Structuri incomplete (Liste & Arbori)

1 Obiective

În lucrarea de față vom prezenta un caz special de structuri. Structurile incomplete nu se termină într-o constantă (ex: *[]* pentru liste sau *nil* pentru arbori), ci se termină într-o variabilă liberă "_".

2 Considerații teoretice

2.1 Reprezentare

Structurile incomplete (structuri instanțiate parțial) oferă posibilitatea modificării variabilei de la final (instanțiere parțială), ceea ce permite adăugarea de elemente noi în aceeași structură (concatenarea la final nu necesită un argument de output adițional). Pentru a ajunge la variabilă liberă de la final, listele incomplete sunt parcurse în aceeași manieră ca listele complete.

Dar, clauzele care specifică comportamentul când se ajunge la final trebuie să fie explicite (*chiar și în cazul de failure!*). Prin urmare, la condiția de oprire, trebuie sa verificăm dacă am ajuns la variabila liberă (prin folosirea predicatului predefinit *var/1*). Adițional, aceste clauze trebuie să fie plasate înainte celorlalte clauze ale predicatului – deoarece variabila liberă de la final se poate unifica cu orice. Pentru a evita unificări nedorite, acestea trebuie să fie încercate primele.

Exemple:

```
?- L = [a, b, c|_].
?- L = [1, 2, 3|T], T = [4, 5|U].
?- T = t(7, t(5, t(3, _, _), _), t(11, _, _)).
?- T = t(7, t(5, t(3, A, B), C), t(11, D, E)), D = t(9, F, G).
```

2.2 Liste incomplete

Listele incomplete sunt un tip special de structură incompletă. În loc să se termine în [], o listă incompletă are o variabilă incompletă ca și coadă.

Listele incomplete sunt traversate în același fel ca listele complete, folosind șablonul [H|T] ; diferența apare la procesarea finalului de listă – unde nu întâlnim o listă vidă *[]* la final, ci o variabilă liberă. Acest fapt are următoarele implicații asupra predicatelor care folosesc liste incomplete:

- Testarea finalului de listă trebuie să fie facută ***mereu*****,** chiar și în cazurile de *fail* – iar fail-ul trebuie să fie explicit
- La testarea finalului de listă, trebuie să se verifice dacă s-a ajuns la o varibilă liberă:
    - some_predicate(L, …):-var(L), …
- clauza care verifică finalul listei trebuie ***mereu*** să fie prima clauză a predicatului (De ce?)
- orice listă poate fi adăugată la finalul unei liste incomplete, fără să fie nevoie de o structură separată de output (concatenarea la finalul unei liste incomplete se poate face în aceeași structură de intrare):

    e.g. ?- L = [1, 2, 3|T], T = [4, 5|U], U=[6|_]

Ținând aceste observații în minte, vom continua cu transformările unor predicate cunoscute de liste pentru a funcționa si pe liste incomplete.

2.2.1 Predicatul „member”

Înainte de a scrie noul predicat, urmăriți execuția la vechiul *member*:

```
?- L = [1, 2, 3|_], member1(3, L).
?- L = [1, 2, 3|_], member1(4, L).
?- L = [1, 2, 3|_], member1(X, L).
```

După cum ați observat, când elementul apare în listă, predicatul *member1/2* funcționează corect; dar când elementul nu apare în listă, în loc să dea un răspuns negativ, predicatul adaugă elementul la finalul listei incomplete – funcționalitate incorectă. Pentru a corecta acest predicat, trebuie să adăugăm o clauză care specifică explicit fail-ul când se ajunge la finalul listei (la variabila liberă):

```
% trebuie testat explicit faptul ca am ajuns la sfârșitul listei
% ceea ce înseamnă că nu am găsit elementul căutat, așa că apelăm fail
member_il(_, L):-var(L), !, fail.
% celelalte 2 clauze sunt la fel ca în trecut
member_il(X, [X|_]):-!.
member_il(X, [_|T]):-member_il(X, T).
```

Urmărește execuția la noul predicat pe aceleași întrebări.

2.2.2 Predicatul „insert”

După cum ați observat, adăugare unui element la sfârșitul unei liste se poate realiza pe lista de intrare și nu mai este nevoie de un parametru nou pentru lista rezultat – adăugarea se poate face în structura de intrare.

Pentru a face asta, trebuie să parcurgem lista de intrare element cu element și când ajungem la finalul listei, modificăm variabila liberă astfel încât să conțină elementul nou. Dacă elementul deja există, atunci nu îl mai adăugăm.

```
% am ajuns la finalul listei atunci putem adăuga elementul
insert_il1(X, L):-var(L), !, L=[X|_].
insert_il1(X, [X|_]):-!. % elementul există deja
insert_il1(X, [_|T]):- insert_il1(X, T). %traversăm lista să ajungem la final / X
```

Observație. Există similarități între *insert_il* și *member_il* – singura diferență este funcționalitatea când ajung la finalul listei.

De asemenea, predicatele *insert_il* și *member1* sunt foarte similare. De fapt, dacă comparăm execuția de aproape, produc același comportament! Înseamnă că testarea apartenenței într-o listă completă este echivalentă cu inserarea într-o listă incompletă – putem să stergem prima clauză a predicatului *insert_il* (De ce?).

```
insert_il2(X, [X|_]):-!.
insert_il2(X, [_|T]):- insert_il2(X, T).
```

Funcționalitatea primei clauze poate fi realizată și de a doua clauză. Dacă elementul căutat nu a fost găsit înseamnă că am ajuns la finalul listei. Caz în care variabila din coada se va unifica cu [X|_], ceea ce înseamnă că se inserează elementul căutat la sfârșitul listei.

Urmărește execuția la:

```
?- L = [1, 2, 3|_], insert_il2(3, L).
?- L = [1, 2, 3|_], insert_il2(4, L).
?- L = [1, 2, 3|_], insert_il2(X, L).
```

2.2.3 Predicatul „delete”

În predicatul *delete/3* vom păstra aceeași variabilă neinițializată de la final și pentru lista de intrare și pentru lista rezultat. În rest predicatul este la fel cu varianta de la liste complete.

```
delete_il(_, L, L):-var(L), !. % am ajuns la finalul listei
delete_il(X, [X|T], T):- !. % găsim, ștergem prima apariție și ne oprim
delete_il(X, [H|T], [H|R]):- delete_il(X, T, R). % traversăm și căutam elementul
```

Observație. În condiția de oprire, care corespunde finalul listei de intrare, este prima clauză și are forma știută. Clauzele 2 și 3 sunt aceleași care apar la predicatul *delete/3* pentru liste complete.

Urmărește execuția la:

```
?- L = [1, 2, 3|_], delete_il(2, L, R).
?- L = [1, 2, 3|_], delete_il(4, L, R).
?- L = [1, 2, 3|_], delete_il(X, L, R).
```

2.3 Arbori incompleți

Arborii incompleți sunt un caz special de structură incompletă – o ramură nu se mai termină cu *nil*, ci cu o variabilă liberă.

Observațiile făcute pentru scrierea predicatelor pe liste incomplete se aplică și în cazul arborilor incompleți. Prin urmare, vom aplica aceste observații pentru a dezvolta predicatele discutate pentru liste în secțiunea anterioară: *search_it/2, insert_it/2, delete_it/3* – pentru arbori binari de căutare incompleți.

2.3.1 Predicatul „search”

La fel ca în cazul listelor, predicatul care face căutarea unei chei într-un arbore complet nu se descurcă pe un arbore incomplet – trebuie să adăugăm explicit o clauză pentru situațiile de fail (când se ajunge la variabila liberă):

```
search_it(_, T):- var(T), !, fail.
search_it(Key, t(Key, _, _)):- !.
search_it(Key, t(K, L, _)):- Key<K, !, search_it(Key, L).
search_it(Key, t(_, _, R)):- search_it(Key, R).
```

Urmărește execuția la:
?- T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), search_it(6, T).
?- T=t(7, t(5, t(3,_,_), _), t(11,_,_)), search_it(9, T).

2.3.2 Predicatul „insert”

Deoarece inserarea unui nod nou se realizează în frunze (la finalul structurii), ne putem folosi de același arbore de intrare și pentru rezultat, fără să avem nevoie de un argument adițional de ieșire. Dacă ne întoarcem la analogia cu listele incomplete, găsim că predicatul care face căutarea pe structura completă se va comporta ca un predicat care inserează pe arbori incompleți (De ce?). Prin urmare, nu avem nevoie de un argument pentru arborele rezultat.

```
% inserează sau verifică dacă elementul există deja în arbore
insert_it(Key, t(Key, _, _)):-!.
insert_it(Key, t(K, L, _)):-Key<K, !, insert_it(Key, L).
insert_it(Key, t(_, _, R)):-insert_it(Key, R).
```

Urmărește execuția la:
?- T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), insert_it(6, T).
?- T=t(7, t(5, t(3,_,_), _), t(11,_,_)), insert_it(9, T).

2.3.3 Predicatul „delete”

Ștergerea dintr-un arbore incomplet de căutare este similară cu ștergerea dintr-un arbore complet. La fel ca în cazul predicatului *delete* de la liste incomplete vom păstra aceleași variabile neinițializate din arborele de intrare și pe arborele rezultat.

```
delete_it(_, T, T):- var(T), !.  % elementul nu este parte din arbore
delete_it(Key, t(Key, L, R), L):- var(R), !.
delete_it(Key, t(Key, L, R), R):- var(L), !.
delete_it(Key, t(Key, L, R), t(Pred,NL,R)):- !, get_pred(L,Pred,NL).
delete_it(Key, t(K,L,R), t(K,NL,R)):- Key<K, !, delete_it(Key,L,NL).
delete_it(Key, t(K,L,R), t(K,L,NR)):- delete_it(Key,R,NR).

% caută nodul predecesor
get_pred(t(Pred, L, R), Pred, L):- var(R), !.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):- get_pred(R, Pred, NR).
```

Urmărește execuția la:
?- T=t(7, t(5, t(3,_,_), t(6,_,_)), t(11,_,_)), delete_it(6, T, R).
?- T=t(7, t(5, t(3,_,_), _), t(11,_,_)), delete_it(9, T, R).

3 Exerciții

```
% Arbori:
incomplete_tree(t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _))).
complete_tree(t(7, t(5, t(3, nil, nil), t(6, nil, nil)), t(11, nil, nil))).
```

Scrieți un predicat care:

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

```
?- convertIL2CL([1,2,3|_], R).
R = [1, 2, 3].

?- convertCL2IL([1,2,3], R).
R = [1, 2, 3|_].
```

2. Concatenează 2 liste incomplete (rezultatul este tot o listă incompletă – *De câte argumente este nevoie, am putea renunța la unul?*).

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

3. Inversează o listă incompletă (rezultatul este tot o listă incompletă). Implementați în ambele tipuri de recursivitate.

```
?- reverse_il_fwd([1,2,3|_], R).
R = [3, 2, 1|_].

?- reverse_il_bwd([1,2,3|_], R).
R = [3, 2, 1|_].
```

4. Aplatizează o listă adâncă incompletă (rezultatul este o listă simplă incompletă).

```
?- flat_il([[1|_], 2, [3, [4, 5|_]|_]|_], R).
R = [1, 2, 3, 4, 5|_] ? ;
false.
```

5. Convertește un arbore incomplet într-un arbore complet și viceversa.

?- incomplete_tree(T), convertIT2CT(T, R).
R = t(7,t(5,t(3,nil,nil),t(6,nil,nil)),t(11,nil,nil))

?- complete_tree(T), convertCT2IT(T, R).
R = t(7, t(5, t(3, _, _), t(6, _, _)), t(11, _, _))

6. Traversează un arbore incomplet în pre-ordine (cheile se adaugă într-o listă incompletă).

?- incomplete_tree(T), preorder_it(T, R).
R = [7, 5, 3, 6, 11|_]

7. Calculează înălțimea unui arbore incomplet.

?- incomplete_tree(T), height_it(T, R).
R=3

8. Calculează diametrul unui arbore incomplet.

$$diam(T) = \max\{diam(T.left), diam(T.right), height(T.left) + height(T.right) + 1\}$$

?- incomplete_tree(T), diam_it(T, R).
R=4

9. Determină dacă o listă incompletă este o sub-listă într-o altă listă incompletă.

?- subl_il([1, 1, 2|_], [1, 2, 3, 1, 1, 3, 1, ***1, 1, 2***|_]).
true

?- subl_il([1, 1, 2|_], [1, 2, 3, 1, 1, 3, 1, 1, 1, 3, 2|_]).
false