Operații pe liste

1 Obiective

Lucrarea de față îl familiarizează pe student cu operațiile ce se pot realiza pe liste: adăugarea, ștergerea, căutarea și înlocuirea de elemente.

2 Considerații teoretice

2.1 Reprezentarea unei liste

Lista vida este reprezentata prin simbolul **[ ]**

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

Exemple:

```
?- L=[1,2,3].
?- [1,2,3]=[1|[2|[3]]].
?- L=[a,b,c].
?- L=[a, [b, [c]]].
?- L=[a, [b], [[c]]].
```

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:

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

În Prolog se va scrie:

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

Prima clauza din predicatul *member* 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 *member* devine:

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

Pentru a urmări execuția predicatului *member* va trebui să-l redenumiți (ex: *member1*) deoarece acest predicat este predefinit in interpretorul Prolog.

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 face unificarea 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 aibă în al doilea parametru 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.
```

Urmărește 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
posibila
X = 2 ;
X = 3 ;
false. % nu mai exista alte soluții

?- member1(1, L).
L = [1|_] ;     % prima soluție este o listă care începe cu 1
L = [_, 1|_] ;  % soluția doi este o listă care are 1 pe poziția 2
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 formulata în felul următor:

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

În cazul în care L1 este lista vida atunci R va fi egal cu L2. În Prolog se va scrie:

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

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

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

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
```

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.
```

Urmărește execuția la:

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

Inversați ordinea clauzelor din predicatul *append* și reluați trasarea 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:

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

În Prolog se va scrie:

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

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).
```

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

Urmărește 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 *append3(L1,L2,L3,R)* care să realizeze concatenarea a 3 liste.
2. Scrieți predicatul *add_first(X,L,R)* care adaugă X la începutul listei L și pune rezultatul în R.
3. Scrieți un predicat care realizează suma elementelor dintr-o lista dată.
4. Scrieți un predicat care separă numerele pare de cele impare.

```
?- 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

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
```