Liste adânci

1 Obiective

În lucrarea de față vom prezenta o generalizare a tipului listă. Elementele listei pot să fie la rândul lor tot liste. Astfel o listă adâncă poate să aibă elemente la diverse nivele de imbricare.

Exemple:

```
L1 = [1,2,3,[4]]
L2 = [[1],[2],[3],[4,5]]
L3 = [[],2,3,4,[5,[6]],[7]]
L4 = [[[[1]]],1,[1]]
L5 = [1,[2],[[3]],[[[4]]],[5,[6,[7,[8,[9],10],11],12],13]]
L6 = [alpha,2,[beta],[gamma,[8]]]
```

O listă adâncă (DL) în Prolog este reprezentată de o structură recursivă, unde multiple liste de adâncimi diferite sunt imbricate una în cealaltă. În mod formal, o listă adâncă poate fi definită ca:

$$DL = [H|T] \; unde \; H \in \{atom, listă, listă \; adâncă\} \; și \; T = listă \; adâncă$$

Rețineți. O listă simplă este un caz trivial de listă adâncă.

Toate operațiile definite pe liste simple pot fi folosite și pe liste adânci, inclusiv (dar nu numai) predicatele *member/2*, *append/3* și *delete/3*. Pentru a înțelege modul de funcționare a acestor operații pe liste adânci, trebuie să considerăm o listă adâncă echivalentă cu o listă simplă cu elemente de un tip diferit, dar doar cele de pe primul nivel.

Testați următoarele întrebări:

```
?- member(2,L5).
?- member([2], L5).
?- member(X, L5).
?- append(L1,R,L2).
?- append(L4,L5,R).
?- delete(1, L4,R).
?- delete(13,L5,R).
```

2 Considerații teoretice

2.1 Predicatul „atomic”

Pentru a executa diferite operații pe toate elementele listei adânci, trebuie să știm cum să lucrăm cu aceste elemente în funcție de tipul lor (dacă sunt atomi, le procesăm, dacă sunt liste, trebuie să continuăm cu decompoziția pentru a procesa).

Vom folosi predicatul predefinit *atomic(X)* pentru a verifica dacă X este un element simplu (număr, simbol) sau este o structură complexă (ex: o altă listă).

Testați următoarele întrebări:

```
? – atomic(apple).
? – atomic(4).
? – atomic(X).
? – atomic(apple(2)).
? – atomic([1,2,3]).
? – atomic([]).
```

2.2 Predicatul „depth”

Acest predicat numără nivelul maxim de imbricare a unei liste adânci. Adâncimea unui atom este definită ca fiind 0, iar adâncimea unei liste simple (incluzând lista vidă) este 1.

Acest predicat calculează nivelul maxim de imbricare al unei liste adânci, iterează peste elementele listei și verifică natura lor (folosind predicatul predefinit *atomic/1*). Dacă elementul este atomic, apelează predicatul pe coada listei (clauza 2), pe când dacă elementul nu este atomic atunci se apelează recursiv pe acel element (clauza 2 returnează fail și face backtracking pe clauza 3). Adâncimea listei este egală cu adâncimea maximă a unui element plus unu.

```
max(A, B, A):- A>B, !.
max(_, B, B).

depth([],1).
depth([H|T],R):- atomic(H), !, depth(T,R).
depth([H|T],R):- depth(H,R1), depth(T,R2), R3 is R1+1, max(R3,R2,R).
```

Când calculăm adâncimea maximă vom avea trei ramuri:

1. Ajungem la lista vidă. Adâncimea este 1. **(clauza 1)**
2. Avem un prim element atomic, îl ignorăm deoarece nu influențează adâncimea. Adâncimea unei listei este egală cu adâncimea cozii. Prin urmare, trebuie să apelăm recursiv pe coadă pentru a continua parcurgerea listei. **(clauza 2)**
3. Avem o listă ca primul element al listei adânci. În acest caz, adâncimea listei va fi fie adâncimea cozii, fie adâncimea primei liste incrementată cu unu (*De ce?).* **(clauza 3)**

Testați predicatul pentru listele L1-L6 de mai sus, de exemplu:

```
?- depth(L1, R).
```

2.3 Predicatul „flatten”

Acest predicat aplatizează o listă adâncă. Lista rezultat va conține numai elemente atomice (lista simplă echivalentă listei adânci conținând aceleași elemente la nivelul 1 de imbricare). Și în acest caz trebuie să se verifice natura fiecărui elementul. Dacă elementul este la rândul lui o listă atunci va trebui aplatizată.

Această operație reprezintă obținerea unei liste simple. Pentru a face acest lucru, vom lua doar elementele atomice din lista de intrare și le vom adăuga la rezultat.

```
flatten([],[]).
flatten([H|T], [H|R]):- atomic(H), !, flatten(T,R).
flatten([H|T], R):- flatten(H,R1), flatten(T,R2), append(R1,R2,R).
```

Vom avea 3 cazuri principale:

- Trebuie să ne ocupăm de lista vidă. Aplatizarea unei liste vide rezultă într-o listă vidă. **(clauza 1)**
- Dacă primul element al listei este atomic, atunci îl adăugăm la rezultat și procesăm restul listei. **(clauza 2)**
- Dacă primul element nu este atomic, rezultat trebuie compus ca toate elementele atomice ale primului element și toate elementele atomice ale cozii. (*Cum colectăm cele două rezultatele într-o singura listă?)* **(clauza 3)**

Observație. Este greu de observat unde în acest predicat se întâmplă aplatizarea. Este făcută de predicatul *append* și funcționalitatea acestuia. De exemplu:
L = [[1],2,3,4] unde H=[1] și T = [2,3,4]

Când facem append între primul element și coadă, obținem:
?- append(H,T,R).
R = [1,2,3,4]

Încercați să modificați append-ul din flatten în:
append([R1],R2,R).

Testați predicatul pentru listele L1-L6, de exemplu:

?- flatten(L1, R).

2.4 Predicatul „heads"

Acest predicat extrage toate elementele atomice de la capul fiecărei liste.

% Varianta 1

Ne vom folosi de un predicat auxiliar, care trece peste elementele atomice ale unei liste. Vom lua doar primul element din fiecare listă, apoi vom folosi predicatul auxiliar pentru a trece peste restul elementelor atomice.

```
skip([],[]).
skip([H|T],R):- atomic(H),!,skip(T,R).
skip([H|T],[H|R]):- skip(T,R).

% luăm primul element din fiecare listă,
% după trecem peste restul elementelor
% prin apelul predicatului skip, și apelăm recursiv pentru liste
heads1([],[]).
heads1([H|T],[H|R]):- atomic(H),!,skip(T,T1), heads1(T1,R).
heads1([H|T],R):- heads1(H,R1), heads1(T,R2),append(R1,R2,R).
```

% Varianta 2

Ne vom folosi de un al treilea parametru (*flag*) pentru a ști dacă am extras capul listei curente sau nu.

```
heads2([],[],_).

% dacă flag=1 atunci suntem la început de lista și putem extrage capul listei; în apelul recursiv setam flag=0
heads2([H|T],[H|R],1):- atomic(H), !, heads2(T,R,0).

% dacă flag=0 atunci nu suntem la primul element atomic și
% atunci continuam cu restul elementelor
heads2([H|T],R,0):- atomic(H), !, heads2(T,R,0).

% dacă am ajuns la aceasta clauza înseamnă că primul element nu este atomic și atunci trebuie să apelam recursiv și pe acest element
heads2([H|T],R,_):- heads2(H,R1,1), heads2(T,R2,0), append(R1,R2,R).

% un wrapper pentru predicatul heads
heads2(L,R):- heads2(L, R, 1).
```

Testați predicatul pentru listele L1-L6, de exemplu:

?- heads1(L1, R).

?- heads2(L1, R).

2.5 Predicatul „member”

Putem folosi predicate pe care le-am învățat deja pentru o implementare simplă a predicatului member/2 pentru liste adânci în felul următor:

```
% Varianta 1
member1(X, L):- flatten(L,L1), member(X,L1).
```

Observație: Această versiune poate găsi doar elemente atomice. Verificați a doua variantă a predicatului membru pentru o soluție mai generală.

Cu toate acestea, dacă dorim să extindem predicatul member/2 implementat în sesiuni anterioare de laborator pentru a funcționa și pe liste adânci, trebuie să modificăm predicatul member/2 astfel:

```
% Varianta 2
member2(H, [H|_]).
member2(X, [H|_]):- member1(X,H).
member2(X, [_|T]):- member1(X,T).
```

Funcționează similar cu predicatul member pentru liste simple, considerăm membru al listei toate elementele care apar în listă, atomice sau nu, la orice nivel.

Testați următoarele întrebări:

```
?- member2(1,L1).
?- member2(4,L2).
?- member2([5,[6]], L3).
?- member2(X,L4).
?- member2(X,L6).
?- member2(14,L5).
```

3 Exerciții

1. Scrieți predicatul *count_atomic(L,R)* care calculează numărul de elemente atomice din lista *L* (*toate elementele atomice de la toate adâncimile*).

```
?- count_atomic([1,[2],[[3]],[[[4]]],[5,[6,[7,[8,[9],10],11],12],13]], R).
R = 13;
false.
```

2. Scrieți predicatul *sum_atomic(L,R)* care calculează suma elementelor atomice din lista *L* (*toate elementele atomice de la toate adâncimile*).

```
?- sum_atomic([1,[2],[[3]],[[[4]]]], R).
R = 10;
false.
```

3. Scrieți predicatul *replace(X,Y,L,R)* care înlocuiește pe *X* cu *Y* în lista adâncă *L* (la orice nivel de imbricare) și pune rezultatul în *R.*

```
?- replace(1, a, [[[[1,2], 3, 1], 4],1,2,[1,7,[[1]]]], R).
R = [[[[a, 2], 3, a], 4], a, 2, [a, 7, [[a]]]];
false.
```

4. Scrieți predicatul *lasts(L,R)* care extrage elementele atomice de pe ultima poziție (imediat anterior *']'*) din fiecare sublistă din *L*.

```
?- lasts([1,2,[3],[4,5],[6,[7,[9,10],8]]], R).
R = [3,5,10,8] ;
false.
```

5. Înlocuiți fiecare secvență constantă de o anumită adâncime cu lungimea ei (**fără** să utilizați predicatul *length/2*).

```
? -  len_con_depth([[1,2,3],[2],[2,[2,3,1],5],3,1],R).
R = [[3],[1],[1,[3],1],2].
```

6. Înlocuiți fiecare secvență constantă de o anumită adâncime cu adâncimea ei (**fără** să utilizați predicatul *depth/2*).

```
? -  depth_con_depth([[1,2,3],[2],[2,[2,3,1],5],3,1],R).
R = [[1], [1], [1, [2], 1], 0].
```

7. Modificați predicatul *member2/2* pentru liste adânci astfel încât să fie determinist (în acest caz, deterministic se referă la faptul că predicatul va returna un singur răspuns – nu va exista opțiunea de *next* – este nevoie de multiple teste ale predicatului *member_deterministic/2* cu input-uri variate pentru a verifica).

```
?- member_deterministic(1, [1,2,1]).
true.

?- member_deterministic(1, [[1],2,1]).
true.
```

8. Scrieți un predicat care să sorteze o listă adâncă în funcție de adâncimea fiecărui element. Dacă două elemente au aceeași adâncime atunci se vor compara în funcție de elementele atomice pe care le conțin.

   Sugestie:

   - L1 < L2, dacă L1 și L2 sunt liste, sau liste adânci, și dacă adâncimea listei L1 este mai mică decât adâncimea listei L2.
   - L1 < L2, dacă L1 și L2 sunt liste, sau liste adânci cu adâncime egală, toate elementele până la poziția *k* sunt egale, iar al *k+1-lea* element din lista L1 este mai mic decât al *k+1-lea* element din lista L (la adâncimi egale, lista cu indexul mai mic sublistei care va da ultima adâncime este considerat mai mare – ca în exemplu la comparația dintre 5 și [5])

```
?- sort_depth([[[[1]]], 2, [5,[4],7], [[5],4], [5,[0,9]]], R).
R = [2, [5,[0,9]], [[5],4], [5,[4],7], [[[1]]]] ;
false
```