Liste diferență și Efecte laterale

1 Obiective

În lucrarea de față vom prezenta un nou tip de reprezentare pentru liste. Listele incomplete au fost un pas intermediar spre liste diferență. Dacă în listele incomplete, variabila din coadă era anonimă, în cazul listelor diferență variabila este dată într-un nou parametru.

Un exemplu care ilustrează cum listele diferență ne pot face viața mai ușoară: dacă în listele complete/incomplete pentru a adăuga un element la final trebuie să parcurgem toată lista, în cazul listelor diferență se poate adăuga direct în variabila din coada listei (prin utilizarea acestui nou argument).

2 Considerații teoretice

2.1 Reprezentare

Listele diferență se reprezintă prin două părți: **începutul** listei și **sfârșitul** listei. De exemplu:

Lista L=[1,2,3] poate fi reprezentată în forma de listă diferență de variabilele:

S=[1,2,3|X] și E=X, *unde S reprezintă începutul (start), iar E sfârșitul (end)*

Denumirea de „diferență” vine de la faptul că lista L poate fi calculată prin diferența dintre S și E.

Lista vidă este reprezentată prin 2 variabile egale (S=E). Astfel condiția de oprire se schimbă:

- Pentru liste complete: pred(L,…):- L=[].
- Pentru liste incomplete: pred(L,…):- var(L), !, … .
- Pentru liste diferență: pred (S, E, …):- S=E, var(E), !, … .

Exemplu:

S: [1,2,3,4]
E: [3,4]
S-E: [1,2]
S-E (lista diferență) reprezintă lista obținută prin scoaterea părții lui E din S

Nu există avatanje când folosim liste diferență ca în exemplul anterior, dar când combinăm conceptele de variabilă liberă și unificare, listele diferență devin unelte utile. De exemplu, lista [1,2] poate fi reprezentată ca lista diferență [1,2|X]-X, unde X este o variabilă liberă.

2.2 Predicatul „add”

O listă Prolog este accesată prin primul element și prin coadă. Piedica acestui mod de a vedea lista este faptul că pentru a accesa al n-lea element, trebuie să accesăm toate elementele de dinaintea lui. Dacă, de exemplu, avem nevoie să adăugăm un element la finalul listei, atunci trebuie să trecem prin toate elementele din listă pentru a ajunge la final.

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

Alternativa este să folosim Liste Diferență (reprezentate prin două părți, începutul liste S și finalul listei E):

```
add_dl(X, LS, LE, RS, RE):- RS = LS, LE = [X|RE].
% variabila LE va conține pe prima poziție elementul adăugat
```

Dacă îl testăm în interpretorul de Prolog ne va da următorul răspuns:

```
?- LS=[1,2,3,4|LE], add_dl(5,LS,LE,RS,RE).
LE = [5|RE],
LS = [1,2,3,4,5|RE],
RS = [1,2,3,4,5|RE]
```

Putem viziona acest proces pas cu pas:

- Inițial avem → LS = [1,2,3,4|LE]
- Prima operație RS=LS → RS = [1,2,3,4|LE]
- A doua operație LE=[X|RE] → RS = [1,2,3,4|[X|RE]]
- Înlocuim X cu valoarea sa numerică →RS = [1,2,3,4|[5|RE]]
- Simplificăm lista RS → RS = [1,2,3,4,5|RE]

Pentru a înțelege mai bine modul de funcționare a predicatului, ne putem imagina lista ca fiind reprezentată prin doi "pointeri", unul care arată începutul listei (LS) și al doilea finalul listei (LE), o variabilă fără o valoare atribuită.

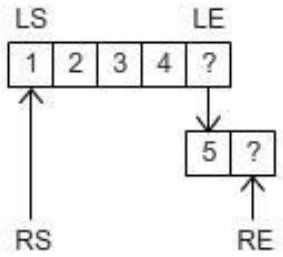

Figura 1. Adăugarea unui element la sfârșitul unei liste diferență

Rezultatul este, de asemenea, reprezentat prin doi "pointeri". Lista rezultat va fi lista de intrare cu un element inserat la final. Începutul listei de intrare și lista rezultată sunt aceeași, deci putem unifica varibila de început a listei de intrare cu variabila de început a listei rezultat (**RS=LS**).
Lista rezultat trebuie să aibă un final, ca si lista de intrare, aceasta va fi într-o variabilă (*RE*), dar trebuie cumva, să modificăm lista de intrare pentru a adăuga un nou element la final. Deoarece finalul listei de intrare este o varibilă liberă, putem să o unificăm cu lista care începe cu acest nou element și o variabilă nouă, finalul listei rezultat (**LE=[X|RE]**). După finalizarea execuției predicatului, putem observa ca lista de intrare *LS* și lista rezultat *RS* au aceeași valoare, iar finalul listei de intrare nu mai este o variabilă liberă (LE=[5|RE]).

2.3 Predicatul „append"

În cazul listelor diferență, predicatul *append* se va scrie pe o singură linie:

```
append_dl(LS1,LE1, LS2,LE2, RS,RE):- RS=LS1, LE1=LS2, RE=LE2.
```

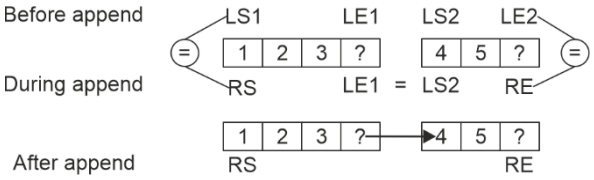

Figura 2. Concatenarea a două liste

Dacă îl testăm în interpretorul de Prolog ne va da următorul răspuns:

?- LS1=[1,2,3|LE1], LS2=[4,5|LE2], append_dl(LS1, LE1, LS2, LE2, RS, RE).
LE1 = LS2, LS2 = [4, 5|RE],
LE2 = RE,
LS1 = RS, RS = [1, 2, 3, 4, 5|RE].

Putem viziona acest proces pas cu pas:

- Inițial avem → LS1=[1,2,3|LE1], LS2=[4,5|LE2]
- Prima operație RS=LS1 → RS=[1,2,3|LE1]
- A doua operație LE1=LS2 → LE1 = [4,5|LE2]
    - Putem substitui LE1 în RS → RS=[1,2,3|[4,5|LE2]]
- Third operation RE=LE2
    - Simplificăm lista RS → RS=[1,2,3|[4,5|RE]]
- Simplify the RS list using the template → RS = [1,2,3,4,5|RE]

2.3.1 Sortarea rapidă

Rețineți algoritmul de sortare rapidă (și predicatul): secvența de intrare este împărțită în două – secvența de elemente mai mici sau egale cu pivotul și secvența de elemente mai mari decât pivotul; această procedură este apelată recursiv pe fiecare partiție și secvențele sortate rezultate sunt concatenate cu pivotul pentru a genera secvența sortată:

```
quicksort([H|T], R):-
   partition(H, T, Sm, Lg),
   quicksort(Sm, SmS),
   quicksort(Lg, LgS),
   append(SmS, [H|LgS], R).
quicksort([], []).
```

Ca în cazul predicatului **inorder**, predicatul `quicksort/2` va pierde timp de execuție când face `append/3` între rezultatele apelurilor recursive. Pentru a evita acest lucru putem folosi liste diferențe:

```
quicksort_dl([H|T], S, E):- % s-a adăugat un parametru nou
   partition(H, T, Sm, Lg), % predicatul partition a rămas la fel
   quicksort_dl(Sm, S, [H|L]), %concatenare implicită
   quicksort_dl(Lg, L, E).
quicksort_dl([], L, L). % condiția de oprire s-a modificat

partition(P, [X|T], [X|Sm], Lg):- X<P, !, partition(P, T, Sm, Lg).
partition(P, [X|T], Sm, [X|Lg]):- partition(P, T, Sm, Lg).
partition(_, [], [], []).
```

Predicatul **partition** rămâne neschimbat, scopul acestuia este de a împărți lista în două prin compararea elementelor cu pivotul. Toate elementele din listă trebuie accesate pentru această operație, prin urmare nu putem îmbunătăți performanța acestui predicat.

Predicatul **quicksort** funcționează în aceeași manieră ca înainte: împarte lista în elemente mai mari și mai mici decât pivotul și aplică quicksort recursiv pe fiecare partiție. Diferența între versiunea originală și aceasta este la lista rezultat, reprezentată prin două elemente, începutul și finalul listei, și în consecință modul prin care cele două rezultate ale apelurilor recursive sunt legate cu pivotul (figura de mai jos).

Urmărește execuția la:
?- quicksort_dl([4,2,5,1,3], L, []).
?- quicksort_dl([4,2,5,1,3], L, _).

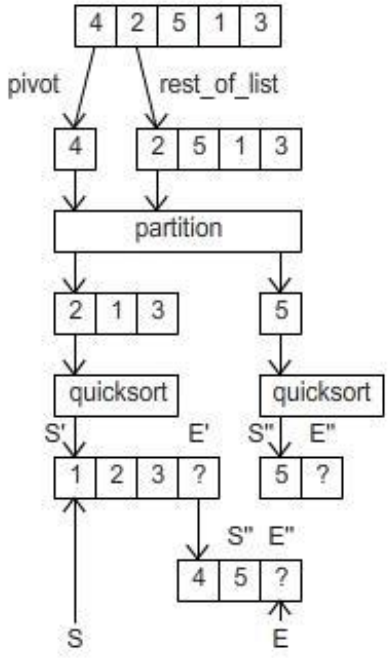

Figura 4. Sortare rapidă folosind liste diferență

2.3.2 Traversare în ordine

Predicatele pentru parcurgerea unui arbore sunt folosite pentru a extrage elementele dintr-un arbore într-o listă într-o ordine specifică. Partea intensiv-computațională a acestor predicate nu este parcurgerea ci combinarea listei rezultat pentru a obține rezulatul final. Cu toate că este ascuns, Prolog va trece prin aceleași elementele ale listei de multiple ori pentru a forma lista rezultat.

Putem reduce munca computațională prin înlocuirea listelor simple cu cele diferență.

Predicatul de parcurgere în inordine folosind liste simple pentru a stoca rezultatul:

```
inorder(t(K,L,R),List):-
   inorder(L,ListL),
   inorder(R,ListR),
   append1(ListL,[K|ListR],List).
inorder (nil,[]).
```

Prin urmărirea execuției predicatului **inorder/2**, putem observa ușor munca făcută de predicatul append. Este, de asemenea, vizibil că predicatul append accesează aceleași elemente din lista rezultat de mai multe ori în timp ce rezultatele intermediare sunt concatenate la cel final:

```
[. . .]
 8     3 Exit: inorder(t(5,nil,nil),[5]) ?
12     3 Call: append1([2],[4,5],_1594) ?
13     4 Call: append1([],[4,5],_10465) ?
13     4 Exit: append1([],[4,5],[4,5]) ?
12     3 Exit: append1([2],[4,5],[2,4,5]) ?
[. . .]
22     2 Call: append1([2,4,5],[6,7,9],_440) ?
23     3 Call: append1([4,5],[6,7,9],_20633) ?
24     4 Call: append1([5],[6,7,9],_21109) ?
25     5 Call: append1([],[6,7,9],_21585) ?
25     5 Exit: append1([],[6,7,9],[6,7,9]) ?
24     4 Exit: append1([5],[6,7,9],[5,6,7,9]) ?
23     3 Exit: append1([4,5],[6,7,9],[4,5,6,7,9]) ?
22     2 Exit: append1([2,4,5],[6,7,9],[2,4,5,6,7,9]) ?
[. . .]
```

Putem îmbunătății eficiența predicatului **inorder/2** prin înlocuirea vechiului *append* cu operații pe liste diferență. Predicatul **inorder_dl/3** va conține 3 argumente: nodul curent în procesare, începutul listei rezultate și finalul listei rezultate:

```
% când ajungem la finalul arborelui, unificăm începutul și finalul listei
% parțiale de rezultat - lista vidă este reprezentată de 2 variabile egale
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
   % obtinem începutul și finalul listelor pentru subarborele stâng și drept
   inorder_dl(L,LSL,LEL),
   inorder_dl(R,LSR,LER),
   % începutul listei rezultat este începutul listei subarborelui stâng
   LS=LSL,
   % cheia K este adăugat între finalul din stânga și începutul din dreapta
   LEL=[K|LSR],
   % finalul listei rezultat este finalul listei subarborelui drept
   LE=LER.
```

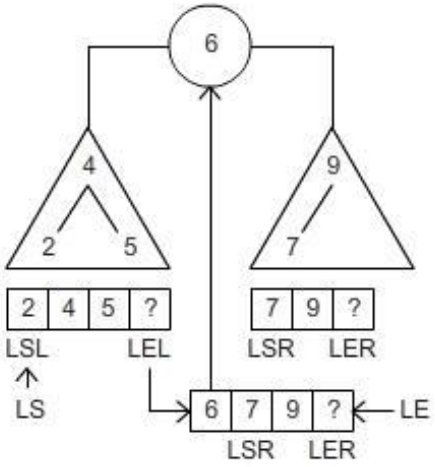

Figura 3. Concatenarea a două liste în traversarea în inordine a unui arbore

Urmărește execuția la:

```
?- tree1(T), inorder_dl(T,L,[]).
?- tree1(T), inorder_dl(T,L,_).
```

Predicatul poate fi simplificat prin înlocuirea unificărilor explicite cu unificări implicite:

```
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
   inorder_dl(L, LS, [K|LT]),
   inorder_dl(R, LT, LE).
```

2.4 Efecte laterale

Efectele laterale se referă la manipularea dinamică a definițiilor de predicate și acest lucru se poate realiza cu următoarele predicate predefinite:

- *assert*/1 (= *assertz*/1) → adaugă la sfârșitul bazei de cunoștințe clauza dată în argument;
- *asserta*/1 → adaugă la începutul bazei de cunoștințe clauza dată în argument;
- *retract*/1 → șterge prima clauză care se unifică cu argumentul;
- *retractall*/1 → șterge toate clauzele care se unifică cu argumentul (în SWI-Prolog va merge la succes chiar dacă nu șterge nimic).

Predicatele care sunt definite și/sau apelate în fișierul „.pl" se numesc predicate statice. Predicatele care sunt manipulate cu assert/retract se numesc predicate dinamice. În contrast cu predicatele statice pe care le-am văzut până acum, predicatele dinamice trebuie să fie declarate ca fiind dinamice.

În Prolog, un predicat este static sau dinamic. Un predicat static are faptele/regulile predefinite la începutul execuției și acestea nu se schimbă pe parcursul exercuției. În mod normal, faptele/regulile sunt scrise într-un fisier de cod Prolog care este încărcat în timpul sesiunii Prolog. Dacă vrem să adăugam faptele adiționale (sau chiar reguli) unui prediat, în timpul execuției unei întrebări, vom folosi "assert/asserta/assertz" sau în cazul în care vrem să ștergem fapte vom folosi "retract/retractall". Pentru a face acestea, predicatul trebuie să fie declarat dinamic.

Dacă un predicat este adăugat pentru prima dată cu *assert* atunci este implicit un predicat dinamic. Dacă vrem să manipulăm un predicat care apare în fișierul „.pl" atunci trebuie să-l setăm ca și predicat dinamic adăugând următoare linie la începutul fișierului:

```
:-dynamic <nume_predicat>/<aritate>.
```

Când folosim aceste predicate trebuie să ținem cont de următoarele aspecte:

- **Backtracking-ul nu invalidează efectul la *assert***, ex: dacă un predicat a fost adăugat cu *assert*, rămâne in baza de predicate până când este șters explicit cu *retract*m chiar dacă nodul corespondent acelui apel de *assert* este șters din arborele de execuție (ex: din cauza backtracking-ului).
- Predicatul *assert* **întotdeauna rezultă în succes; nu face backtracking**.
- Predicatul ***retract* poate să eșueze, caz în care va porni backtracking-ul**. În cazul predicatului *retract*, backtracking-ul invalidează temporar ștergerea

pentru predicatele din același corp al clauzei cu apelul predicatului *retract* și care au fost apelate înaintea predicatului *retract*; astfel se păstrează *logical update view*.

Pentru a înțelege mai bine cum funcționează *retract* urmăriți execuția la:

```
?-	assert(insect(ant)),
assert(insect(bee)),
retract(insect(A)),
writeln(A),
retract(insect(B)),
fail.
```

Ați observat probabil că această întrebare ne dă rezultatul:

```
ant
bee
false
```

Deși al doilea *retract* șterge faptul *insect(bee)*, când se face backtracking și se ajunge la primul apel de *retract*, clauza este încă prezentă în logical view- practic, nu vede faptul că clauza a fost ștearsă de al doilea apel de retract. Astfel *insect(A)* încă se poate unifica cu „bee”.

Prolog are și predicatul **retractall/1**, cu următorul comportament: va șterge toate clauzele predicatelor care se potrivesc cu argumentul. În unele versiuni de Prolog, retractall poate da fail dacă nu există nimic de extras. Pentru a evita acest lucru, ați putea să alegeți să faceți un assert a unei clauze dummy de tipul potrivit. Însă, în SWI Prolog, retractall reușeste chiar și pentru un apel cu niciun fapt/regulă care să se potrivească.

Manipularea dinamică a bazei de cunoștințe prin assert/retract este folosită în special pentru salvarea rezultatelor de la calcule (memorisation/caching), astfel încât să nu fie pierdute prin backtracking. Astfel, dacă aceeași întrebare este pusă în viitor, răspunsul poate fi obținut fără a fi nevoie de a recalcula. Această tehnică se numește memorisation, sau caching, în unele aplicații poate să crească eficiența. De asemenea, aceste operații pot fi folosite pentru a schimba dinamic comportamentul unor predicate la rulare (meta-programming). Acest proces duce la un cod greu de urmărit. În cazurile cu mult backtracking, devine și mai greu. Prin urmare, această caracteristică non-declarativă a Prolog-ului ar trebui folosită cu atenție.

Un exemplu de predicat care memorizează rezultatele parțiale, folosind efecte laterale, pentru a calcula al n-lea număr din *secvența Fibonacci* (ați întâlnit varianta mai puțin eficientă în laboratorul al doilea):

```
:-dynamic memo_fib/2.

fib(N,F):- memo_fib(N,F), !.
fib(N,F):-
   N>1,
   N1 is N-1,
   N2 is N-2,
   fib(N1,F1),
   fib(N2,F2),
   F is F1+F2,
   assertz(memo_fib(N,F)).
fib(0,1).
fib(1,1).
```

Urmărește execuția la:
?- listing(memo_fib/2). % afișează toate definițiile predicatului memo_fib cu 2 parametrii
?- fib(4,F).
?- listing(memo_fib/2).
?- fib(10,F).
?- listing(memo_fib/2).
?- fib(10,F).

2.4.1 Afișarea rezultatelor memorizate

Oricând avem nevoie să colectăm toate răspunsurile stocate în baza de predicate prin assert-uri, putem folosi **failure driven loops** care fortează Prolog-ul să facă backtracking până când rămâne fără posibilități. Tiparul pentru un failure driven loop care citește toate clauzele stocate – de exemplu pentru predicatul *memo_fib/2* de mai sus – și printează toate numerele *fibonacci* deja calculate:

```
print_all:-
   memo_fib(N,F),
   write(N),
```

```
    write(' - '),
    write(F),
    nl,
    fail.
print_all.
```

Ne vom folosi de tehnica backtracking (forțând *fail*) pentru a parcurge toate clauzele predicatului *memo_fib/2* adăugate în baza de cunoștințe cu *assert*.

Urmărește execuția la:
?-print_all.
?-retractall(memo_fib(_,_)).
?-print_all.

2.4.2 Colectarea rezultatelor memorizate

Pentru colectarea rezultatelor într-o listă ne putem folosi de un predicat predefinit: *findall*.

Urmărește execuția la:
?- findall(X, append(X,_,[1,2,3,4]), List).
?- findall(lists(X,Y), append(X,Y,[1,2,3,4]), List).
?- findall(X, member(X,[1,2,3]), List).

Vom vedea un exemplu de folosire a efectelor laterale pentru a găsi toate răspunsurile unei întrebări: vom scrie predicatul care generează toate permutările unei liste și le returnează într-o listă. Vrem ca predicatul să funcționeze în felul următor:

?- all_perm([1,2,3],L).
L=[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]];
no.

Știind că deja puteți implementa predicatul *perm/2* – predicatul care generează o permutare a listei de intrare (vedeți documentul despre Metode de Sortare dacă nu) – specificația predicatului *all_perm/2* este:

```
all_perm(L,_):-
    perm(L,L1),
    assertz(p(L1)),
```

```
    fail.
all_perm(_,R):- collect_perms(R).

collect_perms([L1|R]):- retract(p(L1)), !, collect_perms(R).
collect_perms([]).

perm(L, [H|R]):-append(A, [H|T], L), append(A, T, L1), perm(L1, R).
perm([], []).
```

Urmărește execuția la:

```
?- retractall(p(_)), all_perm([1,2],R).
?- listing(p/1).
?- retractall(p(_)), all_perm([1,2,3],R).
```

Întrebări:

1. De ce am nevoie de **retractall** înainte de apelul la **all_perm/2**?
2. De ce este nevoie de ! după **retract** în predicatul **collect_perms/1**?
3. Ce tip de recursivitate este folosit în predicatul **collect_perms/1**? Se poate face colectarea în celălalt tip de recursivitate? Care este ordinea permutărilor în acest caz?
4. Predicatul **collect_perms/1**distruge rezultatele salvate?

3 Exerciții

Înainte de începe exercițiile, adăugați următoarele fapte care definesc arbori (complet și incomplet binar) în script-ul Prolog:

```
% Arbori:
complete_tree(t(6, t(4,t(2,nil,nil),t(5,nil,nil)), t(9,t(7,nil,nil),nil))).
incomplete_tree(t(6, t(4,t(2,_,_),t(5,_,_)), t(9,t(7,_,_),_))).
```

Scrieți un predicat care:

1. Convertește o listă completă într-o listă diferență și viceversa.

```
?- convertCL2DL([1,2,3,4], LS, LE).
LS = [1, 2, 3, 4|LE]

?- LS=[1,2,3,4|LE], convertDL2CL(LS,LE,R).
R = [1, 2, 3, 4]
```

2. Convertește o listă incompletă într-o listă diferență și viceversa.

```
?- convertIL2DL([1,2,3,4|_], LS, LE).
LS = [1, 2, 3, 4|LE]

?- LS=[1,2,3,4|LE], convertDL2IL(LS,LE,R).
R = [1, 2, 3, 4|_]
```

3. Aplatizează o listă adâncă folosind liste diferență în loc de *append.*

```
?- flat_dl([[1], 2, [3, [4, 5]]], RS, RE).
RS = [1, 2, 3, 4, 5|RE] ;
false
```

4. Generează toate descompunerile posibile a unei liste în doua sub-liste fără a folosi predicatul predefinit *findall*.

```
?- all_decompositions([1,2,3], List).
List=[ [[], [1,2,3]], [[1], [2,3]], [[1,2], [3]], [[1,2,3], []] ] ;
false
```

5. Traversează un arbore în *pre-ordine* și încă unul pentru *post-ordine* folosind liste diferență în manieră implicită

```
?- complete_tree(T), preorder_dl(T, S, E).
S = [6, 4, 2, 5, 9, 7|E]

?- complete_tree(T), postorder_dl(T, S, E).
S = [2, 5, 4, 7, 9, 6|E]
```

6. Colectează toate nodurile care au chei pare, dintr-un arbore binar **complet** folosind liste diferență.

```
?- complete_tree(T), even_dl(T, S, E).
S = [2, 4, 6|E]
```

7. Colectează toate nodurile care au chei între K1 și K2, dintr-un arbore binar de căutare **incomplet** folosind liste diferență.

```
?- incomplete_tree(T), between_dl(T, S, E, 3, 7).
S = [4, 5, 6|E]
```

8. Colectează toate cheile de la o adâncime dată K, dintr-un arbore binar de căutare **incomplet** folosind liste diferență.

```
? – incomplete_tree(T), collect_depth_k(T, 2, S, E).
S = [4, 9|E].
```