Liste diferență. 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 este anonimă, în cazul listelor diferență variabila este dată într-un nou parametru. 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.

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 de variabilele S=[1,2,3|X] și E=X. 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.

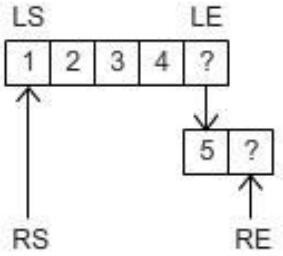
2.2 Predicatul „add”

În cazul listelor complete, adăugarea unui element la finalul listei se realizează în felul următor:

```
add1(X, [H|T], [H|R]):- add1(X, T, R).
add1(X, [], [X]).
```

În cazul listelor diferență, predicatul se va scrie:

```
add2(X, LS, LE, RS, RE):- RS = LS, LE = [X|RE].
% variabila de la finalul listei va conține pe prima poziție
elementul de adăugat
```



Figură 1. Adăugarea unui element la sfârșitul unei liste diferență

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

```
?- LS=[1,2,3,4|LE], add2(5,LS,LE,RS,RE).
LE = [5|RE],
LS = [1,2,3,4,5|RE],
RS = [1,2,3,4,5|RE]
```

2.3 Predicatul „append”

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

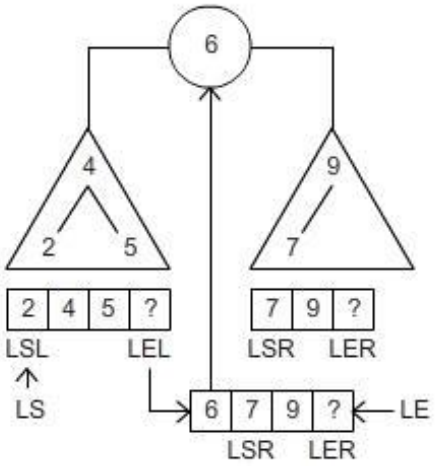
```
append_dl(LS1,LE1, LS2,LE2, RS,RE):- RS=LS1, LE1=LS2, RE=LE2.
```

Exemple de utilizare:

2.3.1 Traversare în ordine

Putem îmbunătății eficiența predicatului *inorder* prin înlocuirea vechiului *append* cu operații pe liste diferență.

```
inorder_dl(nil,L,L). % lista vida este reprezentată de 2 variabile
egale
inorder_dl(t(K,L,R),LS,LE):-
      inorder_dl(L,LSL,LEL), % apel pe subarbore stâng
      inorder_dl(R,LSR,LER), % apel pe subarbore drept
      LS=LSL,
      LEL=[K|LSR], % K este adăugat în fața la LSR
      LE=LER.
```



Figură 2. Traversarea in ordine

Urmărește execuția la:

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

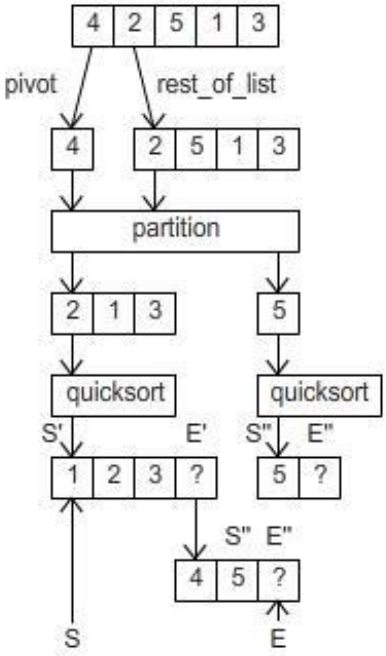
2.3.2 Sortare rapidă

La fel ca mai sus, vom înlocui vechiul *append* cu operații pe liste diferență.

```
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]),
      quicksort_dl(Lg, L, E).
quicksort_dl([], L, L). % condiția de terminare s-a modificat
```

Urmărește execuția la:

```
?- quicksort_dl([4,2,5,1,3], L, []).
?- quicksort_dl([4,2,5,1,3], L, _).
```



Figură 3. Sortare rapidă folosind liste diferență

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. 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*, el poate fi șters numai cu *retract*);
- 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”;
- Predicatul *assert* merge tot timpul la succes;
- Predicatul *retract* poate să eșueze, caz în care va porni backtracking-ul.

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.
```

Deși al doilea *retract* șterge faptul *insect(bee)*, când se face backtracking, primul *retract*, mai exact *insect(A)* nu vede ștergerea lui „bee” și va face unificarea cu *insect(bee)*.

Manipularea dinamică a bazei de cunoștințe este folosită în special pentru salvarea rezultatelor de la calcule (memoisation/caching). De asemenea aceste operații pot fi folosite pentru a schimba dinamic comportamentul unor predicate la rulare (meta-programming).

Exemplu de predicat care memorizează rezultatele parțiale:

```
:-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

Ne vom folosi de tehnica backtracking pentru a parcurge toate predicatele adăugate în baza de cunoștințe cu *assert*.

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

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).
```

Ne vom folosi de efecte laterale pentru a colecta toate permutările unei liste de elemente.

```
all_perm(L,_):- perm(L,L1), % predicatul de generare a unei permutări (vezi lucrare de laborator cu sortări)
                assertz(p(L1)),
                fail.
all_perm(_,R):- collect_perms(R).
```

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

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. De ce este nevoie de ! după *retract* în predicatul *collect_perms*?
3. Ce tip de recursivitate este folosit în predicatul *collect_perms*?
4. Predicatul *collect_perms* distruge rezultatele salvate?

3 Exerciții

Scrieți un predicat care:

1. Convertește o listă incompletă într-o listă diferență și viceversa.
2. Convertește o listă completă într-o listă diferență și viceversa.
3. 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
```

4. 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
```

5. Colectează toate nodurile care au chei pare, dintr-un arbore binar folosind liste diferență.
6. Colectează toate nodurile care au chei între K1 și K2, dintr-un arbore binar folosind liste diferență.