T, R).
```

% dacă s-a șters prima apariție se va continua și pe restul elementelor

```
delete_all(X, [H|T], [H|R]) :- delete_all(X, T, R).
```

```
delete_all(_, [], []).
```

Observație. Observăm că diferența dintre *delete_all/3* și *delete/3* apare când ajungem la elementul pe care vrem să îl ștergem (prima clauză). Restul predicatului rămâne la fel. În predicatul *delete/3*, prima clauză este o condiție de oprire și astfel șterge doar prima apariție a elementului respectiv. Prin modificarea într-o clauza recursivă, va șterge toate aparițiile acestui element.

Clarificare Suplimentară. Putem extinde predicatul *delete_all/3* pentru a arată de ce este în continuare recursie înapoi și faptul că unificarea se face implicit.

```
delete_all(X, [X|T], R) :- delete_all(X, T, R1), R=R1.
```

```
delete_all(X, [H|T], R):- delete_all(X, T, R1), R=[H|R1].
```

```
delete_all(_, [], []).
```

Adițional, prima clauză a predicatului *delete_all/3* conține încă o simplificare (egalitate implicită):

```
delete_all(X, [H|T], R) :- X=H, delete_all(X, T, R1), R=R1.
```

Veți observa că acest predicat permite backtracing, o opțiune de reducere a ramurilor de backtracing este de a adăuga condiția opusă în a doua clauză:

```
delete_all(X, [H|T], R):- not(X=H), delete_all(X, T, R1), R=[H|R1].
```

Urmăriți execuția la:

```
?- delete_all(3, [1, 2, 3, 4], R).
```

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

```
?- delete_all(3, [1, 2, 4], R).
```

```
?- delete_all(X, [1, 2, 4], R).
```

3. Exerciții

1. Scrieți predicatul *add_first(X,L,R)* care adaugă X la începutul listei L și pune rezultatul în R.

Sugestie: simplificați pe cât de mult posibil acest predicat.

```
% add_first(X,L,R). - adaugă X la începutul listei L și pune rezultatul în R
?- add_first(1,[2,3,4],R).
R=[1,2,3,4].
```

2. Scrieți predicatul *append3/4* care să realizeze concatenarea a 3 liste.

Sugestie: Nu folosiți *append*-ul a două liste.

```
% append3(L1,L2,L3,R). - va realiza concatenarea listelor L1,L2,L3 în R
?- append3([1,2],[3,4,5],[6,7],R).
R=[1,2,3,4,5,6,7] ;
false.
```

3. Scrieți un predicat care realizează suma elementelor dintr-o lista dată.

```
% sum(L, S). - calculează suma elementelor din L și returnează suma în S
?- sum_bwd([1,2,3,4], S).
R=10.

?- sum_fwd([1,2,3,4], S).
R=10.
```

4. Scrieți un predicat care separă numerele pare de cele impare. (*Întrebare:* de ce avem nevoie pentru recursivitate înainte?)

```
?- separate_parity([1, 2, 3, 4, 5, 6], E, O).
E = [2, 4, 6]
O = [1, 3, 5] ;
false
```


5. Scrieți un predicat care să șteargă toate elementele duplicate dintr-o listă.

```
?- remove_duplicates([3, 4, 5, 3, 2, 4], R).  
R = [3, 4, 5, 2] ; % păstrează prima apariție  
false
```

SAU

```
R = [5, 3, 2, 4] ; % păstrează ultima apariție  
false
```

6. Scrieți un predicat care să înlocuiască toate aparițiile lui X în lista L cu Y și să pună rezultatul în R.

```
?- replace_all(1, a, [1, 2, 3, 1, 2], R).  
R = [a, 2, 3, a, 2] ;  
false
```

7. Scrieți un predicat care șterge tot al k-lea element din lista de intrare.

```
?- drop_k([1, 2, 3, 4, 5, 6, 7, 8], 3, R).  
R = [1, 2, 4, 5, 7, 8] ;  
false
```

8. Scrieți un predicat care șterge duplicatele consecutive fără a modifica ordinea elementelor din listă.

Sugestie: căutați observația despre șablonul extins în Laboratorul 1.

```
?- remove_consecutive_duplicates([1,1,1,1, 2,2,2, 3,3, 1,1, 4, 2], R).  
R = [1,2,3,1,4,2] ;  
false
```

9. Scrieți un predicat care adăugă duplicatele consecutive într-o sub-listă fără a modifica ordinea elementelor din listă.

```
?- pack_consecutive_duplicates([1,1,1,1, 2,2,2, 3,3, 1,1, 4, 2], R).  
R = [[1,1,1,1], [2,2,2], [3,3], [1,1], [4], [2]] ;  
false
```