

PL/Rodica Potolea (vol1)

Prefața

Începuturile limbajului Prolog sunt undeva în anii '70, când lucrând la idei similare, Robert Kowalski la Edingburg, și Alain Colmerauer la Marseille au ajuns la formularea filosofiei programării logice, a primului model computațional (Kowalski, 1974), și respectiv la proiectarea și implementarea primului limbaj logic, Prolog (Colmerauer, 1973). Paradigma a căpătat notorietate și datorită dezvoltării unor tehnici eficiente de implementare (Warren, 1977), compilatorul implementat de acesta identificând cazuri speciale de unificare, care sunt translatate în secvențe eficiente de operații cu memoria.

Principiul rezoluției și al unificării (Robinson 1965) stau la baza demonstrării automate de teoreme și a programării logice. Rezoluția SLD (Kowalski, 1973-74) restricționează rezoluția SL (Colmerauer și Kowalski, 1971) la domeniul clauzelor definite. La începutul anilor '80 Lloyd propune modelul formal al paradigmei. În lucrarea sa [7] este prezentat modelul formal al programării logice, cu demonstrarea corectitudinii și completitudinii acestuia. Într-o abordare sistematică și riguroasă, sunt prezentate semantica declarativă și operațională, și extensiile programării logice.

Sfârșitul anilor '70, marchează deja existența câtorva sisteme de programare logică secvențială disponibile. Deceniul 8 se remarcă prin deschiderea către îmbogățirea paradigmei logice cu noi paradigme specifice altor sisteme de programare: paralelismul și concurența (IC-Prolog, Shapiro, 1983, Clark 1982, PARLOG, Clark și Gregory, 1984, GHC, Ueda, 1985, Concurrent Prolog), obiectualitatea și funcționalitatea, iar deceniul 9 a adus paradigma constrângerilor [8], [12], [14] în prim-plan, ca paradigmă adăugată peste cea a programării logice. Pasul către paralelism și concurență este unul absolut firesc. Mai mult ca orice altă paradigmă de programare, programarea logică este deținătoarea unor surse implicite de paralelism. Pornind de la însăși definiția modelului programării logice, regulile de calcul și selecție sunt unele generice; modelul nu precizează în nici un fel cum se *alege* scopul de evaluat, respectiv clauza pentru unificare, această alegere dând specificitate limbajului (cum am mai menționat, pentru majoritatea limbajelor logice secvențiale, această alegere este specifică, fiind cel mai din stânga, respectiv prima clauză prima dată, iar în lucrarea de față, toate exemplele se supun acestor reguli). Dacă la aceste grade de libertate mai adăugăm însăși unificarea, rezultă trei grade de libertate, care conferă tot atâtea surse de paralelism (concurență). Dacă regulile de căutare și selecție devin unele

paralele, avem o altă reprezentare a modelului logic (de fapt câte o reprezentare pentru fiecare pereche de reguli), cu specificul său, cu generalizări și limitări particulare. În treacăt menționăm că puternicul mecanism de backtracking, cu avantajele sale, nu mai este prezent la implementările logice paralele și concurente, tocmai datorită faptului că la aceste limbaje mecanismul este acela al căutării simultane a soluției în mai multe (eventual toate) ramurile spațiului de căutare, iar găsirea soluției pe una din ramuri produce abandonarea căutării pe celelalte ramuri. De menționat că cea de-a treia sursă de paralelism (unificarea) este o sursă reală și valoroasă, dar la o granularitate diferită, și anume paralelism în unificarea perechilor de argumente (formale/actuale). Obiectualitatea, funcționalitatea și constrângerile, ca mecanisme adăugate paradigmei logice au toate ca și justificare pătrunderea programării logice spre domenii noi, altele decât cele care au stat la originea definirii sale, totodată cu creșterea eficienței anumitor tipuri de operații.

În acest moment există disponibile diverse modele de programare logică, care au înglobat și alte paradigme. Unul dintre sistemele reprezentative este Sicstus Prolog, proiectat și implementat la origine la *Swedish Institute of Computer Science*, Suedia. Sistemul este continuu dezvoltat, lucrându-se în permanență atât la întreținerea sa, cât și adăugarea de noi funcționalități. În momentul de față versiunea cea mai recentă este SICStus 4, disponibilă pentru platforme Windows, Linux, Solaris, Irix, Mac, Aix, HP-UX . Începând de la versiunea inițială, la realizare au colaborat mai multe colective internaționale de cercetători din Europa și America. Limbajul s-a extins, având înglobat paradigme ca programarea cu constrângeri (boleene, domenii finite, domeniul rațional și cel real, chiar și un depanator pentru programarea cu constângeri fiind înglobat), obiecte, programare pe web, socketuri, intergfețe cu C++, Java, Basic și cu BD externă. De asemenea sunt implementate la nivelul bibliotecilor operații de bază (liste, cozi, heapuri, șiruri, seturi, operații pe arbori binari și AVL). Toate exemplele din această carte au fost rulate în SICStus Prolog 3.12.5, ultima versiune disponibilă în momentul în care au fost efectuate rulările.

Sper ca această carte să reprezinte o introducere lentă și plăcută în fascinantul domeniu al programării logice. Ea se adresează în primul rând studenților care se inițiază în acest domeniu. Sunt oferite explicații detaliate ale noțiunilor prezentate, mai degrabă informale (formalismul poate fi regăsit [7] și [11]). Noțiunile și exercițiile sunt prezentate gradual, de la simplu la complex. Unele au prezentate explicații complete, cu codul și rularea execuției (chiar cu trasare), evidențiindu-se eventualele capcane.

Altele sunt tratate mai sumar, fiind prezentat doar codul. Există și cea de-a treia categorie, a exercițiilor propuse spre rezolvare, lăsate în seama cititorului. Majoritatea exercițiilor prezentate sunt originale. Autorii acestora sunt membrii colectivului Programare Logică Multiparadigmă din Catedra de Calculatoare, Universitatea Tehnică din Cluj-Napoca.

Cartea reprezintă primul volum al unei lucrări în două volume, și cuprinde 10 capitole. Primul prezintă o introducere sumară în domeniul programării logice. Sintaxa este prezentată, împreună cu mecanismele fundamentale: unificarea, backtrackingul și tăierea de backtracking, negația. Capitolul al doilea face o descriere amănunțită a modalității de scriere a unui program logic, cu exemplificare pe mulțimi reprezentate ca și liste, fiind analizate trei exemple. Mecanismul fundamental al programării logice (unificarea) este tratat în detaliu în capitolul 3. Pe diferite exemple este tratată problema unificării prin intermediul construirii arborilor de execuție. Operații elementare pe liste sunt prezentate în capitolul următor. Se pornește de la exemple simple ajungându-se la cele complexe. Se evidențiază posibilitatea de obținere de soluții alternative la o problemă (un exercițiu având 5 soluții propuse), realizându-se și o paralelă între predicate similare, dar cu semantică distinctă. Pe de altă parte în acest capitol se evidențiază utilizarea multiplă, în diferite contexte a anumitor predicate, și de asemenea caracterul nedeterminist al acestora. Tipurile de recursivitate sunt analizate prin intermediul unor exerciții prezentate în același capitol 4. Capitolul 5 prezintă soluția în paradigma logică a metodelor directe de sortare. Aceste exerciții reprezintă de asemenea pretext pentru exersarea și analiza comparativă a tipurilor de recursivitate, și de prezentare a buclilor repetitive în programarea logică. Arborii sunt prezentați în prima parte a capitolului 6, partea a doua fiind dedicată structurilor incomplete, liste și arbori terminați în variabilă. Capitolul 7 este dedicat unui mecanism menit să îmbunătățească eficiența programelor logice, și anume listele diferență. Acestea sunt prezentate gradual, în numeroase exemple. Capitolul este bogat în exerciții care evidențiază expresivitatea limbajului. Structurile complexe sunt analizate în capitolul 8, iar capitolul 9 este dedicat efectelor laterale (assert și retract) exemplificate pe arbori AVL. Ultimul capitol este dedicat referințelor bibliografice fundamentale ale domeniului, completate cu o listă de autor.

La nivelul Catedrei de Calculatoare din Universitatea Tehnică din Cluj-Napoca interesul spre programarea logică a fost stimulat odată cu sosirea profesorului Tudor Mureșan în catedră. Încă de la început a introdus Programarea Logică ca și disciplină de studiu, lansând și activitatea de

cercetare/dezvoltare în această direcție. Prezenta lucrare este rodul experienței pe care colectivul inițiat, format, menținut și coordonat de Tudor Mureșan a acumulat-o în cei 18 ani de activitate didactică și de cercetare în acest domeniu. Îi mulțumesc pentru dăruirea cu care ne-a împărtășit experiența și cunoștințele acumulate de-a lungul carierei sale, într-o manieră de mare maestru. Îi sunt recunoscătoare în mod deosebit pentru a fi unul dintre principalii beneficiari ai formării ca dascăl și cercetător în cadrul acestui colectiv, prin constituirea căruia s-a deschis un domeniu nou în catedra. Această direcție s-a dezvoltat continuu, numeroare generații de absolvenți având reprezentanți care își desfășoară activitatea de cercetare de specialitate în centre de renume. Îi mulțumesc de asemenea pentru răbdarea și grija cu care a citit manuscrisul prezentei lucrări chiar din momentul în care era format doar din câteva fascicule. Alături de mulțumirile adresate dumnealui, vreau să mulțumesc colectivului pe care l-a format. Lui Enea Todoran, unul dintre primii colaboratori, care a fost un membru al acestui grup chiar imediat după formarea lui, și împreună cu care am realizat primul îndrumător de laborator. Mulțumiri adresez și colegului Alin Suciu, care după ce a fost un student aparținând uneia dintre primele generații care a studiat cu mare interes acest domeniu, s-a alăturat colectivului nostru. Îi mulțumesc pentru colaborarea profesională pe care o avem, și mai ales pentru continuitatea acestei colaborări, pentru că a ales să rămână fidel domeniului și locului în care s-a format. Mulțumesc tuturor celor care de-a lungul timpului au interacționat direct cu acest colectiv. Mulțumesc de asemenea colegilor din Catedra de Calculatoare a UTCN, actuali dar și foști colegi, pentru mediul profesional și colegial pe care l-au creat. Mulțumesc celor 14 generații de studenți ai specializării Calculatoare pentru interesul manifestat domeniului, pentru creativitatea, inteligența și dăruirea cu care unii dintre ei s-au aplecat spre acest domeniu. Vreau să remarc faptul că unele dintre soluțiile prezentate în această lucrare sunt și rezultatul discuțiilor avute la clasă cu ei. Nu în cele din urmă, mulțumesc în mod deosebit tuturor celor care au contribuit la apariția în bune condiții a acestei lucrări.

Cluj-Napoca
Martie 2007

Rodica Potolea

Cuprins

1. Sintaxa.....	1
1.1 Termeni	1
1.2 Clauze, fapte, întrebări	1
1.3 Interpretor abstract	6
1.4 Unificarea.....	9
1.5 Programarea în Prolog pur	12
1.6 Compararea cu limbajele de programare imperative	16
1.7 Tăierea de backtraking și negația	18
1.7.1 Tăierea de backtraking (!)	18
1.7.2 Negația	20
2. Operații pe mulțimi	21
2.1 Reuniunea.....	22
2.2 Intersecția	29
2.3 Diferența.....	31
2.4 Exerciții și probleme propuse.....	32
3. Arbori de execuție	34
3.1 Member	34
3.2 Append	36
3.3 Reuniune	38
3.4 Intersecție	43
3.5 Diferența.....	49
3.6 Exerciții și probleme propuse.....	56
4. Operații pe liste	60
4.1 Apartenența la o listă.....	60
4.2 Concatenarea listelor	61
4.3 Eliminarea duplicatelor	67
4.4 Ștergerea unui element dintr-o listă	68
4.5 Înlocuirea unui element într-o listă	71
4.6 Lungimea unei liste	72
4.7 Tipuri de recursivitate	77
4.8 Inversarea unei liste.....	82
4.9 Sublistă.....	87
4.10 Exerciții și probleme propuse.....	92
5. Sortarea	97
5.1 Sortarea prin generarea permutărilor	97
5.2 Sortarea prin selecție	103

II

5.3 Sortarea prin inserție	111
5.4 Sortarea prin interschimbare	115
5.5 Exerciții și probleme propuse	123
6. Structuri de date în Prolog	126
6.1 Arbori	126
6.1.1 Căutarea	127
6.1.2 Inserarea	128
6.1.3 Ștergerea.....	130
6.1.4 Traversare.....	142
6.1.5 Exerciții și probleme propuse	146
6.2 Structuri incomplete (terminate în variabile)	148
6.2.1 Liste incomplete (terminate în variabilă)	149
6.2.2 Arbori incompleți (terminați în variabilă).....	156
6.2.3 Exerciții și probleme propuse	162
7. Liste diferență	166
7.1 Transformarea arborilor de căutare în liste	166
7.1.1 Inordinea	167
7.1.2 Preordinea	171
7.1.3 Postordinea.....	173
7.2 Algoritmi de tip divide et impera	175
7.2.1 Sortarea prin partiționare	176
7.2.2 Exerciții și probleme propuse	181
7.3 Transformarea între diferite tipuri de liste	184
7.3.1 Listă completă la listă incompletă.....	184
7.3.2 Listă incompletă la listă completă.....	184
7.3.3 Listă completă la listă diferență	185
7.3.4 Listă diferență la listă completă	185
7.3.5 Listă incompletă la listă diferență	187
7.3.6 Listă diferență la listă incompletă	188
7.3.7 Concatenări între diferite tipuri de liste	190
7.4 Transformări în arbori	191
7.4.1 Transformări prin rotații.....	192
7.4.2 Transformare prin creare copie	198
7.4.3 Exerciții și probleme propuse	200
8. Structuri complexe	203
8.1 Arbori de liste - structuri complete	203
8.1.1 Generare listă	203
8.1.2 Căutare	206
8.2 Arbori de liste - structuri incomplete	209
8.2.1 Căutare	210
8.2.2 Generare listă	213

8.3 Liste adânci	214
8.4 Exerciții și probleme propuse.....	221
9. Accesarea și manipularea programului	223
9.1 Efecte laterale: assert și retract.....	223
9.2 Arbori AVL.....	226
Bibliografie	242

1. Sintaxa

Construcțiile fundamentale ale programării logice sunt relativ puține, dacă le comparăm cu cele ale altor paradigme din programare, și sunt moștenite din logică: termeni și propoziții (statements). La rândul lor, propozițiile sunt de trei tipuri: reguli, fapte și întrebări (vom vedea că faptele și întrebările nu reprezintă altceva decât tipuri particulare de reguli). Regulile permit definirea unor relații în termeni ai unor relații existente.

1.1 Termeni

Un termen este o constantă, o variabilă, sau un termen compus. Constantele reprezintă tipuri particulare de termeni, cum sunt întregii și atomii, iar o variabilă reprezintă o entitate singulară, deși nespecificată. Simbolic un atom se reprezintă printr-o secvență de caractere, în timp ce o variabilă începe în mod obligatoriu cu o majusculă.

Deci 1, 2, 3.14, a, abc reprezintă constante, în timp ce A, Abc reprezintă variabile.

Un termen compus este format dintr-un functor și eventual o secvență de unul sau mai mulți termeni, argumentele functorului. Functorul se caracterizează prin nume (care este un atom) și aritate (număr de argumente). De exemplu functorul $f(t_1, t_2, \dots, t_n)$, cu f reprezentând functorul, iar n aritatea, se identifică sub forma f/n . Functori cu același nume dar arități diferite sunt diferiți, ceea ce înseamnă că $f(t_1, t_2, \dots, t_n)$ și respectiv $f(t_1, t_2, \dots, t_m)$ sunt diferiți dacă $n \neq m$, pentru că au f/n respectiv f/m .

1.2 Clauze, fapte, întrebări

Regulile sunt cele care stipulează relațiile dintre obiectele pe care programul logic le manipulează (reamintim că în paradigma programării logice accentul cade pe relațiile între obiectele implicate, mai mult decât pe

descrierea pașilor de executat în scopul rezolvării problemei). Forma generală a regulilor este cea a clauzelor Horn, din logica predicatelor de ordinul întâi. Toate regulile care se referă la relația între o submulțime anume de obiecte sunt grupate într-un predicat; un predicat poate fi format din una sau mai multe clauze. (O clauză poate fi interpretată ca o funcție în accepțiunea programării procedurale). În cadrul unui predicat logic toate clauzele unui predicat sunt grupate (sunt contigui) nefiind permisă intercalarea clauzelor altui predicat între ele.

În programarea logică o regulă este identică cu o propoziție logică (din logica matematică), cu restricția semanticii programării logice. Adică, o propoziție logică de forma:

Dacă P și Q atunci R .

unde P , Q , R sunt la rândul lor propoziții logice (definite anterior, elementare sau atomice), atunci transcrierea în limbaj de programare logic este sub forma unei clauze Horn:

$r \leftarrow p, q.$

Forma generală a unei clauze Horn (numită din acest punct simplu: clauză) este:

$A \leftarrow B_1, B_2, \dots, B_n.$

Cu $n \geq 0$, A reprezintă capul clauzei, B_i , $i=1, n$ reprezintă corpul. Corpul unei clauze reprezintă o conjuncție de scopuri, fiecare subscop din conjuncție reprezentând apelul unui predicat definit în cadrul aceluiași program. O clauză poate fi privită ca o regulă: corpul reprezintă ipoteza, în timp ce capul reprezintă concluzia respectivei reguli. Având în vedere că o regulă poate avea mai multe alternative (sau la o aceeași concluzie se poate ajunge prin diferite raționamente), această posibilitate este acoperită la clauzele Horn prin posibilitatea definirii mai multor clauze ale aceluiași predicat, adică:

$A_1 \leftarrow B_{11}, B_{12}, \dots, B_{1n}.$

$A_2 \leftarrow B_{21}, B_{22}, \dots, B_{2n}.$

.

.

.

$A_m \leftarrow B_{m1}, B_{m2}, \dots, B_{mn}.$

Aşa cum deja am menţionat, programarea logică posedă trei tipuri de propoziţii: reguli, fapte şi întrebări. Dintre acestea, regulile au forma prezentată mai sus, faptele şi întrebările fiind tipuri particulare de propoziţii, au şi forme particulare, şi anume, o clauză fără corp (în regulă, $n=0$) reprezintă un fapt, şi sunt afirmaţii necondiţionate:

A.

în timp ce o clauză fără cap reprezintă o întrebare:

$\leftarrow B_1, B_2, \dots, B_n.$

La rândul lor, întrebările pot fi întrebări simple (unitare, elementare), formate dintr-un singur subscop (apel de regulă)

$\leftarrow B_i.$

Sau compuse, formate din minim două subscopuri (apeluri de reguli ale aceluiaşi predicat, sau ale unor predicate distincte):

$\leftarrow B_1, B_2, \dots, B_n.$ sau $\leftarrow B_1, C_1, \dots, L_1.$

Faptele reprezintă relaţii existente între obiecte, adevărate necondiţionat. Ele reprezintă axiome. Întrebările reprezintă relaţii potenţiale între obiecte; răspunsul la întrebări stipulează existenţa sau inexistenţa relaţiilor între obiectele care apar în întrebare. Dacă un program logic este văzut ca o colecţie de axiome (faptele) şi reguli (clauzele), o întrebare reprezintă o teoremă, iar execuţia programului pentru obţinerea unui răspuns la întrebare reprezintă demonstrarea teoremei, cu ajutorul axiomelor şi regulilor date prin programul logic. În fapt rezolvarea unei probleme se reduce la căutarea şi obţinerea soluţiei (răspunsul la întrebare), soluţie care poate fi dedusă (demonstrată) pe baza clauzelor şi faptelor (regulilor şi axiomelor) din programul logic.

Regulile (şi forma lor particulară, faptele) pot conţine şi variabile (argumente de tip variabilă, cuantificatorul universal fiind implicit presupus), contextul (scop) acestora (raza de acţiune) fiind regula. Regula anterioară îmbogăţită cu argumente conduce la forma generală a unei reguli, şi anume:

$A(X, Y) \leftarrow B_1(X), B_2(Y), \dots, B_n(X, Y).$

Regulile pot fi văzute în două feluri. În primul rând, ele reprezintă modalități de exprimare a unor întrebări complexe în termenii unor întrebări simple (dacă B_i , $i=1,n$, sunt cunoscuți ca fiind adevărați, atunci $A \leftarrow B_1, B_2, \dots, B_n$ reprezintă doar afirmarea unei astfel de întrebări complexe). O întrebare de forma $\leftarrow A$ adresată programului care conține regula anterior menționată pentru A este tradusă, în conformitate cu regula, în $\leftarrow B_1, B_2, \dots, B_n$ și rezolvată după regulile de rezolvare ale întrebărilor logice. O astfel de interpretare a regulilor se referă la interpretarea *procedurală*. Interpretarea procedurală a regulii anterioare este: *"Pentru a afla răspunsul la întrebarea dacă A este adevărat pentru argumentele X și Y , găsește mai întâi răspunsul la conjuncția de întrebări specifice despre $B_1(X)$ și $B_2(Y)$, și, ..., și $B_n(X, Y)$."*

A doua modalitate în care pot fi privite clauzele este cea menționată deja de reguli, o conjuncție de alte reguli și/sau axiome. Săgeata inversă \leftarrow este utilizată pentru a indica implicația logică. Regula pentru A interpretându-se: *"Pentru orice X și Y , X și Y relaționează prin A dacă relația definită prin B_1 pentru X este adevărată, și relația definită prin B_2 pentru Y este adevărată, și, ..., și în cele din urmă relația definită prin B_n pentru X și Y este adevărată."* În această viziune regulile reprezintă modalități de exprimare a unor noi și complexe relații (A), utilizând relații simple cunoscute (B_i , $i=1,n$). Astfel, predicatul A a fost definit în termeni ai predicatului (predicatelor) B_i , $i=1,n$. Această interpretare asociată cu regula se referă la *semantica declarativă*. Conform semanticii declarative, interpretarea regulii este: *" A este adevărat dacă fiecare termen al conjuncției B_1, B_2, \dots, B_n este adevărat."*

Exemplificăm modelul generic al regulilor cu un caz particular:

```
male(ion) .
male(radu) .
female(maria) .
female(ana) .
parent(ion,radu) .
parent(ion,maria) .
father(X,Y) ← male(X), parent(X,Y) .
```

Deși, așa cum deja am menționat, toate variabilele unei reguli sunt cuantificate universal, domeniul lor de existență fiind regula, există și

variabile cuantificate existențial. Astfel de variabile sunt variabile care apar în corpul clauzei dar nu și în capul clauzei. De exemplu:

$$A(X, Y) \leftarrow B_1(X, Z), B_2(Z, Y).$$

Poate fi interpretată ca: "*Pentru orice X și Y , X și Y relaționează prin A dacă există Z pentru care relația definită prin B_1 pentru X și Z este adevărată, și relația definită prin B_2 pentru Z și Y este adevărată.*" Astfel de variabile (existențiale) se numesc variabile ascunse.

În primul nostru exemplu de program logic toate variabilele din regula $\text{father}(X, Y) \leftarrow \text{male}(X), \text{parent}(X, Y).$ sunt cuantificare universal. Dacă însă îmbogățim programul, definind relația bunic, atunci o regulă de forma:

$\text{grandfather}(X, Z) \leftarrow \text{father}(X, Y), \text{father}(Y, Z).$
are variabila Y cuantificată existențial.

În acest context, putem acum defini un program logic, și anume, el este o mulțime finită de reguli. Pe de altă parte, un scop cuantificat existențial (întrebare) G este o consecință logică a unui program P dacă există în program o instanță legată (o instanță în care variabilele cuantificate universal sunt particularizate la cuantificările existențiale din întrebarea G) $A \leftarrow B_1, B_2, \dots, B_n.$ astfel încât B_1, B_2, \dots, B_n sunt la rândul lor consecințe logice ale programului, iar A o instanță a lui G . De subliniat că G este o consecință logică a lui P dacă și numai dacă G poate fi dedus din P printr-un număr finit de aplicări ale regulilor lui P .

În exemplul anterior, întrebarea G de forma $\leftarrow \text{father}(\text{ion}, \text{Son}).$ reprezintă o consecință logică a lui P dacă există o instanță legată a clauzei (sau a uneia dintre clauzele lui father , în condițiile în care P ar conține mai multe clauze ale lui father) $\text{father}(X, Y) \leftarrow \text{male}(X), \text{parent}(X, Y).$ astfel încât $\text{male}(X)$ și $\text{parent}(X, Y)$ sunt la rândul lor consecințe logice ale lui P . Se observă pe de o parte că instanța legată pentru întrebarea G este $\text{father}(\text{ion}, Y) \leftarrow \text{male}(\text{ion}), \text{parent}(\text{ion}, Y).$, și pe de altă parte că $\text{male}(\text{ion})$ și respectiv $\text{parent}(\text{ion}, Y)$ sunt consecințe logice ale lui P . Dacă pentru $\text{male}(\text{ion})$ consecința logică este imediată (reprezentând chiar un fapt al lui P , deci o axiomă), pentru $\text{parent}(\text{ion}, Y)$ se aplică în acest scop încă o dată definiția, și pentru întrebarea $\leftarrow \text{parent}(\text{ion}, Y)$ se găsește în P instanța legată $\text{parent}(\text{ion}, \text{radu}).$

Operațional, răspunsul la o întrebare reflectă definiția consecinței logice. Trebuie găsită ("ghicită") o instanță legată a unui scop, și o instanță legată a unei reguli, și apoi recursiv, trebuie răspuns la întrebarea compusă (conjuncția) corespunzătoare corpului regulii respective. Astfel, pentru a demonstra un scop A , prin intermediul programului P , trebuie aleasă o regulă $A_1 \leftarrow B_1, B_2, \dots, B_n$ din P și identificată o substituție θ (vezi capitolul unificare) astfel încât $A = A_1\theta$, iar $B_i\theta$ este legat, pentru fiecare $i=1,n$; apoi recursiv, se demonstrează fiecare din $B_i\theta$. Acest proces implică lanțuri de raționamente de lungime arbitrară, fiind dificilă ghicirea instanței legate și alegerea regulii potrivite. În realitate, toate limbajele logice au definit exact mecanismul prin care acest proces devine unul riguros, și pentru programarea logică secvențială el este prezentat în capitolul următor.

Colecția de reguli având același nume în capul clauzei se numește predicat (sau procedură). Din punct de vedere a interpretării operaționale, un predicat este analog unei proceduri din limbajele de programare convenționale (imperative).

1.3 Interpretor abstract

Semantica operațională, care definește mecanismul prin care se obține un răspuns la o întrebare, corespunde unui interpretor abstract. Principal, mecanismul este cel descris anterior, la interpretarea operațională a consecinței logice.

Interpretorul preia o întrebare G (legată sau nu) și un program P , furnizând la ieșire răspunsul *da* (și eventual o instanță a unei variabile cuantificate universal din G , dacă aceasta nu este legată) dacă G este deductibilă din P , sau răspunsul *nu* în caz contrar. De asemenea interpretorul poate eșua în a termina execuția, în situația în care G nu este deductibil din P , caz în care nu se furnizează nici un fel de răspuns. Întrebarea căreia interpretorul trebuie să-i găsească răspunsul se numește *rezolvent*. În exemplul anterior, la întrebarea $G \equiv \leftarrow \text{father}(\text{ion}, \text{Son})$. formulată programului P se obține răspunsul *da*, și $\text{Son}=\text{radu}$.

Pașii unui astfel de interpretor sunt:

Intrare: întrebarea (scopul) G , programul P
 Ieșire: *da*, dacă s-a găsit o demonstrație a lui G prin P
nu, în caz contrar.

Algoritm: **inițializează** rezolventul la întrebarea inițială G
 while rezolventul A_1, A_2, \dots, A_m e **nevid**
 alege un scop $A_i, i=1, m$;
 alege o instanță legată a unei clauze
 $A \leftarrow B_1, B_2, \dots, B_n, \dots$ din P cu proprietatea că $A_i = A\theta$
 (unde θ reprezintă substituția care conduce la instanța
 legată a clauzei). Dacă nu există o astfel de clauză, **ieși**
 din bucla while;
 constituie noul rezolvent
 $A_1, A_2, \dots, A_{i-1}, B_1, B_2, \dots, B_n, A_{i+1}, A_m$ format prin
 înlocuirea lui A_i în rezolventul curent, cu corpul
 instanței legate a clauzei care s-a ales, B_1, B_2, \dots, B_n .
 if rezolvent **vid**
 then output *da*
 else output *nu*.

Rezolventul este reprezentat de scopul curent, la fiecare moment. O trasare a interpretării este o secvență de rezolvenți rezultați în timpul calculelor (deducției) împreună cu alegerile făcute. Pentru întrebarea $G \equiv \leftarrow \text{father}(\text{ion}, \text{Son})$ și programul din exemplu, o trasare este reprezentată prin:

```

←father(ion, Son)
  father(X, Y) ← male(X), parent(X, Y) .
  θ={X=ion; Son=Y}

  ←male(ion), parent(ion, Y)
    male(ion)
    θ=∅

  ← parent(ion, Y)
    parent(ion, radu) .
    θ={Y=radu}

  ←∅

```

Reprezentarea marchează (prin subliniere) scopul care este ales pentru reducere din rezolventul curent. În exemplul nostru, ultimul rezolvent fiind vid, răspunsul este da (mai mult, în fapt se returnează și legarea $Son = radu$, ca rezultat al lanțului de instanțieri ($Son=Y$, $Y=radu$)).

Pe baza interpretorului abstract se poate construi arborele de deducție; acesta constă din noduri și arce. Rădăcina arborelui este scopul inițial G (în cazul în care este o întrebare simplă, nu una compusă), iar nodurile reprezintă scopurile care sunt alese spre reducere din cadrul rezolventului curent. Există un arc de la un nod la un altul conținând o întrebare derivată. Pentru exemplul anterior, arborele are forma:

```
father(ion, Son)
      |
male(ion) — parent(ion, Y)
```

E timpul să facem două remarci pe care în mod intenționat le-am evitat până în acest moment. Există două alegeri pe care interpretorul trebuie să le realizeze (marcate îngroșat în algoritmul interpretorului): alegerea scopului care urmează a fi redus din rezolvent, și a clauzei cu care acesta se va reduce. Deși de natură diferită, modalitatea în care aceste alegeri se realizează definesc semantica procedurală a limbajului logic.

În timp ce alegerea scopului care urmează a fi redus este arbitrară, alegerea clauzei cu care acesta se va reduce este critică. În orice rezolvent, în cele din urmă fiecare scop trebuie redus, și se poate demonstra (pentru demonstrație vezi [7]) că ordinea alegerii scopului este nerelevantă; adică, dacă există o demonstrație pentru un scop, atunci aceasta va fi disponibilă, indiferent de modalitatea de alegere a scopului. Această alegere definește regula de calcul, iar în limbajele logice secvențiale regula este de alegere a scopului celui mai din stânga.

Pentru alegerea clauzei (dar mai ales a instanței sale), în timp ce alegerea propriu-zisă reprezintă o problemă, alegerea instanței este o problemă cu infinit mai multe variante. Alegerea clauzei definește regula de selecție, și este una nedeterministă. O alegere nedeterministă reprezintă alegerea "aleatorie" a unei alternative dintr-un număr de alternative, astfel încât, dacă doar unele dintre alternative conduc la o secvență de calcul (demonstrație) corectă, atunci se alege doar dintre acestea. Deși aparent imposibil de implementat prin intermediul unei mașini o astfel de definiție, limbajele logice o realizează, și

capitolul următor prezintă pe scurt acest mecanism. Alegerea clauzei definește regula de selecție, și în programarea logică secvențială regula este: alege prima clauză prima dată (este importantă specificarea prima dată, pentru că în fapt, prin mecanismul de backtracking, care în limbajele logice secvențiale este implementat, următoarele clauze sunt selectate, dacă mai este nevoie; detalii legate de acest aspect sunt prezentate în capitolul dedicat backtrackingului).

1.4 Unificarea

Nucleul modelului computațional al programelor logice îl constituie mecanismul de unificare. Unificarea este de asemenea baza deducțiilor automate și a inferențelor logice din inteligența artificială.

Definim și/sau redefinim câteva noțiuni necesare prezentării mecanismului unificării.

Un termen t este o *instanță comună* a doi termeni t_1 și t_2 , dacă există substituțiile θ_1 și θ_2 astfel încât $t = t_1\theta_1 = t_2\theta_2$. Un termen t este *mai general* decât un termen u dacă u este o instanță a lui t , dar t nu este o instanță a lui u . Un termen t este un *variant alfabetic* al lui u dacă atât u este o instanță a lui t , cât și t este o instanță a lui u .

De exemplu $p(f(1,2),3)$ este o instanță comună a lui $p(A,B)$ și $p(f(X,Y),Z)$, termenul $p(A,B)$ este mai general decât $p(f(X,Y),Z)$, în timp ce $p(A,B)$ este un variant alfabetic al lui $p(U,V)$.

Un *unificator* a doi termeni este o substituție care face termenii identici. Spunem că doi termeni se pot unifica dacă au un unificator. Există o legătură strânsă între instanțe comune și unificatori, instanța comună determinând unificatorul, și reciproc, unificatorul determinând instanța comună.

Pentru instanțele $p(A,B)$ și $p(f(X,Y),Z)$ cu instanța comună $p(f(1,2),3)$, unificatorii sunt: $\theta_1 = \{A = f(1,2), B = 3\}$ și $\theta_2 = \{X = 1, Y = 2, Z = 3\}$.

Cel *mai general unificator* (notat **mgu**, de la most general unifier) a doi termeni este unificatorul cu proprietatea că instanța comună asociată este cea mai generală. Pentru doi termeni care se pot unifica, mgu este unic (exceptând redenumirea variabilelor), și reciproc, doi termeni au un unic mgu.

Pentru $p(A, B)$ și $p(f(X, Y), Z)$, $\text{mgu} = \{A = f(X, Y), B = Z\}$.

O *substituție* θ e o mulțime finită de forma $\{v_1 / t_1, \dots, v_n / t_n\}$, cu toate variabilele distincte între ele și față de termenii de substituție. $E\theta$ este o *instanță a lui E* prin θ , obținută prin aplicarea substituției θ expresiei E . *Compoziția* a două substituții $\theta = \{u_i / s_i \mid i=1, m\}$ și $\sigma = \{v_j / t_j \mid j=1, n\}$ este substituția $\theta\sigma = \{u_1 / s_1\sigma, \dots, u_m / s_m\sigma, v_1 / t_1, \dots, v_n / t_n\}$ obținută prin eliminarea oricărei legări $u_i / s_i\sigma$ în care $u_i = s_i\sigma$ și a celor de forma v_j / t_j pentru care $v_j = u_i$, pentru o valoare $i=1, m$. Se numește *substituție identică* $\varepsilon = \{ \}$ substituția dată de mulțimea vidă.

De exemplu, fie $E = p(x, y, f(a))$ și substituția $\theta = \{x / b, y / z\}$. Atunci $E\theta = p(b, z, f(a))$.

Fie $\theta = \{x/f(y), y/z\}$ și $\sigma = \{x/a, y/b, z/y\}$. Atunci $\theta\sigma = \{x/f(b), z/y\}$.

Dacă θ, σ și γ sunt substituții și E o expresie oarecare. Avem:

- a) $\theta\varepsilon = \varepsilon\theta = \theta$.
- b) $(E\theta)\sigma = E(\theta\sigma)$.
- c) $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

De exemplu, fie $\theta = \{x / f(y), y / z\}$ și $\sigma = \{x / a, z / b\}$. Atunci $\theta\sigma = \{x / f(y), y / b, z / b\}$. Fie $E = p(x, y, g(z))$. Atunci $E\theta = p(f(y), z, g(z))$ și $(E\theta)\sigma = p(f(y), b, g(b))$. De asemenea $E(\theta\sigma) = p(f(y), b, g(b))$.

Pentru a defini algoritmul de unificare mai avem nevoie de un element:

Se numește *mulțime de dezacord* a unei mulțimi finite S de termeni, mulțimea obținută prin aplicarea repetată a următorului procedeu: se determină poziția celui mai din stânga simbol în care termenii din S nu sunt identici și se extrag din fiecare termen din S subexpresiile corespunzătoare. Mulțimea tuturor acestor subexpresii este mulțimea de dezacord.

Algoritm de unificare

- pas1: $k=0$ și $\sigma_0 = \varepsilon$
- pas2: dacă $S\sigma_k$ este singleton, algoritmul se termină cu $\text{mgu}(S) = \sigma_k$; altfel, se determină mulțimea D_k de dezacord al lui $S\sigma_k$.
- pas3: dacă există în D_k v și t a.î. v nu apare (occur check) în t , atunci $\sigma_{k+1} = \sigma_k \{v/t\}$, se incrementează k și se revine la pasul 2; altfel, algoritmul se termină cu eșec (S neunifiabil).

Observație Acest algoritm reprezintă nucleul algoritmului de unificare în sistemele de programare logică.

În continuare prezentăm două exemple; în primul exemplu unificarea eșuează, iar în al doilea unificarea este cu succes.

Fie $S = \{p(f(a)), g(x), p(y, y)\}$.

- i) $\sigma_0 = \varepsilon$.
- ii) $D_0 = \{f(a), y\}$, $\sigma_1 = \{y / f(a)\}$ și $S \sigma_1 = \{p(f(a), g(x)), p(f(a), f(a))\}$
- iii) $D_1 = \{g(x), f(a)\}$ și deci S nu este unifiabil.

Dacă se aplică algoritmul de unificare pentru mulțimea:

$S = \{p(a, x, h(g(z))), p(z, h(y), h(y))\}$.

- i) $\sigma_0 = \varepsilon$.
- ii) $D_0 = \{a, z\}$, $\sigma_1 = \{z / a\}$ și $S \sigma_1 = \{p(a, x, h(g(a))), p(a, h(y), h(y))\}$.
- iii) $D_1 = \{x, h(y)\}$, $\sigma_2 = \{z / a, x / h(y)\}$ și $S \sigma_2 = \{p(a, h(y), h(g(a))), p(a, h(y), h(y))\}$.
- iv) $D_2 = \{y, g(a)\}$, $\sigma_3 = \{z / a, x / h(g(a)), y / g(a)\}$ și $S \sigma_3 = \{p(a, h(g(a), h(g(a))))\}$, care fiind un singleton, înseamnă că σ_3 este un mgu.

Un alt exemplu: în pasul 3 al algoritmului se face o verificare de apariție ("*occur check*") a lui v în t . Această verificare este exemplificată prin prezentul exemplu.

Fie $S = \{p(x, x), p(y, f(y))\}$.

- i) $\sigma_0 = \varepsilon$.
- ii) $D_0 = \{x, y\}$, $\sigma_1 = \{x / y\}$ și $S \sigma_1 = \{p(y, y), p(y, f(y))\}$.
- iii) $D_1 = \{y, f(y)\}$ și deoarece y apare în $f(y)$ S nu este unifiabil.

Teoremă (de unificare) Algoritmul de unificare se termină. Dacă mulțimea S e unifiabilă, algoritmul furnizează mgu(S), dacă nu, se termină cu eșec.

În lumina algoritmului de unificare, care definește metodologia de calcul a mgu, algoritmul interpretorului abstract poate fi rafinat:

Algoritm: **inițializează** rezolventul la întrebarea inițială G
 while rezolventul A_1, A_2, \dots, A_m e **nevid**
 alege un scop $A_i, i=1,m$;
 alege o instanță a unei clauze $A \leftarrow B_1, B_2, \dots, B_n, .$
 din P cu proprietatea că A_i și A sunt unificabili, cu
 $mgu=\theta$. Dacă nu există o astfel de clauză, **ieși** din
 buclo **while**;
 constituie noul rezolvent:
 $(A_1, A_2, \dots, A_{i-1}, B_1, B_2, \dots, B_n, A_{i+1}, A_m) \theta$
 if rezolvent **vid**
 then output *da*
 else output *nu*.

În arborii de deducție prezentați în capitolul următor, la nivelul unui nod se află scopul returnat (ales) de regula de calcul (cel mai din stânga), care se unifică cu capul clauzei returnată (aleasă) de regula de selecție (prima întâi, în ordinea de apariție în P). Aceste două reguli au ca și consecință strategia de căutare în spațiul soluțiilor la nivelul limbajelor logice secvențiale, și anume strategia de căutare în adâncime.

1.5 Programarea în Prolog pur

Se numește program pur Prolog un program logic în care se definește ordinea clauzelor predicatelor și ordinea scopurilor în corpul clauzelor. Interpretorul abstract este specializat să obțină toate avantajele posibile oferite de această ordonare. În continuare se prezintă mecanismul execuției programelor Prolog, care, sperăm noi, vor conduce la o mai bună și ușoară înțelegere atât a limbajului în sine, dar și a subtilităților care îi conferă puterea.

Programele logice care se execută cu mecanismul de execuție Prolog sunt referite ca programe Prolog pure. Prologul pur reprezintă modalitatea de implementare pe o mașină secvențială a modelului de calcul al programării logice într-o manieră cât mai apropiată de model. Nu este singura manieră, dar reprezintă alternativa cea mai bună din punct de vedere aplicativ, care păstrează echilibrul între rigurozitatea și exactitatea modelului abstract și necesitățile impuse de realizarea unei implementări eficiente. Vom vedea cum, din rațiuni de eficiență (dar și conferirea unor noi valențe din punct de

vedere a expresivității) anumite mecanisme părăsesc sfera Prologului pur (ne referim aici la negație, tăierea de backtracking și efecte laterale).

În conformitate cu algoritmul interpretorului abstract prezentat în capitolul precedent, o materializare a modelului logic (implementarea fizică a interpretorului) necesită nuanțarea acestuia, prin specificarea concretă (una, din multiplele, infinit număr de posibilități) a alegerilor pe care algoritmul le menționează. Este evident că acestea vor trebui să aibă în prim-plan rațiuni de eficiență, păstrând în permanență privirea ațintită asupra corectitudinii (parțiale, dar mai ales totale), urmărind modelul specificat. În acest fel, orice implementare concretă trebuie să menționeze care scop al rezolventului se alege mai întâi, cu alte cuvinte, trebuie precizată o politică de planificare. Pe de altă parte, trebuie precizată și implementată modalitatea de alegere a clauzei cu care scopul ales se unifică.

Din multitudinea de varietăți posibile, în practică există două direcții majore de abordare. Strategia secvențială, abordare prezentă la Prolog și extensiile sale (limbajele de programare logică secvențială) și respectiv strategia paralelă (însușită de limbaje precum PARLOG, Prolog Concurrent, GHC și altele).

Mecanismul de execuție Prolog se obține prin specificarea în interpretorul abstract a scopului din rezolventul ales mai întâi ca fiind cel mai din stânga (leftmost), în loc de unul arbitrar, în timp ce alegerea nedeterministă a clauzei pentru unificare cu scopul ales se înlocuiește cu căutarea secvențială și backtracking a clauzei cu care scopul e unificabil în ordinea apariției în program.

Cu mențiunile făcute, interpretorul abstract se materializează în următorul mecanism de execuție:

Mecanism: inițializează rezolventul la întrebarea inițială $G \equiv A_1, A_2, \dots, A_m$

```

while rezolventul  $A_1, A_2, \dots, A_m$  e nevid
    alege scopul  $A_1$ 
    //  $A_i$  înlocuiește  $A_i, i=1, m;$ 
    alege prima clauză, cu revenire
    // prima clauză, cu revenire înlocuiește o clauză
     $A \leftarrow B_1, B_2, \dots, B_n, \dots$  din P cu proprietatea
  
```

că A_1 și A sunt unificabili, cu $mgu=\theta$. Dacă nu există o astfel de clauză, **ieși** din bucla while;
constituie noul rezolvent:
 $(B_1, B_2, \dots, B_n, A_2, \dots, A_m)\theta$
 $// (B_1, B_2, \dots, B_n, A_2, \dots, A_m)\theta$ înlocuiește
 $// (A_1, A_2, \dots, A_{i-1}, B_1, B_2, \dots, B_n, A_{i+1}, A_m)\theta$
if rezolvent **vid**
 then output *da*
 else output *nu*.

Analizând particularizările, se observă că Prologul adoptă o politică de planificare de tip stivă, menținând rezolventul pe stivă, prelund din vârful ei (pop A_1), și actualizând cu scopul derivat B_1, B_2, \dots, B_n). În plus, căutarea nedeterministă (alegerea nedeterministă a clauzei din regula de calcul) este simulată prin reducerea la prima clauză cu care unificarea scopului ales este posibilă și backtracking. Dacă pentru scopul curent nu se găsește nici o alternativă (unificarea acestui scop cu fiecare dintre capul clauzelor predicatului corespondent eșuează) procesul de calcul revine la nodul precedent (punctul în care s-a produs alegerea scopului anterior, precedentă operație pop) și următoarea clauză unificabilă este aleasă (din lista de clauze program a recentului scop selectat, începând de la cea cu care unificarea s-a produs la pasul precedent).

Calculul unui scop G în contextul unui program logic P presupune generarea tuturor soluțiilor lui G , și deci se realizează o *traversare în adâncime completă* a arborelui de căutare a lui G obținut prin selectarea de fiecare dată a scopului aflat cel mai în stânga.

Există numeroase implementări Prolog, cu diferențe mai mici sau mai mari de sintaxă, procedurale și înlesniri. Majoritatea urmează convențiile Prolog Edinburg. Toate exemplele prezentate în această lucrare sunt rulate în SICStus Prolog 3.12.5.

O trasare a unui calcul Prolog este o extensie a unei trasări a unui program logic, și în continuare sunt prezentate câteva exemple de trasare la execuție: varianta cu prezentarea arborilor de deducție derivați (trasarea programului logic, care se poate realiza independent de existența fizică a unui interpretor concret, care implementează prin mecanisme concrete interpretorul abstract), dar și trasarea urmărită la execuția propriu-zisă.

Backtracking

În precizarea metodei concrete de alegere a clauzei care urmează să se unifice cu scopul selectat (la trecerea de la interpretorul abstract la mecanismul Prolog) se menționează alegerea primei clauze prima dată, urmată de backtracking. Modalitatea concretă de realizare a backtrackingului (prezent în mecanismul Prolog, implementat la nivelul oricărui sistem de limbaj de programare logic secvențial), împreună cu precizarea modului concret de alegere din algoritmul interpretorului abstract, formează mecanismul Prolog, care asigură căutarea întregului spațiu de căutare în manieră depth-first-search (căutare în adâncime) și găsirea tuturor soluțiilor la o întrebare.

Mecanismul de backtracking, implementarea căruia asigură determinarea soluțiilor alternative, își face simțită prezența în două situații distincte, și pentru că acțiunea în fiecare dintre ele (deși principial aceeași) este totuși particulară, s-au introdus noțiuni diferite care denumesc acțiunile în fiecare din cele două cazuri: *shallow* și *deep backtracking* (backtracking de suprafață, superficial, și respectiv de adâncime).

Pornind de la modalitatea de alegere a clauzei pentru unificare, regula de selecție specifică Prolog alege prima clauză prima dată. Cu alte cuvinte, în momentul în care scopul care se reduce la pasul curent s-a precizat, se urmărește găsirea clauzei de unificat în felul următor: din predicatul cu aceeași semnătură (aceiași nume și același număr de argumente) se selectează mai întâi prima clauză pentru care se încearcă unificarea scopului curent (care se reduce) cu capul acestei clauze. Există două alternative pentru această unificare, succes sau eșec.

Dacă *unificarea reușește*, atunci se determină mgu, și se aplică pasul următor al algoritmului (constituie noul rezolvent: $(B_1, B_2, \dots, B_n, A_2, \dots, A_m)\theta$ și se reia bucla while).

Dacă *unificarea eșuează* (nu se poate determina mgu pentru A_1 și A , pentru că aceștia nu sunt unificabili), atunci se instalează *backtrackingul superficial*. Fără a se semnaliza vreo eroare sau eșec (care să solicite eventual utilizatorului să decidă dacă dorește continuarea căutării, și să o lanseze), implicit mecanismul Prolog trece la următoarea alternativă (dacă aceasta există), adică la a doua clauză a predicatului cu aceeași semnătură, și lucrurile se repetă (adică unificarea reușește, sau se instalează backtrackingul superficial). Dacă, în ciuda încercării tuturor alternativelor

posibile (prin backtracking superficial, toate clauzele au fost încercate pentru reducerea scopului curent), unificarea este fără succes (eșuează unificarea scopului cu ultima clauză a predicatului corespondent), se instalează *backtrackingul adânc*.

Prin backtracking adânc, apărut când unificarea scopului curent cu ultima clauză eșuează, controlul revine unui alt scop din arborele de execuție. Concret, se revine un pas înapoi în bucla while, la cel mai din stânga scop al rezolventului precedent (adică, dacă la pasul curent am avut $B_1, B_2, \dots, B_n, A_2, \dots, A_m$, care provenea din reducerea A_1, A_2, \dots, A_m și unificarea A_1 cu capul clauzei $A \leftarrow B_1, B_2, \dots, B_n$, iar B_1 eșuează în a se unifica cu capul vreunei clauze, atunci se revine la unificarea A_1 cu o altă clauză; concret, se alege clauza următoare, $A \leftarrow C_1, C_2, \dots, C_n$, și deci rezolventul se reduce la $C_1, C_2, \dots, C_n, A_2, \dots, A_m$, în loc de $B_1, B_2, \dots, B_n, A_2, \dots, A_m$). Procesul prin care, eșecul unificării B_1 cu oricare din clauzele aceluiasi predicat produce revenirea la A_1 , cerându-i acestuia să satisfacă (în mod diferit) scopul ales, se numește *backtracking de adâncime*.

1.6 Comparația cu limbajele de programare imperative

În programele Prolog *controlul* este similar cu cel din limbajele procedurale atât timp cât calculele avansează la secvența următoare. Apelul unui nou scop (întrebări Prolog) corespunde invocării unei proceduri, în timp ce ordonarea scopurilor în corpul unei clauze corespunde secvențierii instrucțiunilor într-o funcție. Concret, clauza $A \leftarrow B_1, B_2, \dots, B_n$ poate fi văzută în chip procedural sub forma:

```
procedure A
    call B1
    call B2
    .
    .
    .
    call Bn
end.
```

Apelul recursiv în Prolog al unui scop este absolut simetric atât din punct de vedere al comportamentului, cât și a implementării apelului în limbaje procedurale. Unica diferență apare la backtracking, și aceasta este în

favoarea Prologului (care are acest mecanism implementat). La limbajele procedurale, dacă într-un anumit punct execuția nu poate progresa este semnalată o eroare la execuție (de exemplu când toate ramurile unui *case* sunt false, sau ar trebui avansat pe o ramură care nu are cod specific). În Prolog, o astfel de situație este tratată prin reluarea execuției din punctul anterior, adică refacerea stării din nodul precedent al arborelui de execuție, și satisfacerea întrebării corespunzătoare acelui nod cu clauza următoare celeia cu care s-a realizat anterior unificarea. În felul acesta, o altă ramură din spațiul de căutare este abordată, fără a se semnala nici un fel de eroare, și fără intervenția programatorului.

Structurilor de date din limbajele imperative le corespund *termenii* Prolog, limbajul fiind extrem de flexibil din acest punct de vedere; utilizarea unor tipuri de date utilizator este liberă (fără să fie nevoie de vreo declarație), limbajul fiind netipizat. Acest avantaj major, care conferă o libertate și flexibilitate extremă programatorului, are și neajunsul de a fi generator de erori (în lipsa tipizării se presupune că orice este permis, o eroare de tipărire neputând fi identificată, și deci nefiind semnalată; se interpretează simplu ca un nou tip de dată, un nou functor).

Noutatea absolută introdusă de limbaj este însă *variabila logică*. Aceasta se referă la instanțe, și nu la locație de memorie, așa cum este în limbajele imperative (unde fiecărei variabile îi corespunde o locație de memorie, locul unde valoarea variabile se păstrează). În mod absolut simetric, o anumită instanță (reprezentată de o variabilă) nu poate fi "făcută" să refere o altă instanță (o variabilă odată instanțiată, rămâne instanțiată la valoare respectivă până când prin backtracking, datorită eșecului provocat la nivelul nodului la care instanțierea s-a produs, instanțierea se anulează, variabila devenind "liberă" să se instanțieze la o altă valoare/termen Prolog).

În fine *unificarea* Prolog subsumează operațiile de atribuire, pasarea argumentelor, alocarea înregistrărilor și accesul în citire/scriere din limbajele imperative.

1.7 Tăierea de backtracking și negația

1.7.1 Tăierea de backtracking (!)

Prologul furnizează un predicat sistem, *cut* (tăierea de backtracking) care afectează semantica procedurală a programelor, prin utilizarea sa ieșindu-se înafara spațiului programării logice pure (fiind un predicat extra-logic). Funcția sa de bază este reducerea spațiului de căutare a soluțiilor prin eliminarea dinamică a unor ramuri ale arborelui de căutare. Tăierea de backtracking poate fi utilizată pentru a preveni interpretorul Prolog de a urma ramuri pe care programatorul știe că nu există soluții (*green cut*), îmbunătățind în acest fel performanța (timp, printr-o execuție mai rapidă, spațiu, prin evitarea stocării informațiilor care s-ar genera pe ramurile care nu se mai parcurg). Utilizarea acestei forme nu afectează înțelesul programului, semantica declarativă fiind nealterată. Pe de altă parte, tăierea de backtracking poate fi utilizată chiar și pentru a împiedica căutarea pe ramuri care conțin soluții (*red cut*), atunci când nu interesează soluțiile alternative ale unei probleme. Utilizarea în acest scop este influențată de forma slabă a negației.

Tocmai pentru că afectează semantica procedurală, utilizarea tăierii de backtracking este discutabilă; majoritatea utilizărilor sale pot fi interpretate doar procedural, spre deosebire de maniera firească, declarativă de interpretare a programelor Prolog. Este de remarcat că utilizată cu prudență și în cunoștință de cauză (asupra efectelor sale, asupra ceea ce afectează dar și ce nu afectează) tăierea de backtracking poate fi benefică, îmbunătățind performanța, fără afectarea clarității.

Tăierea de backtracking, indicată prin **!**, poate fi utilizată pentru exprimarea naturii mutual exclusive a testelor, fiind plasată în această situație după teste aritmetice. Din punctul de vedere al semanticii operaționale, înțelesul unei tăieri de backtracking (mai precis a unei clauze conținând o tăiere de backtracking) este următorul: *Scopul curent (întrebarea selectată din cadrul rezolventului pentru a fi redusă) reușește iar Prolog comite (se oprește determinist, fără drept de revenire) toate alegerile făcute din momentul în care scopul părinte s-a unificat cu capul clauzei în care tăierea de backtracking apare.*

Deși din punct de vedere operațional descrierea anterioară este completă pentru efectele pe care tăierea de backtracking le are asupra execuției, implicațiile pe care utilizarea sa le are sunt diverse, nu totdeauna intuitive, iar neînțelegerea completă a efectelor sale este o sursă majoră de erori. Aceste neînțelegeri se împart în două categorii complementare: presupunerea că o anumită tăiere de backtracking taie anumite ramuri pe care în realitate NU le taie (de cele mai multe ori acest tip de eroare conducând doar la execuții ineficiente, în situații mai rare apărând eroare), și invers, presupunerea că nu taie ramuri pe care în realitate le taie (mai periculoasă; în acest caz se pot pierde soluții). De aceea trebuie luată în considerare efectiv porțiunea de arbore care este afectată de o tăiere de backtracking. Un ! afectează:

- toate clauzele aflate sub clauza conținând !. Un scop p unificat cu o clauză conținând ! care reușește (clauza respectivă s-a executat cu succes) NU va produce soluții utilizând clauze ale predicatului aflate sub clauza conținând !;
- toate soluțiile alternative ale conjuncției de scopuri aflate la stânga ! (ceea ce înseamnă că un scop compus urmat de ! va produce CEL MULT o soluție);

Pentru un predicat p de forma:

$p_1 : \neg q_{11}, q_{12}, \dots, q_{1m}.$

.

$p_i : \neg q_{i1}, q_{i2}, \dots, q_{ik}, !, q_{ik+1}, \dots, q_{im}.$

.

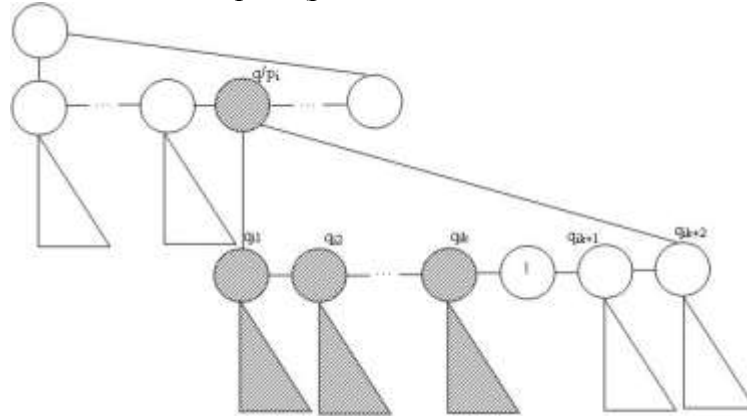
$p_n : \neg q_{n1}, q_{n2}, \dots, q_{nm}.$

Și un scop a cărui unificare cu a i -a clauză a lui p a reușit, următoarele porțiuni nu produc soluții alternative:

- niciuna din clauzele de la p_{i+1} la p_n nu va produce o soluție alternativă
- conjuncția $q_{i1}, q_{i2}, \dots, q_{ik}$, nu va produce soluții alternative.

Pe de altă parte, ! nu afectează scopurile aflate la dreapta sa, acestea putând produce oricâte soluții (toate soluțiile posibile). În exemplu, conjuncția q_{ik+1}, \dots, q_{im} poate produce soluții alternative (dacă există). Totuși, dacă la un moment dat această conjuncție eșuează, ! previne revenirea la q_{ik} ,

controlul întorcându-se la prima alternativă înaintea celei care a ales capul clauzei în care ! apare (precedenta alternativă celei în care s-a ales p_i).



În desenul anterior nodurile hașurate sunt cele care nu fac backtracking.

1.7.2 Negația

Tăierea de backtracking reprezintă nucleul implementării (unei forme limitate a) negației Prolog, *negația ca eşec*. Implementarea acesteia este:

```
not(P) :- P, !, fail.
not(P) .
```

Conform implementării, $\text{not}(\text{Goal})$ este adevărat dacă Goal este fals. Implementarea utilizează de asemenea predicatul sistem *fail* care eşuează (neexistând clauze definite pentru acesta). De remarcat conjuncția $!, \text{fail}$ care este referită în literatură ca și combinație *cut-fail* (intens folosită în tehnica efectelor laterale).

Să analizăm comportamentul secvenței de mai sus la apelul $\text{not}(P)$: prima clauză este încercată, care apelează scopul P (din corp) prin intermediul facilităților furnizate de meta-variabile (semantica variabilei P fiind cea de predicat Prolog, deci meta-variabilă). Dacă P reușește, se execută $!$, determinând comiterea execuției la prima clauză; astfel $\text{not}(P)$ eşuează. Dacă, pe de altă parte, execuția lui P eşuează, (cum nu s-a trecut peste $!$) se va trece la a doua regulă a predicatului not care reușește. Deci dacă P eşuează, $\text{not}(P)$ reușește. Utilizarea $!$ în negație este un *red cut*, deoarece eliminarea sa conduce la schimbarea comportamentului programului.

2. Operații pe mulțimi

Limbajele logice permit efectuarea de raționamente inductive, exprimate prin definiții recursive de predicate. În fapt, recursivitatea este modul natural de exprimare a algoritmilor repetitivi în limbajele declarative. Pornim de la un exemplu banal, de testare a apartenenței unui element la o listă; elementul este în listă fie dacă este primul element al listei, fie, în cazul în care este membru în coada listei. Acest lucru se poate exprima în Prolog în felul următor:

```
%member/2
%member(element, listă)

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

În exemplul de mai sus este introdusă notația Prolog Edinburg pentru listă. Notația pentru listă vidă (fără elemente) este "[]". O listă de numere este "[1,2,7,5]" iar o listă de simboluri "[a,b,c]". O listă poate fi descompusă/recompusă în/din elementele constitutive utilizând notația cu șablon:

```
Lista=[PrimulElement|RestLista]
```

Deci "|" realizează descompunerea/recompunerea listei în/din elementele sale constitutive: primul element, și respectiv restul listei. Ceea ce înseamnă că prin aplicarea șablonului "|" peste o variabilă de tip listă, variabila din stânga se va instanția la primul element al listei, în timp ce variabila din dreapta la lista din care primul element a fost eliminat. Ceea ce înseamnă că semantica celor doi termeni constitutivi este diferită: element și respectiv listă!

De exemplu:

```
?-L=[a,b,c], L=[E|R] .
yes, L=[a,b,c], E=a, R=[b,c]
```

Se remarcă faptul că R este o listă și E este un element. Este admisă și notația mai generală în care sunt evidențiate un număr arbitrar dar fix de elemente din capul unei liste. De exemplu:

```
?-L=[1,2,3,4], L=[E1,E2,E3|R].
yes, L=[1,2,3,4], E1=1, E2=2, E3=3, R=[4]
```

Exemplele de mai sus au prezentat numai liste plate și omogene (cu elemente pe același nivel și de același tip). Prolog este un limbaj slab tipizat și admite și liste adânci și eterogene cum ar fi "[a,2,[3,c],d]". De menționat că nivelul de adâncime este arbitrar (pot exista oricâte nivele de adâncime).

În definiția predicatului "member" s-a utilizat notația "_", care referă o entitate (variabilă) **indiferentă**, a cărei valoare nu este necesară în calculele specificate de clauza în care apare. Ca regulă generală, orice variabilă care apare o singură dată într-o clauză oarecare, și a cărei valoare nu este necesară, poate fi înlocuită cu notația "_".

Un avantaj al slab-tipizării constă în aceea că aceeași definiție a unui predicat (în cazul nostru definiția de mai sus a predicatului member) poate fi utilizată pentru diferite tipuri de argumente.

```
?-member(1,[2,1,3]).
yes
?-member(a,[z,c,d,a,u]).
yes
```

Se observă că același predicat a identificat corect prezența unui întreg, dar și a unui simbol. Mai mult, aceeași definiție poate fi folosită pentru identificarea prezenței unor structuri complexe.

Prin intermediul câtorva operații pe mulțimi evidențiem potențialul listelor, expresivitatea exprimării predicative, și un mecanism nou, caracteristic programării logice, deși el se situează înafara programării logice pure: tăierea de backtracking.

2.1 Reuniunea

În continuare vom considera că mulțimile sunt reprezentate sub formă de liste (impunem deci restricția ca într-o listă toate elementele să fie distincte). Plecând de la definiția reuniunii, în mulțimea soluție se vor afla elementele comune și necomune, o singură dată. Această specificație foarte naturală, a cărei transcriere matematică este imediată, se bucură de o implementare

procedurală firească (vom vedea că citirea clauzelor predicatului urmează întocmai definiția; iar dacă privim invers, de la definiție/specificații se poate trece imediat la implementarea procedurală).

Să pornim de la definiția matematică:

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

Acum, chiar de la definiția matematică plecând, în practică, dacă dorim să identificăm (manual) reuniunea a două mulțimi, trebuie să avem o metodologie prin care să identificăm elementele cu proprietatea: $x \in A \vee x \in B$. Varianta firească este să se baleeze toate elementele unei mulțimi, cu inspectarea existenței elementului curent în cealaltă mulțime. Existența elementului în ambele mulțimi nu necesită adăugarea la momentul curent a elementului în soluție; adăugarea se va produce simultan pentru toate elementele celei de-a doua mulțimi, odată cu întreaga mulțime. Adică dacă x este elementul curent în A , atunci $x \in B$ indică neadăugarea la acest moment în soluție, ci amânarea până la sfârșitul execuției, odată cu toate elementele din B în timp ce dacă elementul $x \notin B$ trebuie adăugat acum în soluție (deoarece $x \in A$ și $x \notin B$ înseamnă că trebuie să fie în soluție, dar nu poate fi adăugat la sfârșitul execuției pentru că nu este în B). Matematic, acest lucru se exprimă:

$$\begin{aligned} A \cup B &= \{ \text{CurentA}, \text{RestA} \} \cup B \\ &= \begin{cases} \text{RestA} \cup B & \text{dacă } \text{CurentA} \in B \\ \text{RestA} \cup B \text{ și } \{ \text{CurentA} \} & \text{dacă } \text{CurentA} \notin B \end{cases} \end{aligned}$$

Pentru cazul în care se ajunge ca $A = \emptyset$, trebuiesc adăugate în soluție simultan toate elementele mulțimii B . Adică:

$$A \cup B = B, \text{ dacă } A = \emptyset$$

Dacă un predicat pentru reuniune este deja scris, atunci reuniunea pentru $\text{RestA} \cup B$ se determină imediat prin apel recursiv. De asemenea, verificările de tipul $\text{CurentA} \in B$ și $\text{CurentA} \notin B$ se realizează natural prin apeluri `member (CurentA, B)`, respectiv `not (member (CurentA, B))`, și în sfârșit, operația de "atașare" a unui element la o soluție parțial construită, specificată în definiția matematică prin: $\text{RestA} \cup B$ și $\{ \text{CurentA} \}$ se

realizează prin aplicarea șablonului de compunere a listelor, sub forma `[CurentA|RestSolutie]` . Cu aceste observații trecem la scrierea predicatului de reuniune:

```
%union/3
%union(multimea A, multimea B, multimea reuniune)

union([HA|TA],B,U):-
    member(HA,B),
    union(TA,B,U).
union([HA|TA],B,[HA|U]):-
    not(member(HA,B)),
    union(TA,B,U).
union([],L,L).
```

O întrebare de forma:

```
| ?- union([1,3,5,6],[2,3,4,5,7],R).
R = [1,6,2,3,4,5,7] ?
Yes
```

furnizează soluția fără a avea soluții alternative, adică:

```
| ?- union([1,3,5,6],[2,3,4,5,7],R).
R = [1,6,2,3,4,5,7] ? ;
no
```

Se observă că elementele din soluție nu sunt ordonate. Cum justificați?

Încercăm să aducem îmbunătățiri predicatului, în sensul că dorim să se execute mai rapid, evitând verificări inutile. De pildă, cunoscând regula de selecție (pentru o întrebare, dacă există mai multe clauze ale unui predicat cu același nume și același număr de argumente, se aleg clauzele de sus în jos, execuția realizându-se pentru cea care are un matching cu succes), putem afirma că în momentul în care se ajunge la execuție să fie testată clauza a doua, în mod negreșit prima a fost încercată și a eșuat. Analizând cele două clauze, observăm că singura deosebire între ele (în afară de pattern-ul la argumentul al treilea, dar care fiind de ieșire îl construim abia acum, deci nu realizează discriminarea clauzelor) o reprezintă testul `member(HA,B)` respectiv `not(member(HA,B))`, care este negatul primului. Dar cum, în momentul când se ajunge la clauza 2 în mod obligatoriu clauza 1 s-a testat și a eșuat, înseamnă că negreșit acest eșec a

provenit de la `member(HA,B)`, și deci `not(member(HA,B))` reușește. Deci nu mai avem nici un motiv să testăm `not(member(HA,B))` în clauza 2. Deci predicatul se poate rescrie:

```
union([HA|TA],B,U):-
    member(HA,B),
    union(TA,B,U).
union([HA|TA],B,[HA|U]):-
    union(TA,B,U).
union([],L,L).
```

O execuție a sa ne dezvăluie răspunsuri surprinzătoare:

```
| ?- union([1,3,5,6],[2,3,4,5,7],R).
R = [1,6,2,3,4,5,7] ? ;
R = [1,5,6,2,3,4,5,7] ? ;
R = [1,3,6,2,3,4,5,7] ? ;
R = [1,3,5,6,2,3,4,5,7] ? ;
no
```

De ce? Deși răspunsul ne nemulțumește, este cât se poate de corect. Prima soluție este cea corectă. Următoarele (evident incorecte, dar problema având soluție unică, ne-am fi așteptat ca predicatul pe care îl implementăm să ofere și el soluție unică) provin de la însuși mecanismul programării logice: backtracking-ul este "built-in", deci fie că avem sau nu nevoie de el, el există, și atunci când îi este solicitat ajutorul (și am făcut-o când am repetat întrebarea ;), acest mecanism intră în acțiune! Ce se întâmplă atunci: deși pentru obținerea soluției inițiale $R = [1, 6, 2, 3, 4, 5, 7]$ clauza 1 a testat că $5 \in B$ prin `member(HA,B)`, la reluare (backtracking), matching-ul realizat s-a anulat, și execuția a încercat suprapunerea aceleiași întrebări peste clauza următoare. Cum clauza 2 nu mai testează `not(member(HA,B))`, deși , se consideră că $5 \notin B$, și atunci șablonul îl adaugă încă o dată pe 5 în soluție (el este adăugat o dată cu toate elementele lui B), astfel încât în soluția a 2a, apare (incorect) de 2 ori $R = [1, \underline{5}, 6, 2, 3, 4, 5, 7]$ (este subliniată adăugarea incorectă). Pe trasarea de mai jos se pot urmări pașii execuției pentru primele 3 soluții. Am îngroșat rezultatele, și liniile de apel responsabile de răspunsurile eronate.

```
| ?- union([1,3,5,6],[2,3,4,5,7],R).
+      1      1 Call: union([1,3,5,6],[2,3,4,5,7],_591) ?
+      2      2 Call: member(1,[2,3,4,5,7]) ?
+      3      3 Call: member(1,[3,4,5,7]) ?
```

```

4      4 Call: member(1, [4,5,7]) ?
5      5 Call: member(1, [5,7]) ?
6      6 Call: member(1, [7]) ?
7      7 Call: member(1, []) ?
7      7 Fail: member(1, []) ?
6      6 Fail: member(1, [7]) ?
5      5 Fail: member(1, [5,7]) ?
4      4 Fail: member(1, [4,5,7]) ?
3      3 Fail: member(1, [3,4,5,7]) ?
2      2 Fail: member(1, [2,3,4,5,7]) ?
+      8      2 Call: union([3,5,6], [2,3,4,5,7], _1319) ?
9      3 Call: member(3, [2,3,4,5,7]) ?
10     4 Call: member(3, [3,4,5,7]) ?
?      10     4 Exit: member(3, [3,4,5,7]) ?
?      9      3 Exit: member(3, [2,3,4,5,7]) ?
+      11     3 Call: union([5,6], [2,3,4,5,7], _1319) ?
12     4 Call: member(5, [2,3,4,5,7]) ?
13     5 Call: member(5, [3,4,5,7]) ?
14     6 Call: member(5, [4,5,7]) ?
15     7 Call: member(5, [5,7]) ?
?      15     7 Exit: member(5, [5,7]) ?
?      14     6 Exit: member(5, [4,5,7]) ?
?      13     5 Exit: member(5, [3,4,5,7]) ?
?      12     4 Exit: member(5, [2,3,4,5,7]) ?
+      16     4 Call: union([6], [2,3,4,5,7], _1319) ?
17     5 Call: member(6, [2,3,4,5,7]) ?
18     6 Call: member(6, [3,4,5,7]) ?
19     7 Call: member(6, [4,5,7]) ?
20     8 Call: member(6, [5,7]) ?
21     9 Call: member(6, [7]) ?
22     10 Call: member(6, []) ?
22     10 Fail: member(6, []) ?
21     9 Fail: member(6, [7]) ?
20     8 Fail: member(6, [5,7]) ?
19     7 Fail: member(6, [4,5,7]) ?
18     6 Fail: member(6, [3,4,5,7]) ?
17     5 Fail: member(6, [2,3,4,5,7]) ?
+      23     5 Call: union([], [2,3,4,5,7], _11015) ?
+      23     5 Exit: union([], [2,3,4,5,7], [2,3,4,5,7]) ?
+      16     4 Exit: union([6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
?+     11     3 Exit: union([5,6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
?+     8      2 Exit: union([3,5,6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
?+     1      1 Exit: union([1,3,5,6], [2,3,4,5,7], [1,6,2,3,4,5,7]) ?
R = [1,6,2,3,4,5,7] ? ;
+      1      1 Redo: union([1,3,5,6], [2,3,4,5,7], [1,6,2,3,4,5,7]) ?
+      8      2 Redo: union([3,5,6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
+      11     3 Redo: union([5,6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
12     4 Redo: member(5, [2,3,4,5,7]) ?
13     5 Redo: member(5, [3,4,5,7]) ?
14     6 Redo: member(5, [4,5,7]) ?
15     7 Redo: member(5, [5,7]) ?
24     8 Call: member(5, [7]) ?
25     9 Call: member(5, []) ?
25     9 Fail: member(5, []) ?
24     8 Fail: member(5, [7]) ?

```

```

15      7 Fail: member(5,[5,7]) ?
13      5 Fail: member(5,[3,4,5,7]) ?
14      6 Fail: member(5,[4,5,7]) ?
12      4 Fail: member(5,[2,3,4,5,7]) ?
+      26 4 Call: union([6],[2,3,4,5,7],_527)
12      4 Fail: member(5,[2,3,4,5,7]) ?
+      26 4 Call: union([6],[2,3,4,5,7],_5273) ?
27      5 Call: member(6,[2,3,4,5,7]) ?
28      6 Call: member(6,[3,4,5,7]) ?
29      7 Call: member(6,[4,5,7]) ?
30      8 Call: member(6,[5,7]) ?
31      9 Call: member(6,[7]) ?
32     10 Call: member(6,[ ]) ?
32     10 Fail: member(6,[ ]) ?
31      9 Fail: member(6,[7]) ?
30      8 Fail: member(6,[5,7]) ?
29      7 Fail: member(6,[4,5,7]) ?
28      6 Fail: member(6,[3,4,5,7]) ?
27      5 Fail: member(6,[2,3,4,5,7]) ?
+      33 5 Call: union([], [2,3,4,5,7], _5975) ?
+      33 5 Exit: union([], [2,3,4,5,7], [2,3,4,5,7]) ?
+      26 4 Exit: union([6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
+      11 3 Exit: union([5,6], [2,3,4,5,7], [5,6,2,3,4,5,7]) ?
?+      8 2 Exit: union([3,5,6], [2,3,4,5,7], [5,6,2,3,4,5,7]) ?
?+      1 1 Exit: union([1,3,5,6], [2,3,4,5,7], [1,5,6,2,3,4,5,7]) ?
R = [1,5,6,2,3,4,5,7] ? ;
+      1 1 Redo: union([1,3,5,6], [2,3,4,5,7], [1,5,6,2,3,4,5,7]) ?
+      8 2 Redo: union([3,5,6], [2,3,4,5,7], [5,6,2,3,4,5,7]) ?
9      3 Redo: member(3,[2,3,4,5,7]) ?
10     4 Redo: member(3,[3,4,5,7]) ?
34     5 Call: member(3,[4,5,7]) ?
35     6 Call: member(3,[5,7]) ?
36     7 Call: member(3,[7]) ?
37     8 Call: member(3,[ ]) ?
37     8 Fail: member(3,[ ]) ?
36     7 Fail: member(3,[7]) ?
35     6 Fail: member(3,[5,7]) ?
34     5 Fail: member(3,[4,5,7]) ?
10     4 Fail: member(3,[3,4,5,7]) ?
9      3 Fail: member(3,[2,3,4,5,7]) ?
+      38 3 Call: union([5,6], [2,3,4,5,7], _2021) ?
39     4 Call: member(5,[2,3,4,5,7]) ?
40     5 Call: member(5,[3,4,5,7]) ?
41     6 Call: member(5,[4,5,7]) ?
42     7 Call: member(5,[5,7]) ?
?      42 7 Exit: member(5,[5,7]) ?
?      41 6 Exit: member(5,[4,5,7]) ?
?      40 5 Exit: member(5,[3,4,5,7]) ?
?      39 4 Exit: member(5,[2,3,4,5,7]) ?
+      43 4 Call: union([6], [2,3,4,5,7], _2021) ?
44     5 Call: member(6,[2,3,4,5,7]) ?
45     6 Call: member(6,[3,4,5,7]) ?
46     7 Call: member(6,[4,5,7]) ?
47     8 Call: member(6,[5,7]) ?

```

```

48      9 Call: member(6, [7]) ?
49     10 Call: member(6, []) ?
49     10 Fail: member(6, []) ?
48      9 Fail: member(6, [7]) ?
47      8 Fail: member(6, [5,7]) ?
46      7 Fail: member(6, [4,5,7]) ?
45      6 Fail: member(6, [3,4,5,7]) ?
44      5 Fail: member(6, [2,3,4,5,7]) ?
+     50      5 Call: union([], [2,3,4,5,7], _8465) ?
+     50      5 Exit: union([], [2,3,4,5,7], [2,3,4,5,7]) ?
+     43      4 Exit: union([6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
?+    38      3 Exit: union([5,6], [2,3,4,5,7], [6,2,3,4,5,7]) ?
?+     8      2 Exit: union([3,5,6], [2,3,4,5,7], [3,6,2,3,4,5,7]) ?
?+     1      1 Exit: union([1,3,5,6], [2,3,4,5,7], [1,3,6,2,3,4,5,7]) ?
R = [1, 3, 6, 2, 3, 4, 5, 7] ?

```

Este evident deci că testul `not (member (HA, B))` este necesar.

Totuși, o variantă pentru evitarea dublei testări (chiar dacă la o execuție anterioară, testul a fost efectuat; nu "ține minte" sistemul rezultatul acestui test, ca să nu-l repetăm?): folosirea tăierii de backtracking.

O modalitate de a transforma un predicat nedeterminist într-unul determinist este tăierea backtracking-ului. Să ne reamintim că atunci când un scop q se unifică cu o clauza de forma:

$$q() :- b_1(), b_2(), \dots, b_{n-1}(), !, b_n(), \dots$$

operatorul de tăierea backtracking-ului (`cut`, `!`) este efectiv doar dacă unificarea scopului cu capul regulii s-a făcut cu succes și de asemenea toate subscopurile din corpul regulii aflate la stânga operatorului s-au executat cu succes (adică $b_1(), b_2(), \dots, b_{n-1}()$). În acest caz, instanțierile făcute nu mai pot fi anulate prin backtracking decât ca un tot unitar. Subscopurile aflate la dreapta operatorului nu sunt afectate: ele fac backtracking în mod obișnuit. Când nici unul din aceste subscopuri nu mai furnizează soluții, se revine (în lanțul de scopuri) la subscopul anterior lui q .

Deci tăierea (`!`) afectează:

- subscopurile din corpul clauzei aflate la stânga operatorului (o conjuncție de scopuri urmată de tăiere va produce cel mult o soluție)
- toate clauzele aflate după clauza curentă (un scop ce se unifică cu o clauză conținând `!`, nu va mai putea furniza soluții utilizând clauzele următoare)

și nu afectează:

- subscopurile din corpul clauzei aflate la dreapta operatorului
- clauzele aflate înaintea celei curente.

În arborele de execuție, odată ce nodul corespunzător tăierii de backtracking a fost generat, nici unul din nodurile aflate pe același nivel la stânga (împreună cu întregi subarborii lor), și nici nodul părinte nu mai fac backtracking. Primul nod care mai poate face backtracking este nodul aflat imediat la stânga nodului părinte și pe același nivel cu el.

În cazul specificat, tăierea de backtracking afectează scopul q și fiecare din scopurile $b_1()$, $b_2()$, ..., $b_{n-1}()$, și nu afectează nimic altceva.

Dacă unificarea lui q cu clauza curentă eșuează din cauza unui matching nereușit cu capul clauzei sau din cauza unui eșec la unul din subscopurile din corp aflate la stânga operatorului $!$, acesta devine inefectiv.

Utilizând operatorul de tăierea backtracking-ului pentru modificarea predicatului de reuniune obținem:

```
union([HA|TA], B, U) :-
    member(HA, B), !,
    union(TA, B, U).
union([HA|TA], B, [HA|U]) :-
    union(TA, B, U).
union([ ], L, L).
```

Acum la execuția

```
| ?- union([1,3,5,6], [2,3,4,5,7], R).
R = [1,6,2,3,4,5,7] ? ;
no
```

comportamentul este determinist, cel așteptat.

2.2 Intersecția

Acum un raționament similar se poate realiza pentru implementarea altor operații pe mulțimi. Să începem cu intersecția. Prin definiția intersecției, în mulțimea soluție se vor afla elementele comune ambelor mulțimi, o singură dată. Această specificație beneficiază de o transcriere matematică imediată, dar și de o implementare procedurală firească..

Să pornim de la definiția matematică:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

Plecând de la definiția matematică dacă dorim să identificăm intersecția a două mulțimi, metodologia identifică elementele cu proprietatea: $x \in A \wedge x \in B$. Din nou varianta naturală este să se baleeze toate elementele unei mulțimi, cu inspectarea existenței elementului curent în cealaltă mulțime. Existența elementului în ambele mulțimi necesită adăugarea la soluție, în timp ce apartenența doar într-una din mulțimi doar avansul în mulțime. Adică dacă x este elementul curent în A , atunci $x \in B$ indică adăugarea în soluție, în timp ce dacă elementul $x \notin B$ acesta nu trebuie adăugat. Matematic, acest lucru se exprimă:

$$\begin{aligned} A \cap B &= \{CurentA, RestA\} \cap B \\ &= \begin{cases} RestA \cap B & \text{dacă } CurentA \notin B \\ RestA \cap B \text{ și } \{CurentA\} & \text{dacă } CurentA \in B \end{cases} \\ A \cap B &= \emptyset, \text{ dacă } A = \emptyset \end{aligned}$$

Dacă un predicat pentru intersecție este deja scris, atunci intersecția pentru $RestA \cap B$ se determină imediat prin apel recursiv. Ca și pentru reuniune, verificările de tipul $CurentA \in B$ și $CurentA \notin B$ se realizează simplu prin apeluri `member(CurentA, B)`, respectiv `not(member(CurentA, B))`. Tot ca în cazul precedent, adăugarea unui element în soluție se realizează cu ajutorul șablonului. Deci predicatul intersecție sete:

```
%intersection/3
%intersection(multimea A, multimea B, multimea
reuniune)

intersection([HA|TA], B, [HA|U]) :-
    member(HA, B),
    intersection(TA, B, U).
intersection([HA|TA], B, U) :-
    not(member(HA, B)),
    intersection(TA, B, U).
intersection([], _, []).
```

O întrebare de forma:


```
| ?- intersection ([1,3,5,6],[2,3,4,5,7],R).
R = [3,5] ?
yes
```

Și evident, ca și pentru reuniune, dubla căutare în a doua mulțime poate fi evitată prin folosirea tăierii de backtracking, obținându-se:

```
intersection([HA|TA],B,[HA|U]):-
    member(HA,B),!,
    intersection(TA,B,U).
intersection([HA|TA],B,U):-
    intersection(TA,B,U).
intersection([],_,[]).
```

2.3 Diferența

Pentru diferența a două mulțimi se poate face un raționament analog.

$$\begin{aligned}
 A \setminus B &= \{ \text{CurentA}, \text{RestA} \} \setminus B \\
 &= \begin{cases} \text{RestA} \setminus B & \text{dacă } \text{CurentA} \in B \\ \text{RestA} \setminus B \text{ și } \{ \text{CurentA} \} & \text{dacă } \text{CurentA} \notin B \end{cases} \\
 A \setminus B &= \emptyset, \text{ dacă } A = \emptyset
 \end{aligned}$$

Aparent definiția diferenței este identică cu cea a reuniunii. Ce le deosebește este regula care se aplică pentru cazul când prin recursivitate A ajunge la \emptyset . Pentru acest caz, se aplică practic definiția intersecției. Ceea ce înseamnă că diferența are clauzele recursive ale reuniunii, și condiția de terminare a recursivității (faptul) de la intersecție. Predicatul se scrie:

```
%difference/3
%difference(multimea A, multimea B, multimea reuniune)
difference([HA|TA],B,U):-
    member(HA,B),
    difference(TA,B,U).
difference([HA|TA],B,[HA|U]):-
    not(member(HA,B)),
    difference(TA,B,U).
difference([],L,[]).
```

O întrebare de forma:

```
| ?- difference([1,3,5,6],[2,3,4,5,7],R).
R = [1,6] ?
yes
```

Evitarea dublei căutări în B, cu utilizarea tăierii backtracking-ului:

```
difference([HA|TA],B,U):-
    member(HA,B),!,
    difference(TA,B,U).
difference([HA|TA],B,[HA|U]):-
    difference(TA,B,U).
difference([],L,[]).
```

2.4 Exerciții și probleme propuse

1. Cum se modifică rezultatul obținut la întrebarea:

```
| ?- union([1,3,5,6],[2,3,4,5,7],R).
```

dacă în predicatul `union/3` se interschimbă pozițiile clauzelor 1 și 2 (cauza 1 devine a doua, respectiv a doua clauza se mută pe prima poziție)?

2. În ce fel este afectat rezultatul dacă în predicatul `union/3` faptul devine prima clauză? Ce implicații are această schimbare?

3. Ce soluție se obține la întrebarea

```
| ?- union([2,3,4,5,7],[1,3,5,6],R).
```

dacă predicatul `union/3` rămâne cu definiția inițială (din manual)?

4. Care este ordinea elementelor în soluție la întrebarea anterioară dacă se interschimbă poziția relativă a clauzelor 1 și 2 în definiția predicatului `intersection/3`?

5. Cum se modifică rezultatul obținut la întrebarea:

```
| ?- intersection([1,3,5,6],[2,3,4,5,7],R).
```

dacă în predicatul `intersection/3` se interschimbă pozițiile clauzelor 1 și 2 (cauza 1 devine a doua, respectiv a doua clauza se mută pe prima poziție)?

6. În ce fel este afectat rezultatul dacă în predicatul `intersection/3` faptul devine prima clauză? Ce implicații are această schimbare?

7. Ce soluție se obține la întrebarea
| ?- `intersection([2,3,4,5,7],[1,3,5,6],R)`.
dacă predicatul `intersection/3` rămâne cu definiția inițială (din manual)?

8. Care este ordinea elementelor în soluție la întrebarea anterioară dacă se interschimbă poziția relativă a clauzelor 1 și 2 în definiția predicatului `intersection /3`?

9. Cum se modifică rezultatul obținut la întrebarea:
| ?- `difference([1,3,5,6],[2,3,4,5,7],R)`.
dacă în predicatul `difference/3` se interschimbă pozițiile clauzelor 1 și 2 (cauza 1 devine a doua, respectiv a doua clauza se mută pe prima poziție)?

10. În ce fel este afectat rezultatul dacă în predicatul `difference/3` faptul devine prima clauză? Ce implicații are această schimbare?

11. Ce soluție se obține la întrebarea
| ?- `difference([2,3,4,5,7],[1,3,5,6],R)`.
dacă predicatul `difference/3` rămâne cu definiția inițială (din manual)?

12. Care este ordinea elementelor în soluție la întrebarea anterioară dacă se interschimbă poziția relativă a clauzelor 1 și 2 în definiția predicatului `difference/`

3. Arbori de execuție

În continuare se prezintă mecanismul Prolog prin intermediul unor exemple deja prezentate, de manipulare a listelor.

3.1 Member

În esență, execuția unei întrebări Prolog se reduce la rezolvarea unui sistem de ecuații liniare ce rezultă din unificări (întrebare/cap regulă, și parametri actuali/parametri formali).

Să vedem ce se întâmplă în fapt la o execuție a întrebării:

$q: \text{?-member}(2, [1, 2, 3])$.

Vom indexa cu numărul nodului din arborele de execuție (din care va rezulta arborele de deducție) fiecare din variabilele/parametri formal, pentru a identifica mai ușor locul apariției unei variabile. De asemenea, sistemul de ecuații va purta un indicator reprezentând nodul în care s-a format.

Mai întâi, conform regulilor de calcul și căutare, se încearcă matching-ul între cel mai din stânga (singurul în cazul de față) subscop din scopul original și prima clauză a predicatului cu același nume,

$m1: \text{member}(H, [H|T])$.
 $m2: \text{member}(H, [HL|T]) :-$
 $\quad \text{member}(H, T)$.

rezultând sistemul de ecuații dat de $q/m1$:

$$(1) \quad \begin{cases} H_1=2 \\ [H_1|T_1]=[1,2,3] \end{cases} \Rightarrow \begin{cases} H_1=2 \\ H_1=1 \\ T_1=[2,3] \end{cases} \quad \begin{array}{c} \textcircled{1} \\ q/m_1 \\ (\text{fail}) \end{array}$$

sistem evident incompatibil, datorită primelor două ecuații.

Backtracking-ul care se instalează automat datorită eșecului unificărilor conduce la suprapunerea aceleiași întrebări peste clauza $m2$ a predicatului, rezultând sistemul de ecuații dat de $q/m2$:

$$\begin{array}{c} \textcircled{1'} \\ q/m_2 \end{array}$$

$$(1') \quad \left\{ \begin{array}{l} H_1=2 \\ [HL_1|T_1]=[1,2,3] \end{array} \right. \Rightarrow \left\{ \begin{array}{l} H_1=2 \\ HL_1=1 \\ T_1=[2,3] \end{array} \right.$$

sistem care este compatibil (de data aceasta primele două ecuații ale sistemului nu mai încearcă să unifice aceeași variabilă la două valori distincte; avem o variabilă H_1 pentru argumentul 1, și o alta HL_1 pentru primul element al argumentului doi). Dar execuția nu s-a terminat, deoarece întrebarea s-a suprapus peste o clauză completă, a cărui succes este condiționat de succesul apelului din corp. Deci, scopul derivat din matching-ul anterior este:

$q1?-member(H, T)$.

și deoarece variabilele sunt aceleași cu variabilele din capul regulii, întrebarea este de fapt:

$q1?-member(H1, T1)$.

Adică

$q1?-member(2, [2, 3])$.

Rezolvarea acesteia respectă regulile de calcul și căutare (din lista de scopuri se alege cel mai din stânga, pentru soluționarea căruia alegându-se prima clauză a predicatului cu același nume), conducând la suprapunerea întrebării $q1$ peste capul primei clauze, adică $q1/m1$:

$$(2) \quad \left\{ \begin{array}{l} H_2=2 \\ [H_2|T_2]=[2,3] \end{array} \right. \Rightarrow \left\{ \begin{array}{l} H_2=2 \\ T_2=[3] \end{array} \right. \quad \begin{array}{c} \textcircled{1'} \\ | \\ \textcircled{2'} \end{array} \quad \begin{array}{l} q1/m3 \\ q1/m1 \end{array}$$

Fig. 3.1

sistemul fiind compatibil, matching-ul $q1/m1$ este reușit, și nodul 2 al arborelui de execuție va fi un nod al arborelui de deducție. Cum întrebarea curentă s-a suprapus cu succes peste un fapt, ea este adevărată necondiționat (în arbore acest lucru este remarcat prin faptul că nodul curent (2) este o frunză, subarborele având ca rădăcină acest nod fiind vid). Pe de altă parte, întrebarea $q1$ a fost singura în lista de întrebări (la selecția celei mai din stânga întrebări, nu mai era nici una spre dreapta), deci execuția s-a încheiat (și arborele nu are nici o continuare nici spre dreapta). Arborele de deducție are forma din Fig. 3.1.

Urmărind (la trasare) execuția avem:

```
% trace
| ?- member(2, [1, 2, 3]).
```

```

1      1 Call: member(2, [1, 2, 3]) ? //apel initial
2      2 Call: member(2, [2, 3]) ?   //apel recursiv
?      2      2 Exit: member(2, [2, 3]) ? //revenire din
                                           //apel recursiv
?      1      1 Exit: member(2, [1, 2, 3]) ? //revenire din
                                           //apel initial
yes

```

Sunt reprezentate doar apelurile de succes (matchingul întrebării `member(2, [1, 2, 3]` cu clauza 1 neputându-se realiza, datorită incompatibilității $H=2$ și simultan $H=1$).

3.2 Append

```
q: ?-append([1, 2], [3, 4], Result).
```

```

a1: append([], L, L).
a2: append([H|T], L, [H|R]) :-
    append(T, L, R).

```

Prima încercare, care suprapune întrebarea inițială cu prima clauză a predicatului q/a_1 (de fapt cel mai din stânga subscop al scopului inițial, care coincide cu întregul scop inițial, cu prima clauză în ordinea de definiție) conduce la sistemul (1) de ecuații liniare:

$$(1) \quad \begin{cases} [1, 2] = [] \\ L_1 = [3, 4] \\ \text{Result} = L_1 \end{cases} \quad \begin{array}{c} \textcircled{1} \\ q/a_1 \\ (\text{fail}) \end{array}$$

sistem incompatibil (ecuația 1!).

Eșecul conduce la suprapunerea q/a_2 și sistemul:

$$(1') \quad \begin{cases} [H_1|T_1] = [1, 2] \\ L_1 = [3, 4] \\ \text{Result} = [H_1|R_1] \end{cases} \Rightarrow \begin{cases} H_1 = 1, T_1 = [2] \\ L_1 = [3, 4] \\ \text{Result} = [H_1|R_1] \end{cases} \quad \begin{array}{c} \textcircled{1'} \\ q/a_2 \end{array}$$

Clauza fiind una completă (cu corp) succesul este condiționat de succesul întrebării care se formulează din :

```
append([H1|T1], L1, [H1|R1]) :-
    append(T1, L1, R1).
```

(se observă că variabilele din corpul clauzei sunt aceleași cu cele din capul clauzei), deci forma particulară a clauzei la momentul execuției este:

```
append([2], [3,4], [1|R1]) :-
    append([2], [3,4], R1).
```

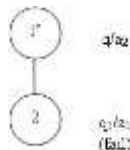
Și deci avem de rezolvat întrebarea derivată, adică întrebarea:

```
q1: ?- append(T1, L1, R1).
```

```
q1: ?- append([2], [3,4], R1).
```

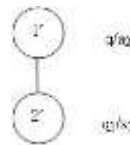
Imediat se constată că suprapunerea $q1/a1$ eșuează (ecuația1):

$$(2) \quad \begin{cases} [2]=[] \\ L_2=[3,4] \\ R_1=L_2 \end{cases}$$



și deci la reluare vom avea $q1/a2$:

$$(2') \quad \begin{cases} [H_2|T_2]=[2] \\ L_2=[3,4] \\ R_1=[H_2|R_2] \end{cases} \Rightarrow \begin{cases} H_2=[2], T_2=[] \\ L_2=[3,4] \\ R_1=[H_2|R_2] \end{cases}$$



care este urmat de apelul recursiv:

```
q2: ?- append(T2, L2, R2).
```

```
q2: ?- append([], [3,4], R2).
```

De data aceasta suprapunerea peste clauza 1 este una cu succes $q2/a1$:

$$(3) \quad \begin{cases} []=[] \\ L_3=[3,4] \\ R_2=L_3 \end{cases}$$

În acest moment execuția se încheie: întrebarea curentă s-a suprapus cu succes peste un fapt (deci scopul curent e adevărat necondiționat, și deci nodul curent este o frunză, nodul neavând nici un succesor), iar din scopul inițial, la selectarea celui mai din stânga subscop, niciodată n-a existat un altul spre dreapta (și deci nu sunt alte întrebări care își așteaptă răspunsul; din punct de vedere al arborelui, nici nodul curent, și nici un alt nod, creat anterior, nu are noduri frate spre dreapta).

Răspunsul se obține dacă luăm în considerare toate sistemele de ecuații valide (adică (1'), (2') și (3)), iar arborele de deducție este cel din Fig. 3.2.

Rezultatul este:

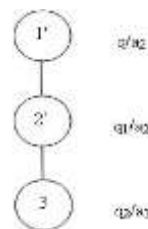


Fig. 3.2

(1')	(1')
Result = [H ₁ R ₁]	= [1 R ₁]
(2')	rescriere (2')
= [1 [H ₂ R ₂]	= [1,H ₂ R ₂] = [1,2 R ₂]
(3)	(3)
= [1,2 L ₃]	= [1,2,3,4].

La execuția cu trasare:

```
% trace
| ?- append([1,2],[3,4],Result).
    1      1 Call: append([1,2],[3,4],_541) ?
    2      2 Call: append([2],[3,4],_1178) ?
    3      3 Call: append([], [3,4], _1824) ?
    3      3 Exit: append([], [3,4], [3,4]) ?
    2      2 Exit: append([2],[3,4],[2,3,4]) ?
    1      1 Exit: append([1,2],[3,4],[1,2,3,4]) ?
Result = [1,2,3,4] ?
yes
```

3.3 Reuniune

```
u1: union([HA|TA],B,U):-
    member(HA,B),
    union(TA,B,U).
u2: union([HA|TA],B,[HA|U]):-
    union(TA,B,U).
u3: union([ ],L,L).

m1: member(H,[H|T]).
m2: member(H,[HL|T]):-
    member(H,T).

q: ?- union([3,5,1],[1,2,3,4],Result).
```

În conformitate cu regulile cu care deja ne-am obișnuit, întrebarea *q* se suprapune peste capul primei clauze a predicatului cu același nume, *u1*, unificarea reușită necesitând continuarea cu scopurile din corpul clauzei, în

timp ce eșecul instalează imediat mecanismul de backtracking, care propune unificarea aceluiași scop cu următoarea clauză. Să analizăm așadar rezultatul unificării perechilor de parametri actuali/formali, rezultate din matchingul q/u_1 :

$$(1) \quad \begin{cases} [HA_1|TA_1]=[3,1,5] \\ B_1=[1,2,3,4] \\ \text{Result}=U_1 \end{cases} \Rightarrow \begin{cases} HA_1=3, TA_1=[1,5] \\ B_1=[1,2,3,4] \\ \text{Result}=U_1 \end{cases} \quad \begin{array}{c} \textcircled{1} \\ q/u_1 \end{array}$$

Sistemul este compatibil, și deci unificarea dintre întrebare și capul clauzei se realizează cu succes, ceea ce implică transformarea corpului respectivei clauze în noul scop (întrebare) pentru care se așteaptă răspuns, adică:

$$\begin{array}{l} q_1: ?\text{-member}(HA_1, B_1), \text{union}(TA_1, B_1, U_1) . \\ q_1: ?\text{-member}(3, [1, 2, 3, 4]), \text{union}([1, 5], [1, 2, 3, 4], U_1) . \\ \leftarrow \text{-----} q_{11} \text{-----} \rightarrow \quad \leftarrow \text{-----} q_{12} \text{-----} \rightarrow \end{array}$$

În conformitate cu regula de calcul, dintr-un scop compus se execută mai întâi cel mai din stânga, deci, mai întâi:

$$q_{11}: ?\text{-member}(3, [1, 2, 3, 4]) .$$

Și doar în caz de succes, și după obținerea acestuia se trece la restul scopului (q_{12}).

Pentru rezolvarea lui vom urmări doar nodurile de succes, și sistemele de ecuații corespunzătoare (suprapunerile care conduc la eșec le vom menționa doar, pentru a justifica felul în care se ajunge la clauza selectată, fără a mai scrie sistemele de ecuații corespunzătoare).

Tentativa de suprapunere q_{11}/m_1 va eșua (prin eșecul unificării simultane a lui H cu 3 și 1), ceea ce conduce la q_{11}/m_2 și sistemul:

$$(2) \quad \begin{cases} H_1=3 \\ [HL_1|T_1]=[1,2,3,4] \end{cases} \Rightarrow \begin{cases} H_1=3 \\ HL_1=1 \\ T_1=[2,3,4] \end{cases} \quad \begin{array}{c} \textcircled{1} \\ | \\ \textcircled{2'} \end{array} \quad \begin{array}{c} q/u_1 \\ q_{11}/m_2 \end{array}$$

cu întrebarea derivată:

$$\text{member}(H_1, T_1) \equiv \text{member}(3, [2, 3, 4]) .$$

Această întrebare va avea succes tot doar la suprapunerea peste clauza a doua, m_2 , rezultând sistemul:

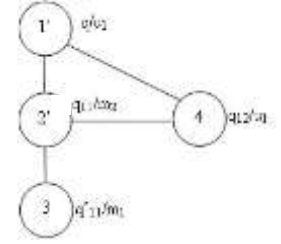
$$(3) \quad \begin{cases} H_2=3 \\ [HL_2|T_2]=[2,3,4] \end{cases} \Rightarrow \begin{cases} H_2=3 \\ HL_2=2 \\ T_2=[3,4] \end{cases} \quad \begin{array}{c} \textcircled{1} \\ | \\ \textcircled{2'} \\ | \\ \textcircled{3} \end{array} \quad \begin{array}{c} q/u_1 \\ q_{11}/m_2 \\ q'_{11}/m_1 \end{array}$$

și întrebarea derivată:

$\text{member}(H_2, T_2) \equiv \text{member}(3, [3, 4])$.

A cărei suprapunere peste m1 reușește de data aceasta.

$$(4) \quad \begin{cases} H_3=3 \\ [H_3|T_3]=[3,4] \end{cases} \Rightarrow \begin{cases} H_2=3 \\ T_3=[4] \end{cases}$$



sistem de ecuații liniare compatibil determinat, și deci întrebarea $\text{member}(3, [3, 4])$ se termină cu succes, ceea ce implică succesul întrebării $\text{member}(3, [2, 3, 4])$ care la rândul său implică succesul întrebării $\text{member}(3, [12, 3, 4])$. Deci q_{11} se termină cu succes.

Revenim acum la întrebarea

$q1: ?\text{-member}(3, [1, 2, 3, 4]), \text{union}([1, 5], [1, 2, 3, 4], U1)$.

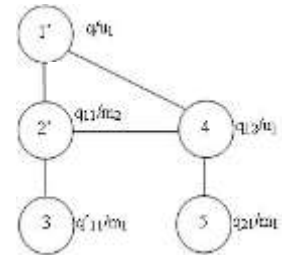
←----- q_{11} -----→ ←----- q_{12} -----→

și facem observația că odată ce q_{11} s-a executat cu succes, se alege următoarea întrebare (de la stânga spre dreapta) din $q1$; deci q_{12} este întrebarea care urmează a fi executată. Menționăm în același timp că, la nivelul arborelui de execuție (și deci în cele din urmă și a celui de deducție) nodul corespunzător lui q_{12} va fi la același nivel cu nodul pentru q_{11} , și va fi un nod frate spre dreapta al acestuia. Deci, în arborele de execuție (și deci și în cel de deducție) trecerea din capul clauzei în corpul clauzei (peste semnul:-) se reprezintă prin trecerea de la nivelul curent la nivelul următor al arborelui (se coboară un nivel), în timp ce trecerea de la scopul curent la cel următor (de la un apel la apelul următor în corp, deci sintactic peste o virgulă) se reprezintă prin trecerea de la stânga la dreapta, în cadrul nivelului curent.

Revenind la execuție:

$q12: ?\text{-union}([1, 5], [1, 2, 3, 4], U1)$.

$$(5) \quad \begin{cases} [HA_5|TA_5]=[1,5] \\ B_5=[1,2,3,4] \\ U_1= U_5 \end{cases} \Rightarrow \begin{cases} HA_5=1, TA_5=[5] \\ B_5=[1,2,3,4] \\ U_1= U_5 \end{cases}$$



și întrebarea derivată:

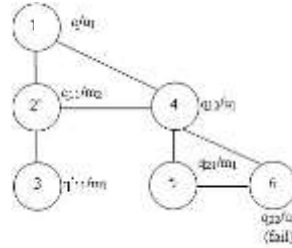
$q2: ?\text{-member}(HA_5, B_5), \text{union}(TA_5, B_5, U_5)$.

$q2: ?\text{-member}(1, [1, 2, 3, 4]), \text{union}([5], [1, 2, 3, 4], U_5)$.

←----- q_{21} -----→ ←----- q_{22} -----→

Din care se alege spre execuție mai întâi $q_{21}/m1$:

$$(6) \quad \begin{cases} H_6=1 \\ [H_6|T_6]=[1,2,3,4] \end{cases} \Rightarrow \begin{cases} H_2=1 \\ T_2=[2,3,4] \end{cases}$$



Sistemul fiind compatibil determinat, iar clauza fiind un fapt (necondiționat de execuția vreunui alt apel în corp), suprapunerea se termină cu succes.

Urmează acum q_{22} :

$q_{22} \text{ ?-union}([5], [1,2,3,4], U5)$.

În mod firesc ar urma $q_{22}/u1$; deși suprapunerea întrebare/cap clauză are succes, întrebarea derivată ar fi:

$\text{?-member}(5, [1,2,3,4], \text{union}([], [1,2,3,4], U7))$.

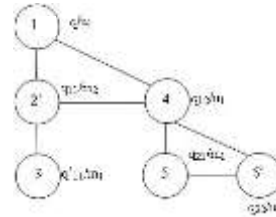
și este evident că

$\text{?-member}(5, [1,2,3,4])$,

eșuează. Backtracking-ul care se lansează implicit produce re-execuția întrebării q_{22} prin suprapunerea acesteia peste capul clauzei a doua a predicatului, $/u1$ și deci unificarea $q_{22}/u1$ care va fi una cu succes. Deci:

$\text{union}([5], [1,2,3,4], U5) / \text{union}([HA|TA], B, [HA|U])$

$$(7) \quad \begin{cases} [HA_7|TA_7]=[5] \\ B_7=[1,2,3,4] \\ U_5=[HA_7|U_7] \end{cases} \Rightarrow \begin{cases} HA_7=5, TA_7=[] \\ B_7=[1,2,3,4] \\ U_5=[5|U_7] \end{cases}$$



sistemul fiind compatibil determinat, se continuă execuția, corpul clauzei cu care s-a realizat ultima unificare devenind noua întrebare, adică:

$q_3 \text{ ?-union}(TA_7, B_7, U_7) \equiv \text{union}([], [1,2,3,4], U_7)$.

Această întrebare se va rezolva cu succes doar prin suprapunerea peste $u3$, deci:

$\text{union}([], [1,2,3,4], U_7) / \text{union}([], L, L)$.
producând sistemul:

$$(8) \quad \begin{cases} []=[] \\ L_8=[1,2,3,4] \\ U_7=L_8 \end{cases}$$

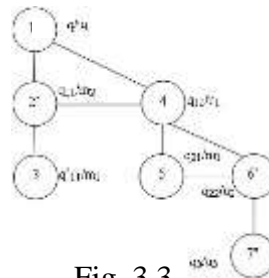


Fig. 3.3

sistem compatibil determinat. Cum întrebarea curentă s-a suprapus peste un fapt, ea este adevărată necondiționat, și deci nu este nici o întrebare derivată din aceasta. Pe de altă parte, în scopurile compuse pe care le-am avut de-a lungul execuției (q_1 , q_2) din toate s-a selectat spre execuție și cel mai din dreapta sub-scop (q_{12} și respectiv q_{22}), deci execuția s-a încheiat cu succes, iar răspunsul:

$$\begin{array}{cccccc} (1) & (5) & (7) & (7) & (8) & (8) \\ \text{Result} = U_1 & = U_5 & = [HA_7|U_7] & = [5|U_7] & = [5|L_8] & = \\ (8) & & \text{rescriere} & & & \\ = [5|[1,2,3,4]] & = [5,1,2,3,4]. & & & & \end{array}$$

Result=[5,1,2,3,4].

Arborele de deducție este cel din Fig. 3.3.

Dacă urmărim la execuție trasarea întrebării pentru care tocmai am dezvoltat arborele de deducție:

```
q(Result):-union([3,5,1],[1,2,3,4],Result).
% trace
| ?- q(Result).
      1      1 Call: q(_473) ?
      2      2 Call: union([3,5,1],[1,2,3,4],_473) ?
      3      3 Call: member(3,[1,2,3,4]) ?
      4      4 Call: member(3,[2,3,4]) ?
      5      5 Call: member(3,[3,4]) ?
?      5      5 Exit: member(3,[3,4]) ?
?      4      4 Exit: member(3,[2,3,4]) ?
?      3      3 Exit: member(3,[1,2,3,4]) ?
      6      3 Call: union([5,1],[1,2,3,4],_473) ?
      7      4 Call: member(5,[1,2,3,4]) ?
      8      5 Call: member(5,[2,3,4]) ?
      9      6 Call: member(5,[3,4]) ?
     10      7 Call: member(5,[4]) ?
     11      8 Call: member(5,[]) ?
     11      8 Fail: member(5,[]) ?
     10      7 Fail: member(5,[4]) ?
      9      6 Fail: member(5,[3,4]) ?
      8      5 Fail: member(5,[2,3,4]) ?
      7      4 Fail: member(5,[1,2,3,4]) ?
     12      4 Call: union([1],[1,2,3,4],_6210) ?
     13      5 Call: member(1,[1,2,3,4]) ?
?      13      5 Exit: member(1,[1,2,3,4]) ?
     14      5 Call: union([], [1,2,3,4],_6210) ?
     14      5 Exit: union([], [1,2,3,4], [1,2,3,4]) ?
```

```

      12      4 Exit: union([1],[1,2,3,4],[1,2,3,4]) ?
      6      3      Exit:
union([5,1],[1,2,3,4],[5,1,2,3,4]) ?
      2      2      Exit:
union([3,5,1],[1,2,3,4],[5,1,2,3,4]) ?
      1      1 Exit: q([5,1,2,3,4]) ?
Result = [5,1,2,3,4] ?
Yes

```

3.4 Intersecție

```

i1: intersection([HA|TA],B,[HA|U]):-
    member(HA,B),!,
    intersection(TA,B,U).
i2: intersection([HA|TA],B,U):-
    intersection(TA,B,U).
i3: intersection([],_,[]).

m1: member(H,[H|T]).
m2: member(H,[HL|T]):-
    member(H,T).
q: ?- intersection([2,4,5],[1,2,3,5,6],Result).

```

Pentru acest exemplu vom trasa arborele de deducție și sistemele de ecuații liniare care se generează la nivelul fiecărui nod, justificând foarte pe scurt modalitatea în care unificările se produc, și soluția (întrebare/cap clauză) care se aplică. Vom insista în schimb asupra influenței pe care tăierea de backtracking o are asupra arborelui de deducție și execuție, și implicit asupra soluției obținute.


Datorită faptului că o unificare reușită dintre întrebare și capul unei clauze, pentru a conduce la succes final, este condiționată de succesul scopului derivat din corpul clauzei, de fiecare dată când vom aplica unificarea (întrebare / cap clauză) vom privi în avans, să intuim dacă scopurile din corp pot sau nu să aibă succes.

Suprapunerea $q/i1$ dintre întrebare/cap clauză are succes, și în corp sub-scopurile derivate vor avea succes (2 fiind un element al listei $[1,2,3,5,6]$).

```

intersection([2,4,5],[1,2,3,5,6],Result)/
intersection([HA|TA],B,[HA|U])

```

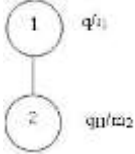
$$(1) \quad \begin{cases} [HA_1|TA_1]=[2,4,5] \\ B_1=[1,2,3,5,6] \\ \text{Result}=[HA_1|U_1] \end{cases} \Rightarrow \begin{cases} HA_1=2, TA_1=[4,5] \\ B_1=[1,2,3,5,6] \\ \text{Result}=[2|U_1] \end{cases}$$


sistem compatibil determinat; corpul clauzei devine noul scop:

```
q1: ?- member(HA1,B1),!,intersection(TA1,B1,U1).
q1: ?- member(2,[1,2,3,5,6]),!,intersection([4,5],[1,2,3,5,6],U1).
<-----q11-----> <-----q12----->
```

```
q11: ?-member(2,[1,2,3,5,6]).
```

```
q11/m2 ≡ member(2,[1,2,3,5,6])/member(H,[HL|T])
```

$$(2) \quad \begin{cases} H_2=2 \\ [HL_2|T_2]=[1,2,3,5,6] \end{cases} \Rightarrow \begin{cases} H_2=2 \\ HL_2=1, T_2=[2,3,5,6] \end{cases}$$


Sistem compatibil determinat, care implică următoarea întrebare derivată (din corpul m2):

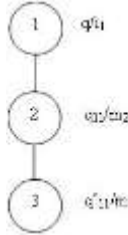
```
member(H2,T2) ≡ member(2,[2,3,5,6]).
```

Această întrebare este derivată din q_{11} , deci pentru finalizarea acesteia, $member(2,[2,3,5,6])$ trebuie finalizată. Deci, în acest moment, forma întrebării care se află pentru rezolvare este:

```
q'1: ?-member(2,[2,3,5,6]),!,intersection([4,5],[1,2,3,5,6],U1).
<-----q'11-----> <-----q12----->
```

Începând cu sub-scopul aflat la stânga:

```
q'11/m1 ≡ member(2,[2,3,5,6])/member(H,[H|T]).
```

$$(3) \quad \begin{cases} H_3=2 \\ [H_3|T_3]=[2,3,5,6] \end{cases} \Rightarrow \begin{cases} H_3=2 \\ H_3=2, T_3=[2,3,5,6] \end{cases}$$


Sistem compatibil determinat, care încheie cu succes unificarea $q'_{11}/m1$, deoarece $m1$ fiind fapt, este adevărat necondiționat, și deci orice unificare reușită întrebare/fapt este finală. Ceea ce înseamnă că în $q'1$ am terminat cu q'_{11} și trecem la următorul subscop:

$q'1: ?-!, \text{intersection}([4,5], [1,2,3,5,6], U1) .$

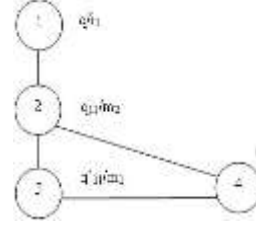
←----- q_{12} -----→

Tăierea de backtracking (!) se execută, are succes, și nu face revenire; în arborele de execuție reprezintă un nod, (4).

(4) $!!$ adevărat necondiționat.

$q'1: ?-\text{intersection}([4,5], [1,2,3,5,6], U1) .$

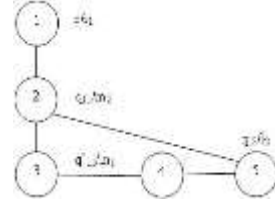
←----- q_{12} -----→



Inițial matchingul $q_{12} / i1$ doar întrebare / cap clauză are succes; dar privind la pasul următor, sub-scopul derivat va eșua, și deci backtracking-ul se va instala. În pasul următor (singurul pe care îl urmărim explicit), $q_{12} / i2$.

$q_{12} / i2 \equiv \text{intersection}([4,5], [1,2,3,5,6], U1) /$
 $\text{intersection}([HA|TA], B, U)$

(5) $\begin{cases} [HA_5|TA_5]=[4,5] \\ \{B_5=[1,2,3,5,6] \Rightarrow \\ U_1=U_5 \end{cases} \quad \begin{cases} [HA_5=4, TA_5=[5] \\ \{B_5=[1,2,3,5,6] \\ U_1=U_5 \end{cases}$



Sistem compatibil determinat; $i2$ fiind o clauză

completă, succesul $q_{12} / i2$ este condiționat de succesul scopului derivat din corpul clauzei, adică:

$\text{intersection}(TA_5, B_5, U_5)$

\equiv

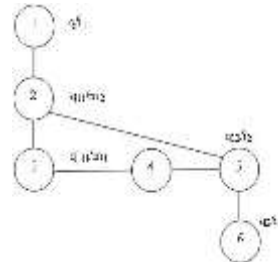
$\text{intersection}([5], [1,2,3,5,6], U_5)$

care devine noua întrebare spre execuție (și unica, din întrebarea curentă fiind deja selectat cel mai din dreapta sub-scop, q_{12}).

$q'2: ?-\text{intersection}([5], [1,2,3,5,6], U_5) .$

$q'2 / i1 \equiv \text{intersection}([5], [1,2,3,5,6], U_5) /$
 $\text{intersection}([HA|TA], B, [HA|U])$

(6) $\begin{cases} [HA_6|TA_6]=[5] \\ \{B_6=[1,2,3,5,6] \Rightarrow \\ U_5=[HA_6|U_6] \end{cases} \quad \begin{cases} [HA_6=5, TA_6=[] \\ \{B_6=[1,2,3,5,6] \\ U_5=[5|U_6] \end{cases}$



Sistem compatibil determinat, cu întrebarea derivată din corpul $i1$:

$q'2: ?-\text{member}(HA_6, B_6), !, \text{intersection}(TA_6, B_6, U_6) .$

$q'2: ?-\text{member}(5, [1,2,3,5,6]), !, \text{intersection}([], [1,2,3,5,6], U_6) .$

←----- q'_{21} -----→ ←----- q'_{22} -----→

De la stânga la dreapta:

$q'_{21} : ?-member(5, [1, 2, 3, 5, 6]) .$

$q'_{21}/m2 \equiv member(5, [1, 2, 3, 5, 6]) / member(H, [HL|T]) .$

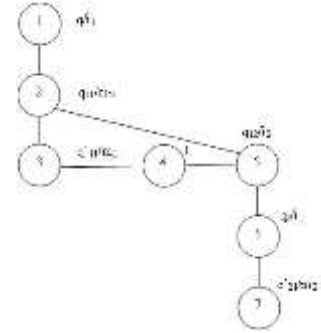
$$(7) \quad \left\{ \begin{array}{l} H_7=5 \\ [HL_7|T_7]=[1,2,3,5,6] \end{array} \right. \Rightarrow \left\{ \begin{array}{l} H_7=5 \\ HL_7=1, T_7=[2,3,5,6] \end{array} \right.$$

Sistem compatibil determinat, care implică următoarea întrebare derivată (din corpul m2):

$member(H_7, T_7)$

$\equiv member(5, [2, 3, 5, 6]) .$

Această întrebare este derivată din q'_{21} , deci pentru finalizarea acesteia, $member(5, [2, 3, 5, 6])$ trebuie finalizată. Deci, în acest moment, forma întrebării care se află pentru rezolvare este:



$q''_{22} : ?-member(5, [2, 3, 5, 6]), !, intersection([], [1, 2, 3, 5, 6], U6) .$

←----- q'_{21} -----→ ←----- q'_{22} -----→

Începând cu sub-scopul aflat la stânga:

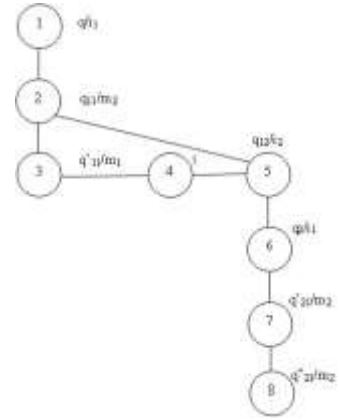
$q'_{21}/m2 \equiv member(5, [2, 3, 5, 6]) / member(H, [HL|T]) .$

$$(8) \quad \left\{ \begin{array}{l} H_8=5 \\ [HL_8|T_8]=[2,3,5,6] \end{array} \right. \Rightarrow \left\{ \begin{array}{l} H_8=5 \\ HL_8=2, T_8=[3,5,6] \end{array} \right.$$

Sistemul (8) fiind și el compatibil determinat, iar clauza cu care s-a realizat unificarea fiind din nou una completă (cu corp), soluționarea întrebării este condiționată de soluționarea întrebării derivate:

$member(H_8, T_8) \equiv member(5, [3, 5, 6]) .$

Această întrebare este derivată din q'_{21} , deci pentru finalizarea acesteia întrebarea $member(5, [3, 5, 6])$ trebuie rezolvată. Deci, în acest moment, forma întrebării care se află pentru rezolvare este:



$q'''_{22} : ?-member(5, [3, 5, 6]), !, intersection([], [1, 2, 3, 5, 6], U6) .$

←--- q'_{21} ---→ ←----- q'_{22} -----→

Sub-scopul aflat la extremitatea stângă (furnizat de regula de calcul):

$q'''_{21}/m2 \equiv member(5, [3, 5, 6]) / member(H, [HL|T]) .$

$$(9) \quad \begin{cases} H_9=5 \\ [HL_9|T_9]=[3,5,6] \end{cases} \Rightarrow \begin{cases} H_9=5 \\ HL_9=3, T_9=[5,6] \end{cases}$$

Din care derivă o nouă întrebare

$\text{member}(H_9, T_9) \equiv$
 $\text{member}(5, [5, 6])$.

care înlocuiește pe q''_{21} în q''_2 , și aceasta devine:

$q^{iv}_2: ?-\text{member}(5, [5, 6]), !, \text{intersection}([], [1, 2, 3, 5, 6], U6)$.

$\leftarrow q^{iv}_{21} \rightarrow \leftarrow q'_{22} \rightarrow$

$q^{iv}_{21}/m1 \equiv \text{member}(5, [5, 6]) / \text{member}(H, [H|T])$.

$$(10) \quad \begin{cases} H_{10}=5 \\ [H_{10}|T_{10}]=[5,6] \end{cases} \Rightarrow \begin{cases} H_{10}=5 \\ H_{10}=5, T_{10}=[6] \end{cases}$$

Și în sfârșit, întrebarea curentă suprapunându-se peste un fapt, răspunsul se obține odată cu soluția la sistemul (10). Întrebarea care a rămas:

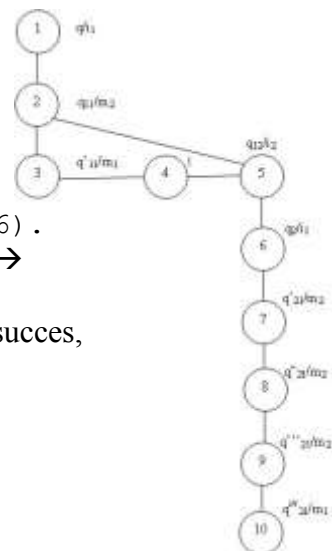
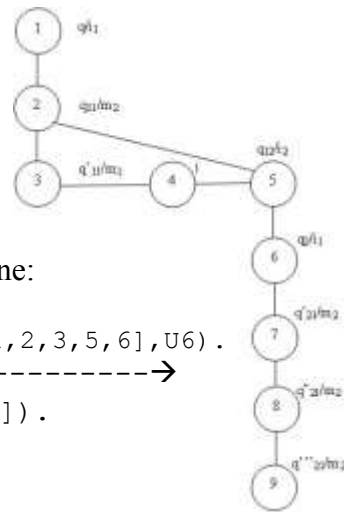
$q^{iv}_2: ?-!, \text{intersection}([], [1, 2, 3, 5, 6], U6)$.

$\leftarrow q'_{22} \rightarrow$

Tăierea de backtracking (!) se execută imediat, are succes, și nu face revenire (nodul 11 în arbore)

(11) $! = !$

și deci urmează:



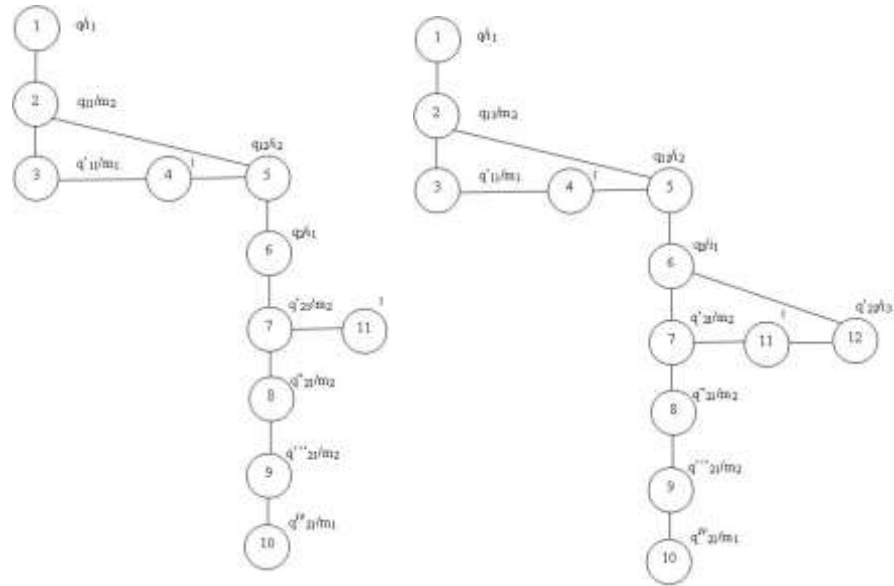


Fig.3.4

```

qiv2: ?-intersection([], [1,2,3,5,6], U6) .
      ←-----q'22-----→
q'22/i3      ≡      intersection([], [1,2,3,5,6], U6) /
intersection([], _, []).
(12)  { []=[]
      { _=[1,2,3,5,6]
      { U6=[]

```

Suprapunere peste fapt, care produce terminarea execuției întrebării curente, care fiind ultima, conduce la terminarea întregii execuții, cu forma finală a arborelui conținând 12 noduri.

Soluția se obține din ecuațiile celor 12 sisteme:

(1)	(1)	(5)	(6)
Result = [HA ₁ U ₁]	= [2 U ₁]	= [2 U ₅]	= [2 [HA ₆ U ₆]]
(6)	rescriere	(12)	rescriere
= [2 [5 U ₆]]	= [2,5 U ₆]	= [2,5 []]	= [2,5].

Result=[2,5].

Arborele de deducție din Fig. 3.4.

```

i(Result):-intersection([2,4,5], [1,2,3,5,6], Result) .
% trace
| ?- i(Result) .

```

```

1      1 Call: i(_473) ?
2      2 Call: intersection([2,4,5],[1,2,3,5,6],_473)
?
3      3 Call: member(2,[1,2,3,5,6]) ?
4      4 Call: member(2,[2,3,5,6]) ?
?      4      4 Exit: member(2,[2,3,5,6]) ?
?      3      3 Exit: member(2,[1,2,3,5,6]) ?
5      3 Call: intersection([4,5],[1,2,3,5,6],_1749)
?
6      4 Call: member(4,[1,2,3,5,6]) ?
7      5 Call: member(4,[2,3,5,6]) ?
8      6 Call: member(4,[3,5,6]) ?
9      7 Call: member(4,[5,6]) ?
10     8 Call: member(4,[6]) ?
11     9 Call: member(4,[ ]) ?
11     9 Fail: member(4,[ ]) ?
10     8 Fail: member(4,[6]) ?
9      7 Fail: member(4,[5,6]) ?
8      6 Fail: member(4,[3,5,6]) ?
7      5 Fail: member(4,[2,3,5,6]) ?
6      4 Fail: member(4,[1,2,3,5,6]) ?
12     4 Call: intersection([5],[1,2,3,5,6],_1749) ?
13     5 Call: member(5,[1,2,3,5,6]) ?
14     6 Call: member(5,[2,3,5,6]) ?
15     7 Call: member(5,[3,5,6]) ?
16     8 Call: member(5,[5,6]) ?
?      16     8 Exit: member(5,[5,6]) ?
?      15     7 Exit: member(5,[3,5,6]) ?
?      14     6 Exit: member(5,[2,3,5,6]) ?
?      13     5 Exit: member(5,[1,2,3,5,6]) ?
17     5 Call: intersection([], [1,2,3,5,6],_5642) ?
17     5 Exit: intersection([], [1,2,3,5,6],[]) ?
12     4 Exit: intersection([5],[1,2,3,5,6],[5]) ?
5      3 Exit: intersection([4,5],[1,2,3,5,6],[5]) ?
2      2      Exit:
intersection([2,4,5],[1,2,3,5,6],[2,5]) ?
1      1 Exit: i([2,5]) ?
Result = [2,5] ? ;
no

```

3.5 Diferența

```

d1: difference([HA|TA],B,U):-
    member(HA,B),!,
    difference(TA,B,U).
d2: difference([HA|TA],B,[HA|U]):-
    difference(TA,B,U).

```

d3: difference([], L, []).

m1: member(H, [H|T]).

m2: member(H, [HL|T]) :-
member(H, T).

q: ?- difference([1, 3, 5, 6], [2, 3, 4, 5, 7], Result).

q/d2 = difference([1, 3, 5, 6], [2, 3, 5, 7], Result) /
difference([HA|TA], B, [HA|U])

$$(1) \quad \begin{cases} [HA_1|TA_1]=[1,3,5,6] \\ B_1=[2,3,5,7] \\ \text{Result}=[HA_1|U_1] \end{cases} \Rightarrow \begin{cases} HA_1=1, TA_1=[3,5,6] \\ B_1=[2,3,5,7] \\ \text{Result}=[1|U_1] \end{cases} \quad \begin{array}{c} \textcircled{1} \quad q/d_2 \end{array}$$

q1: ?-difference(TA1, B1, U1) = difference([3, 5, 6], [2, 3, 5, 7], U1).

q1/d1 = difference([3, 5, 6], [2, 3, 5, 7], U1) / difference([HA|TA], B, U).

$$(2) \quad \begin{cases} [HA_2|TA_2]=[3,5,6] \\ B_2=[2,3,5,7] \\ U_1=U_2 \end{cases} \Rightarrow \begin{cases} HA_2=3, TA_2=[5,6] \\ B_2=[2,3,5,7] \\ U_1=U_2 \end{cases} \quad \begin{array}{c} \textcircled{1} \quad q/d_2 \\ \textcircled{2} \quad q_1/d_1 \end{array}$$

q2: ?-member(HA2, B2), !, difference(TA2, B2, U2).

←-----q₂₁-----→ ←-----q₂₂-----→

q2: ?-member(3, [2, 3, 5, 7]), !, difference([5, 6], [2, 3, 5, 7], U2).

←-----q₂₁-----→ ←-----q₂₂-----→

q₂₁/m2 = member(3, [2, 3, 5, 7]) / member(H, [HL|T]).

$$(3) \quad \begin{cases} H_3=3 \\ [HL_3|T_3]=[2,3,5,7] \end{cases} \Rightarrow \begin{cases} H_3=3 \\ HL_3=2, T_3=[3,5,7] \end{cases} \quad \begin{array}{c} \textcircled{1} \quad q/d_2 \\ \textcircled{2} \quad q_1/d_1 \\ \textcircled{3} \quad q_1/m_2 \end{array}$$

⇒ q'₂₁: ?-member(3, [2, 3, 5, 7]). ⇒ q'₂

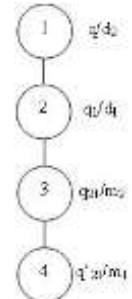
q'₂: ?-member(H3, T3), !, difference(TA2, B2, U2).

q'₂: ?-member(3, [3, 5, 7]), !, difference([5, 6], [2, 3, 5, 7], U2).

←-----q'₂₁-----→ ←-----q₂₂-----→

q'₂₁: ?-member(3, [3, 5, 7]).

q'₂₁/m1 = member(3, [3, 5, 7]) /
member(H, [H|T]).



$$(4) \quad \begin{cases} H_4=3 \\ [H_4|T_4]=[3,5,7] \end{cases} \Rightarrow \begin{cases} H_4=3 \\ H_4=3, T_4=[5,7] \end{cases}$$

fapt. Deci q'_{21} adevărat și soluționat de (4). $\Rightarrow q'2$

$q'2: ?-!, \text{difference}([5,6],[2,3,5,7],U2).$
 $\leftarrow \text{-----} q_{22} \text{-----} \rightarrow$

(5) !=!

$q'2: ?-\text{difference}([5,6],[2,3,5,7],U2).$
 $\leftarrow \text{-----} q_{22} \text{-----} \rightarrow$

$q_{22}: ?-\text{difference}([5,6],[2,3,5,7],U2).$

$q_{22}/d1$
 $\text{difference}([5,6],[2,3,5,7],U2) /$
 $\text{difference}([HA|TA],B,U)$

$$(6) \quad \begin{cases} [HA_6|TA_6]=[5,6] \\ TA_6=[6] \\ B_6=[2,3,5,7] \\ U_2=U_6 \end{cases} \Rightarrow \begin{cases} HA_6=5, \\ B_6=[2,3,5,7] \\ U_2=U_6 \end{cases}$$

$q'2 \Rightarrow q3$

$q3: ?-\text{member}(HA_6,B_6), !, \text{difference}(TA_6,B_6,U6).$
 $\leftarrow \text{-----} q_{31} \text{-----} \rightarrow \quad \leftarrow \text{-----} q_{32} \text{-----} \rightarrow$

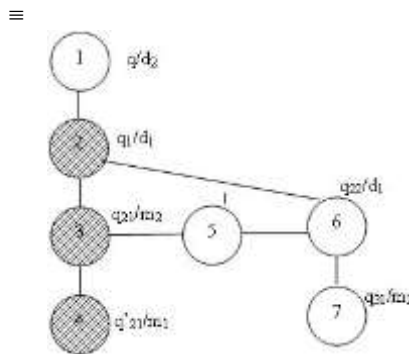
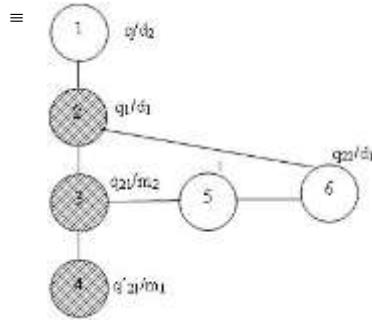
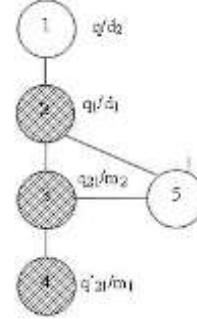
$q3: ?-\text{member}(5,[2,3,5,7]), !, \text{difference}([6],[2,3,5,7],U6).$
 $\leftarrow \text{-----} q_{31} \text{-----} \rightarrow \quad \leftarrow \text{-----} q_{32} \text{-----} \rightarrow$

$q_{31}: ?-\text{member}(5,[2,3,5,7]).$

$q_{31}/m2$
 $\text{member}(5,[2,3,5,7]) /$
 $\text{member}(H,[HL|T])$

$$(7) \quad \begin{cases} H_7=5 \\ [HL_7|T_7]=[2,3,5,7] \end{cases} \Rightarrow$$

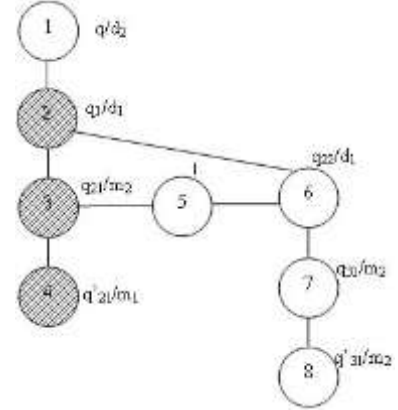
$$\Rightarrow \begin{cases} H_7=5 \\ HL_7=2, T_7=[3,5,7] \end{cases}$$



$q_{31} \Rightarrow q'_{31}$
 $?- \text{member}(H7, T7) \equiv$
 $\text{member}(5, [3, 5, 7]) .$
 $\Rightarrow q'3$
 $q'3: ?- \text{member}(H7, T7), !, \text{difference}(TA6, B6, U6) .$
 $q'3: ?- \text{member}(5, [3, 5, 7]), !, \text{difference}([6], [2, 3, 5, 7], U6) .$
 $\leftarrow \text{-----} q'_{31} \text{-----} \rightarrow \leftarrow \text{-----} q_{32} \text{-----} \rightarrow$
 $q'_{31}: ?- \text{member}(5, [3, 5, 7]) .$
 $q'_{31}/m2 \equiv \text{member}(5, [3, 5, 7]) /$
 $\text{member}(H, [HL | T])$

$(8) \quad \begin{cases} H_8=5 \\ [HL_8|T_8]=[3,5,7] \end{cases} \Rightarrow$

$\Rightarrow \begin{cases} H_8=5 \\ HL_8=3, T_8=[5,7] \end{cases}$



$q'_{31} \Rightarrow q''_{31}: ?- \text{member}(H8, T8) \equiv$
 $\text{member}(5, [5, 7]) .$

$\Rightarrow q''3$

$q''3: ?- \text{member}(H8, T8), !, \text{difference}(TA6, B6, U6) .$

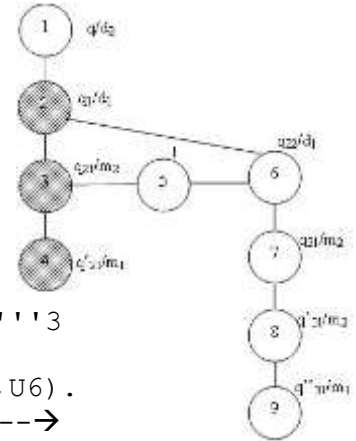
$q''3: ?- \text{member}(5, [5, 7]), !, \text{difference}([6], [2, 3, 5, 7], U6) .$

$\leftarrow \text{-----} q''_{31} \text{-----} \rightarrow \leftarrow \text{-----} q_{32} \text{-----} \rightarrow$
 $q''_{31}: ?- \text{member}(5, [5, 7]) .$

$q''_{31}/m1 \equiv \text{member}(5, [5, 7]) / \text{member}(H, [H | T])$

$(9) \quad \begin{cases} H_9=5 \\ [H_9|T_9]=[5,7] \end{cases} \Rightarrow$

$\Rightarrow \begin{cases} H_9=5 \\ H_9=5, T_9=[7] \end{cases}$



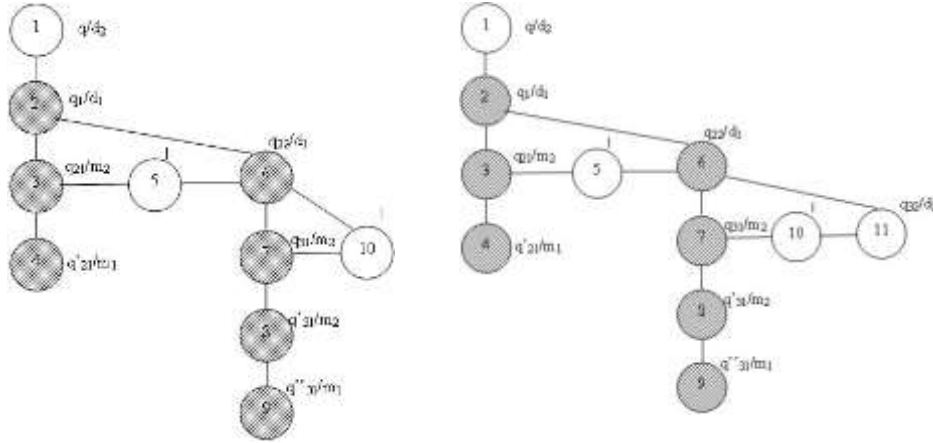
Deci q''_{31} adevărat și soluționat de (9). $\Rightarrow q'''3$

$q'''3: ?- !, \text{difference}(TA6, B6, U6) .$

$q'''3: ?- !, \text{difference}([6], [2, 3, 5, 7], U6) .$

$\leftarrow \text{-----} q_{32} \text{-----} \rightarrow$

(10) !=!



$q^{iv}3: ?\text{-difference}(TA6, B6, U6) .$

$q^{iv}3: ?\text{-difference}([6], [2, 3, 5, 7], U6) .$

←----- q_{32} -----→

$q_{32}?\text{-difference}([6], [2, 3, 5, 7], U6) .$

$q_{32}/d2$

$\text{difference}([6], [2, 3, 5, 7], U6) / \text{difference}([HA|TA], B, [HA|U])$

≡

$$(11) \quad \begin{cases} [HA_{11}|TA_{11}]=[6] \\ B_{11}=[2,3,5,7] \\ U_6=[HA_{11}|U_{11}] \end{cases} \Rightarrow \begin{cases} HA_{11}=6, TA_{11}=[] \\ B_{11}=[2,3,5,7] \\ U_6=[6|U_{11}] \end{cases}$$

$q_{32} \Rightarrow q'_{32} \text{ difference}(TA_{11}, B_{11}, U_{11}) \equiv$
 $\text{difference}([], [2, 3, 5, 7], U_{11}) .$

$\Rightarrow q^v3$

$q^v3: ?\text{-difference}([], [2, 3, 5, 7], U_{11}) .$

←----- q'_{32} -----→

$q'_{32}: ?\text{-difference}([], [2, 3, 5, 7], U_{11}) .$

$q'_{32}/d3 \equiv \text{difference}([], [2, 3, 5, 7], U_{11}) /$
 $\text{difference}([], L, []).$

$$(12) \quad \begin{cases} []=[] \\ L_{12}=[2,3,5,7] \\ U_{11}=[] \end{cases} \Rightarrow \begin{cases} HA_{11}=6, TA_{11}=[] \\ L_{12}=[2,3,5,7] \\ U_{11}=[] \end{cases}$$

Deci q'_{32} adevărat și soluționat de (12). $\Rightarrow q^v3$ devine întrebarea vidă, ceea ce înseamnă că s-a terminat execuția, iar răspunsul se obține din ecuațiile celor 12 sisteme:

$$\begin{array}{llll}
 \text{Result} & \stackrel{(1)}{=} [HA_1|U_1] & \stackrel{(1)}{=} [1|U_1] & \stackrel{(2)}{=} [1|U_2] & \stackrel{(6)}{=} [1|U_6] \\
 & \stackrel{(11)}{=} [1|[HA_{11}|U_{11}]] & \stackrel{(11)}{=} [1|[6|U_{11}]] & \text{rescriere} & \stackrel{(12)}{=} [1,6|[]] \\
 & \text{rescriere} & & & \\
 & = [1,6] & & &
 \end{array}$$

Result=[1,6].

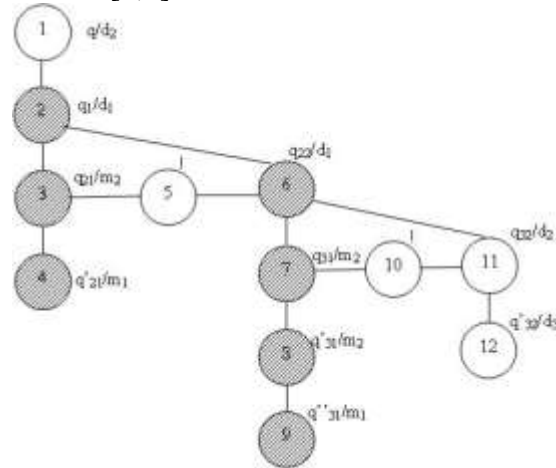


Fig. 3.5

În arborele de deducție din Fig. 3.5 sunt hașurate nodurile care nu fac backtracking (datorită !).

Trasarea execuției:

```

d(Result):-difference([1,3,5,6],[2,3,4,5,7],Result).
| ?- d(Result).
Result = [1,6] ? ;
no
| ?- trace.
% The debugger will first creep -- showing everything
(trace)
yes
% trace
| ?- d(Result).
1          1 Call: d(_473) ?

```



```

      2                               2      Call:
difference([1,3,5,6],[2,3,4,5,7],_473) ?
      3      3 Call: member(1,[2,3,4,5,7]) ?
      4      4 Call: member(1,[3,4,5,7]) ?
      5      5 Call: member(1,[4,5,7]) ?
      6      6 Call: member(1,[5,7]) ?
      7      7 Call: member(1,[7]) ?
      8      8 Call: member(1,[]) ?
      8      8 Fail: member(1,[]) ?
      7      7 Fail: member(1,[7]) ?
      6      6 Fail: member(1,[5,7]) ?
      5      5 Fail: member(1,[4,5,7]) ?
      4      4 Fail: member(1,[3,4,5,7]) ?
      3      3 Fail: member(1,[2,3,4,5,7]) ?
      9                               3      Call:
difference([3,5,6],[2,3,4,5,7],_1746) ?
      10      4 Call: member(3,[2,3,4,5,7]) ?
      11      5 Call: member(3,[3,4,5,7]) ?
?      11      5 Exit: member(3,[3,4,5,7]) ?
?      10      4 Exit: member(3,[2,3,4,5,7]) ?
      12                               4      Call:
difference([5,6],[2,3,4,5,7],_1746) ?
      13      5 Call: member(5,[2,3,4,5,7]) ?
      14      6 Call: member(5,[3,4,5,7]) ?
      15      7 Call: member(5,[4,5,7]) ?
      16      8 Call: member(5,[5,7]) ?
?      16      8 Exit: member(5,[5,7]) ?
?      15      7 Exit: member(5,[4,5,7]) ?
?      14      6 Exit: member(5,[3,4,5,7]) ?
?      13      5 Exit: member(5,[2,3,4,5,7]) ?
      17                               5      Call:
difference([6],[2,3,4,5,7],_1746) ?
      18      6 Call: member(6,[2,3,4,5,7]) ?
      19      7 Call: member(6,[3,4,5,7]) ?
      20      8 Call: member(6,[4,5,7]) ?
      21      9 Call: member(6,[5,7]) ?
      22     10 Call: member(6,[7]) ?
      23     11 Call: member(6,[]) ?
      23     11 Fail: member(6,[]) ?
      22     10 Fail: member(6,[7]) ?
      21      9 Fail: member(6,[5,7]) ?
      20      8 Fail: member(6,[4,5,7]) ?
      19      7 Fail: member(6,[3,4,5,7]) ?
      18      6 Fail: member(6,[2,3,4,5,7]) ?

```

```

24                                     6      Call:
difference([], [2,3,4,5,7], _11352) ?
24          6 Exit: difference([], [2,3,4,5,7], []) ?
17          5 Exit: difference([6], [2,3,4,5,7], [6])
?
12                                     4      Exit:
difference([5,6], [2,3,4,5,7], [6]) ?
9                                     3      Exit:
difference([3,5,6], [2,3,4,5,7], [6]) ?
2                                     2      Exit:
difference([1,3,5,6], [2,3,4,5,7], [1,6]) ?
1          1 Exit: d([1,6]) ?
Result = [1,6] ?
Yes

```

3.6 Exerciții și probleme propuse

1. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `member/2` și întrebarea
`| ?- member(c, [a,b,c,d]) .`

2. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `member/2` și aceeași întrebare, în condițiile în care prima clauză a predicatului devine:
`member(H, [H|T]) :- ! .`

3. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `member/2` și aceeași întrebare, în condițiile în care clauzele își schimbă poziția relativă (clauza 2 devine prima, iar prima devine a doua).

4. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `member/2` și aceeași întrebare, în condițiile în care clauzele își schimbă poziția relativă, iar faptul se rescrie sub forma:
`member(H, [H|T]) :- ! .`

5. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `member/2` și întrebarea
`| ?- member(X, [a,b,c,d]) .`

6. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `append/3` și întrebarea
`| ?- append([a,b,c],[d,e],Result).`

7. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `append/3` și aceeași întrebare, în condițiile în care prima clauză a predicatului devine:
`append([],L,L):-!.`

8. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `append/3` și aceeași întrebare, în condițiile în care clauzele își schimbă poziția relativă (clauza 2 devine prima, iar prima devine a doua).

9. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `append/3` și aceeași întrebare, în condițiile în care clauzele își schimbă poziția relativă, iar faptul se rescrie sub forma:
`append([],L,L):-!.`

10. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `union/3` și întrebarea:
`| ?- union([2,4,6],[1,3,4,5,6,7],R).`

11. Ce se întâmplă la repetarea întrebării de la exercițiul precedent?

12. Cum este afectată execuția de la precedentele două exerciții în cazul în care după apelul predicatului `member` din prima clauză se adaugă tăierea de backtracking, adică:

```
union([HA|TA],B,U):-
    member(HA,B),!,
    union(TA,B,U).
```

Desenați arborele de deducție precizând nodurile care sunt împiedicate să facă backtracking. Analizați care sunt nodurile care la repetarea întrebării se refac, specificând care este forma lor nouă.

13. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `union/3` și întrebarea:
`| ?- union([2,4,6],[1,3,4,5,6,7],R).`
 în condițiile în care faptul (clauza a treia) devine prima clauză a predicatului `union/3`. Comparați arborele de deducție cu cel obținut în forma inițială. Ce observații se pot face în legătură cu eficiența.

14. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `intersection/3` și întrebarea:

```
| ?- intersection([2,3,4],[1,3,4,5,6,7],R).
```

15. Ce se întâmplă la repetarea întrebării de la exercițiul precedent?

16. Desenați arborele de deducție de la precedentele două exerciții precizând nodurile care sunt împiedicate să facă backtracking. Analizați care sunt nodurile care la repetarea întrebării se refac, specificând care este forma lor nouă.

17. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `intersection/3` și întrebarea:

```
| ?- intersection([2,3,4],[1,3,4,5,6,7],R).
```

în condițiile în care faptul (clauza a treia) devine prima clauză a predicatului `intersection/3`. Comparați arborele de deducție cu cel obținut în forma inițială. Ce observații se pot face în legătură cu eficiența.

18. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `difference/3` și întrebarea:

```
| ?- difference([1,2,3,4],[2,4,5,6,7],R).
```

19. Ce se întâmplă la repetarea întrebării de la exercițiul precedent?

20. Desenați arborele de deducție de la precedentele două exerciții precizând nodurile care sunt împiedicate să facă backtracking. Analizați care sunt nodurile care la repetarea întrebării se refac, specificând care este forma lor nouă.

21. Trasați execuția scriind sistemele de ecuații cu instanțierile la nivelul fiecărui nod pentru predicatul `difference/3` și întrebarea:

```
| ?- difference([1,2,3,4],[2,4,5,6,7],R).
```

în condițiile în care faptul (clauza a treia) devine prima clauză a predicatului `difference/3`. Comparați arborele de deducție cu cel obținut în forma inițială. Ce observații se pot face în legătură cu eficiența.

22. Analizați asemănările și deosebirile (altele decât numele predicatelor) dintre `union/3` și `intersection/3`. Argumentați.

23. Analizați asemănările și deosebirile (altele decât numele predicatelor) dintre `union/3` și `difference/3`. Argumentați.

24. Analizați asemănările și deosebirile (altele decât numele predicatelor) dintre `intersection/3` și `difference/3`. Argumentați.

4. Operații pe liste

4.1 Apartenența la o listă

Așa cum deja am menționat în capitolele precedente, unul dintre predicatele de bază în manipularea listelor este `%member/2`. Am văzut în capitolul 2 definiția sa

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

Și câteva execuții, ca de exemplu:

```
?-member(2, [1,2,3]) .
yes
```

În toate exemplele de până acum predicatul a fost utilizat în sensul în care el a fost construit: să identifice existența/inexistența unui (anumit) element (precizat) într-o listă precizată. Dar, contrar altor limbaje de programare, nu am precizat nimic referitor la "fluxul" argumentelor: și anume dacă ele sunt de intrare sau intrare. În accepțiunea de mai sus a predicatului `member/2` ambele argumente sunt de ieșire. Este firesc: dorim să verificăm dacă un anumit element este prezent într-o anumită listă. Deși argumentele, în conformitate cu semantica lor declarativă, sunt a fi de intrare, vom demonstra în continuare că au diferite valențe. Presupunem că același predicat i se lansează întrebarea:

```
?- member(X, [1,2,3]) .
```

Surprinzător, sistemul "nu este surprins" de o astfel de întrebare. Ba mai mult, are soluții multiple, de forma:

```
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
no
```

Comportamentul prezentat mai sus este nedeterminist. Interpretarea este următoarea: X este membru în lista $[1, 2, 3]$ dacă $X = 1$ sau $X = 2$ sau X

= 3. Nedeterminismul este modelat în Prolog prin mecanismul implicit de backtracking. Cititorul este invitat să construiască cei trei arbori de execuție corespunzători celor trei valori posibile pentru X.

Atunci când este necesar Prolog "crează șablon" pentru datele structurate, utilizând dacă este cazul notația pentru element indiferent "_".

```
?-member(1,L).
L=[1|_]? ;

L=[_,1|_]? ;
...
```

Apelat ca mai sus, predicatul `member` returnează în principiu o infinitate de soluții creând șablon pentru variabila de ieșire, care aici este chiar lista la care se testează apartenența elementului "1".

În capitolele următoare vom vedea predicatul `member/2` în acțiune, cu ambele sale valențe, și cu o expresivitate surprinzătoare pentru o secvență atât de simplă.

4.2 Concatenarea listelor

O altă problemă frecvent necesar a fi rezolvată este cea de concatenare a două liste. Abordarea naturală, cea de "adăugare" a unei liste la sfârșitul celeilalte, face imediată identificarea soluției care necesită trei argumente: cele două liste de concatenat (argumente de "intrare") și respectiv lista "rezultat" (argument de ieșire). Soluția imediată este să parcurgem prima listă, și când ajungem la sfârșitul ei să poziționăm cea de-a doua listă.

```
%append/3
%append(prima lista de concatenat, a doua lista de
concatenat, lista rezultat)

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

Clauza a doua preia primul element din lista de intrare și-l depune la ieșire, urmând ca restul listei să fie construit prin apel recursiv, pe restul listei dată ca

prim argument și întreaga listă a doua. Clauza întâi, de terminare a recursivității, precizează că lista de ieșire este lista a doua de intrare în cazul în care prima listă de intrare este (sau ajunge prin recursivitate) listă vidă. Acest lucru se realizează prin unificare implicită, adică prin faptul că pentru două argumente diferite s-a folosit aceeași variabilă. În acest fel, variabila liberă (care nu este instanțiată la apel) primește valoarea variabilei legate (care este instanțiată la apel). În cazul nostru, parametrul 3 (argumentul de ieșire) primește ca valoare valoarea parametrului 2. Procesul este mai evident dacă clauza se scrie sub forma:

```
append([ ], L, R) :- L=R.
```

În această formă a clauzei, deși semantic reprezintă același lucru, este mai evident procesul de instanțiere a rezultatului.

La execuție, se obține:

```
?- append([1,2],[3,4],L).
L = [1,2,3,4] ? ;
no
```

Pentru un predicat există în general atâtea interpretări câte șabloane de intrare/ieșire sunt posibile. Desigur, nu toate interpretările unui predicat sunt interesante pentru o anumită aplicație, după cum unele pot fi de interes pur teoretic. Pentru predicatul "append" reținem însă utilizarea sa pentru concatenarea a două liste (prezentată mai sus). Acesta este un comportament "funcțional", în sensul că pentru date de intrare stabilite (două liste arbitrare) valoarea datelor de ieșire este unic definită (concatenarea celor două liste în ordinea în care au fost prezentate la apel). Dar așa cum am precizat anterior (la exemplul cu `member`) nu există un flux al argumentelor care trebuie specificat; mai mult, chiar atunci când un predicat este proiectat în ideea unui anumit flux al argumentelor, și alte variante pot să fie posibile. Un predicat stabilește însă în general o relație n -ară între argumente (o funcție fiind un caz particular de relație). O asemenea relație este construită cu ajutorul mecanismului de backtracking care permite ca o anumită întrebare să fie interpretată în general conform mai multor clauze din definiția unui predicat. O altă utilizare frecventă a predicatului "append" este pentru generarea "descompunerilor" unei liste. Să vedem ce se întâmplă dacă se apelează predicatul cu argumentele "pe flux invers" decât cel avut în vedere la proiectarea lui: adică primele două de ieșire și cel de-al treilea de intrare.


```
?- append(X,Y,[1,2,3,4]).
X = [],Y = [1,2,3,4] ? ;
X = [1],Y = [2,3,4] ? ;
X = [1,2],Y = [3,4] ? ;
X = [1,2,3],Y = [4] ? ;
X = [1,2,3,4],Y = [] ? ;
no
```

În acest caz caracterul nedeterminist al limbajului este pus în evidență, și, se "caută" toate variantele de liste a căror concatenare ar putea furniza lista $[1, 2, 3, 4]$. Soluțiile se generează la repetarea întrebării pe backtracking. Dialogul de mai sus sugerează utilizarea predicatului "append" pentru alegerea nedeterministă a unui element dintr-o listă astfel:

```
?-append(_,[E|_],[1,2,3,4]).
E=1? ;
E=2? ;
E=3? ;
E=4? ;
no
```

Ca și în cazul lui `member`, această capacitate va fi intens utilizată pentru o diversitate de probleme (unele de o complexitate ridicată) în capitolele următoare.

Ce se întâmplă dacă se apelează predicatul în manieră nedeterministă, dar ordinea clauzelor este inversată, adică:

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

Aceeași întrebare va obține aceleași rezultate, dar în ordine inversă, adică:

```
?- append(X,Y,[1,2,3,4]).
X = [1,2,3,4],Y = [] ? ;
X = [1,2,3],Y = [4] ? ;
X = [1,2],Y = [3,4] ? ;
X = [1],Y = [2,3,4] ? ;
X = [],Y = [1,2,3,4] ? ;
no
```

Cum se explică această ordonare?

Să vedem o utilizare imediată a acestui predicat, și anume concatenarea a trei liste. O variantă imediată ar fi:

```
%append3_1/4
%append3_1(prima lista de concatenat, a doua lista de
concatenat, a treia lista de concatenat, lista rezultat)

append3_1(L1,L2,L3,Result):-
    append(L1,L2,Int),
    append(Int,L3,Result).
```

Dar la fel de naturală (și corectă) ar fi și soluția:

```
%append3_2/4
%append3_2(prima lista de concatenat, a doua lista de
concatenat, a treia lista de concatenat, lista rezultat)

append3_2(L1,L2,L3,Result):-
    append(L2,L3,Int),
    append(L1,Int,Result).
```

Rezultatul este corect, în ambele cazuri lista L2 adăugându-se la sfârșitul lui L1, iar L3 la sfârșitul lui L2, adică pentru liste de forma:

$L1=[1,2,3,4]$, $L2=[5,6]$, $L3=[7,8,9]$ rezultatul va fi în ambele cazuri corect, $Result=[1,2,3,4,5,6,7,8,9]$. Dar care din variante e mai eficientă? Pentru a găsi răspuns la această întrebare vom face un raționament pe lungimile listelor. Presupunem $|L1|=l1$, $|L2|=l2$, $|L3|=l3$ și vom determina numărul de operații elementare (descompuneri după șablonul $([H|T])$ efectuate în `append/3`.

```
?-append3_1([1,2,3,4],[5,6],[7,8,9],Result).
```

are nevoie de $l1$ descompuneri pentru apelul `append(L1,L2,Int)` și construirea listei `Int`, care la rândul ei va avea lungimea $l1+l2$, deoarece va conține toate elementele din $L1$ și $L2$. Pentru apelul `append(Int,L3,Result)` vom avea nevoie de $l1+l2$ descompuneri, corespunzător lungimii listei aflată pe poziția primului parametru. În total avem: $l1 + (l1+l2) = 2 \cdot l1 + l2$ șabloane de aplicat pentru calcularea rezultatului (în fapt un număr dublu, dacă luăm în considerare faptul că la fiecare pas se

aplică un șablon de descompunere a unui argument, și unul de compunere a altui argument; vom neglija în mod intenționat acest aspect, având în vedere că afectează cu un factor de proporționalitate – 2- ambele variante).

?-append3_2([1,2,3,4],[5,6],[7,8,9],Result) .
 are nevoie de l_2 descompuneri pentru apelul `append(L2,L3,Int)` și construirea listei `Int`, care la rândul ei va avea lungimea l_2+l_3 , deoarece va conține toate elementele din `L2` și `L3`. Până aici totul este similar cu cazul anterior, și pare fără sens analiza efectuată; lucrurile se schimbă însă în acest moment. Pentru apelul `append(L1,Int,Result)` vom avea nevoie de un număr de descompuneri proporțional cu lungimea primului argument, adică l_1 . În total avem: $l_2 + l_1 = l_1 + l_2$ șabloane de aplicat pentru calcularea rezultatului

Deci:

?-append3_1([1,2,3,4],[5,6],[7,8,9],Result) .
 Result=[1,2,3,4,5,6,7,8,9] .

Și se aplică $2 \cdot l_1 + l_2$ șabloane,

?-append3_2([1,2,3,4],[5,6],[7,8,9],Result) .
 Result=[1,2,3,4,5,6,7,8,9] .

Și se aplică $l_1 + l_2$ șabloane.

Cum $2 \cdot l_1 + l_2 \geq l_1 + l_2$, pentru orice l_1, l_2, l_3 (valori pozitive, reprezintă lungimi de listă), rezultă imediat că `append3_2` e variantă mai eficientă necondiționat!

Vom arăta și practic acest lucru prin adăugarea unui contor care să memoreze numărul de șabloane aplicate pentru realizarea compunerii/descompunerii. Predicatul de bază de concatenare a 2 liste va avea deci un argument suplimentar, contorul.

```
%append_count/4
%append_count(prima lista de concatenat, a doua lista de
concatenat, lista rezultat, contorul)
```

```
append_count([],L,L,0) .
append_count([H|T],L,[H|R],Count) :-
    append_count(T,L,R,TCount),
    Count is TCount+1.
```

În această variantă, `append3_1` și `append3_2` se rescriu pentru a contoriza șabloanele:

```
%append3_1/5
%append3_1(prima lista de concatenat, a doua lista de
concatenat, a treia lista de concatenat, lista
rezultat,contorul)
```

```
append3_1(L1,L2,L3,Result):-
    append_count(L1,L2,Int Steps1),
    append_count(Int,L3,Result Steps2),
    Steps is Steps1 + Steps2.
```

și respectiv:

```
%append3_2/5
%append3_2(prima lista de concatenat, a doua lista de
concatenat, a treia lista de concatenat, lista
rezultat,contorul)
```

```
append3_2(L1,L2,L3,Result):-
    append_count(L2,L3,Int, Result, Steps1),
    append_count(L1,Int,Steps2),
    Steps is Steps1 + Steps2.
```

Cu rulările:

```
?- append3_1([1,2,3], [4,5,6,7], [8,9], Res, Steps).
Res = [1,2,3,4,5,6,7,8,9],
Steps = 10 ?
Și
?- append3_2([1,2,3], [4,5,6,7], [8,9], Res, Steps).
Res = [1,2,3,4,5,6,7,8,9],
Steps = 7 ?
```

Să vedem cum se poate scrie append3 ca predicat de sine stătător.

```
%append3_3/4
%append3_3(prima lista de concatenat, a doua lista de
concatenat, a treia lista de concatenat, lista rezultat)
```

```
append3_3([],[],L,L).
append3_3([], [H|T],L, [H|R]):-
    append3_3([],T,L,R).
append3_3([H|T],L1,L2, [H|R]):-
    append3_3(T,L1,L2,R).
```

La execuție, rezultatul va fi cel așteptat, adică:

```
?-append3_3([1,2,3,4],[5,6],[7,8,9],Result).
Result=[1,2,3,4,5,6,7,8,9].
```

În continuare se prezintă alte câteva predicate simple pe liste, a căror soluție nu necesită o tehnică particulară, dar care utilizează unele din predicatele descrise până în acest moment.

4.3 Eliminarea duplicatelor

Eliminarea duplicatelor dintr-o listă:

```
%remove_dupl/2
%remove_dupl(intrare, iesire)

remove_dupl([], []).
remove_dupl([H|T], R) :-
    member(H,T), !,
    remove_dupl(T,R).
remove_dupl([H|T], [H|R]) :-
    remove_dupl(T,R).
```

Care dintre dubluri rămâne în listă? Urmărind predicatul se observă că ultima apariție este cea care se păstrează. Un apel ne demonstrează acest lucru:

```
?- remove_dupl([3,2,5,3,6,7,2,6,2], Res).
Res = [5,3,7,6,2] ?
```

Cum trebuie modificat predicatul pentru păstrarea primei apariții?
Simpla contorizare a elementelor distincte în listă se realizează cu:

```
%count_dist/2
%count_dist(lista, contor)

count_dist([],0).
count_dist([H|T], R) :-
    member(H,T), !,
    count_dist(T,R).
count_dist([_|T], R1) :-
    count_dist(T,R),
    R1 is R+1.
```

```
?- count_dist([3,2,1,5,4,2,7,4,2],C) .
C = 6 ?
```

4.4 Ștergerea unui element dintr-o listă

Un predicat care șterge un element specificat dintr-o listă parcurge lista până la identificarea elementului, refăcând ieșirea din toate elementele, cu excepția celui de șters.

```
%delete/3
%delete(element de sters, lista din care se sterge,
lista rezultat)

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

Și execuția returnează rezultatul așteptat:

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

Este această soluție unică? Evident nu. Câte soluții are problema? Se pot obține și celelalte soluții? În ce ordine?

```
| ?- delete(1, [1,2,1,3,1], R) .
R = [2,1,3,1] ? ;
R = [1,2,3,1] ? ;
R = [1,2,1,3] ? ;
R = [1,2,1,3,1] ? ;
no
```

Se observă că în situația în care întrebarea se repetă succesiv până la obținerea răspunsului `no`, ultima soluție returnată este chiar lista originală. Cum explicați? De ce se obțin toate aceste soluții? Datorită caracterului nedeterminist al predicatului; deși la execuția inițială elementul de șters este identificat ca fiind primul din lista de intrare (matching reușit pe clauza 1, care încheie cu succes execuția, returnând o soluție), la repetarea întrebării, pe backtracking, deși elementul de șters (`x`) este identic cu elementul din

capul listei (H), regula de calcul Prolog specifică reîncercarea suprapunerii celei mai recente întrebări peste clauza următoare (a 2a în cazul nostru), suprapunere cu succes. Chiar dacă cei doi termeni (X și H) sunt egali și s-ar unifica, clauza 2 neglijează intenționat acest lucru, menținând elementul în rezultat (compunerea $[H|R]$ din argumentul al 3lea). Se avansează în lista de intrare până la următorul (dacă mai este vreunul) element care poate fi șters. Se observă că în exemplul ales, în listă există elementul de 3 ori, deci se obțin 4 rezultate distincte: 3 care elimină câte o apariție a elementului precizat (de la stânga la dreapta), și ultima în care nici o apariție nu e eliminată.

Pe de altă parte, se pune întrebarea care este rezultatul returnat în cazul în care elementul de șters nu se află în listă? Această situație corespunde ultimei soluții din exemplul precedent (repetarea întrebării până când practic nu se mai face nici o eliminare). Deci în această situație, se returnează direct lista originală, fără să se șteargă nimic.

```
| ?- delete(5,[1,2,1,3,1],R) .
R = [1,2,1,3,1] ? ;
no
```

Evident problema a fost gândită pentru situația (firească) în care se precizează expres elementul care să se șteargă din listă. Ce se întâmplă însă în situația în care elementul de șters nu este precizat (este variabilă liberă; în loc să fie argument de intrare este unul de ieșire).

```
| ?- delete(X,[1,2,3,4],R) .
X=1, R = [2,3,4], ? ;
X=2, R = [1,3,4], ? ;
X=3, R = [1,2,4], ? ;
X=4, R = [1,2,3], ? ;
no
```

Comportamentul nedeterminist este și mai pregnant în această situație; nu doar că șterge o anumită instanță a unui element, dar realizează simultan și selecția elementului de șters. Acest comportament nedeterminist va fi valorificat în capitolele următoare.

Dacă vrem să conferim caracter determinist predicatului, nu trebuie decât să precizăm faptul că, odată ce un element a fost identificat ca fiind cel de eliminat, ștergerea lui este irevocabilă (fără posibilitatea de revenire).

```

delete(X, [X|T], T) :- !
delete(X, [H|T], [H|R]) :-
    delete(X, T, R) .
delete(_, [ ], [ ]) .

```

Tăierea de backtracking din clauza 1 are semantica: dacă elementul de șters este chiar primul element al listei de intrare, instanțiază restul listei de ieșire la restul listei de intrare (T) și nu mai încerca NICI o altă variantă. Cu alte cuvinte, atunci când are loc o suprapunere cu succes peste o clauză care conține tăierea de backtracking, și aceasta devine efectivă (se trece peste !), este ca și cum nu ar mai exista nici o clauză după acea clauză (la execuția cu succes a clauzei 1, clauzele 2 și 3 dispar).

Pentru o astfel de implementare, se obține rezultatul:

```

| ?- delete(1, [1, 2, 1, 3, 1], R) .
R = [2, 1, 3, 1] ? ;
no

```

iar pentru

```

| ?- delete(X, [1, 2, 3, 4], R) .
X=1, R = [2, 3, 4], ? ;
no

```

Dacă se dorește eliminarea tuturor aparițiilor unui element dintr-o listă, avem:

```

%delete_all/3
%delete_all(element de sters, lista din care se sterge,
lista rezultat)

```

```

delete_all(X, [X|T], R) :-
    delete_all(X, T, R) .
delete_all(X, [H|T], [H|R]) :-
    delete_all(X, T, R) .
delete_all(_, [ ], [ ]) .

```

```

| ?- delete_all(1, [1, 2, 1, 3, 1], R) .
R = [2, 3]

```


4.5 Înlocuirea unui element într-o listă

Dacă se dorește înlocuirea unui element cu un alt element:

```
%subst/4
%subst(      element de înlocuit,
           element cu care se înlocuiește,
           lista în care se face înlocuirea,
           lista rezultat)

subst(H,X,[H|T],[X|T]).
subst(Y,X,[H|T],[H|R]):-
    subst(Y,X,T,R).
subst(_,_,[ ],[ ]).

| ?- subst(1,4,[1,2,1,3,1],R).
R = [4,2,1,3,1] ? ;
R = [1,2,4,3,1] ? ;
R = [1,2,1,3,4] ? ;
R = [1,2,1,3,1] ? ;
no
```

Varianta deterministă a predicatului este:

```
subst(H,X,[H|T],[X|T]) :-!
subst(Y,X,[H|T],[H|R]):-
    subst(Y,X,T,R).
subst(_,_,[ ],[ ]).
```

Iar varianta care substituie toate aparițiile unui element cu un alt element:

```
%subst_all/4
%subst_all(      element de înlocuit,
                element cu care se înlocuiește,
                lista în care se face înlocuirea,
                lista rezultat)

subst_all(H,X,[H|T],[X|R]) :-
    subst_all(H,X,T,R).
subst_all(Y,X,[H|T],[H|R]):-
    subst_all(Y,X,T,R).
subst_all(_,_,[ ],[ ]).
```

4.6 Lungimea unei liste

Vom implementa în cele ce urmează un predicat care calculează lungimea unei liste (deși programarea logică este una specifică calculului simbolic, există situații în care calcule numerice sunt necesare). O abordare naturală este cea prin care se aplică șabloane de descompunere a listei, și se contorizează numărul de șabloane aplicate. Soluția ar fi:

```
%list_length_1/2
%list_length_1(lista, lungime lista)

list_length_1([],0).
list_length_1([H|T],Length):-
    list_length_1(T,TailLength),
    Length is TailLength + 1.
```

Clauza recursivă (a doua) aplică șablonul de descompunere a listei căreia i se calculează lungimea; apelul recursiv din corpul clauzei calculează lungimea cozii listei TailLength, după care, incrementarea Length is TailLength + 1 corespunde chiar numărării șablonului aplicat. De câte ori se aplică șablonul, de atâtea ori se incrementează o valoare ce pornește de la 0 (lista vidă are lungimea 0). De remarcat faptul că lungimea finală is TailLength + 1 și nu = TailLength + 1 deoarece se cere evaluarea expresiei din stânga, și nu expresia însăși). La execuție se obține:

```
?- list_length_1([1,2,3,4,5,6,7],Number).
Number = 7? ;
no.
```

Datorită faptului că evaluarea se face după apelul recursiv, și deci calculele se efectuează pentru porțiuni dinspre sfârșitul spre începutul listei, spunem că avem recursivitate înapoi. Pentru o mai bună înțelegere a fenomenului, rescriem predicatul, cu tipărirea lungimilor porțiunilor de listă, pe măsură ce ele se calculează.

```
%list_length_1_w /2
%list_length_1_w (lista, lungime lista)

list_length_1_w([],0).
list_length_1_w([H|T],Length):-
```

```

list_length_1_w(T,TailLength),
write(T),write(' '), // tipărește lista
write(TailLength),nl, //tipărește
                        //lungimea listei

Length is TailLength + 1.

% run_list_length_1_w /2
% run_list_length_1_w (lista, lungime lista)

run_list_length_1_w(List,Length):- //apelul tipăririlor
    write('T='),write(' '),
    write('TailLength='),nl,
    list_length_1_w(List,Length),
    write(List),write(' '),
    write(Length),nl.

```

La execuție se obține:

```

?- run_list_length_1_w([1,2,3,4,5,6,7],Length).
T=      TailLength=
[]      0
[7]     1
[6,7]   2
[5,6,7] 3
[4,5,6,7] 4
[3,4,5,6,7] 5
[2,3,4,5,6,7] 6
[1,2,3,4,5,6,7] 7
Length = 7 ? ;
no

```

Dacă utilizatorul este interesat de lungimile porțiunilor de listă dinspre începutul listei, o altă abordare este necesară. Și anume, incrementarea trebuie făcută imediat la aplicarea șablonului, și abia ulterior avansul recursiv.

```

%list_length_2/2
%list_length_2(lista, lungime lista)

list_length_2([],Length).
list_length_2([H|T],PartialLength):-
    NewPartialLength is PartialLength +1,
    list_length_2(T,NewPartialLength).

```

Predicatul al doilea este o variantă ușor modificată a primului, și anume: incrementarea se aplică înainte de apelul recursiv (**imediat** ce aplicăm șablonul contorizăm operația), iar la atingerea listei vide, variabila va conține valoarea lungimii listei (și nu 0, deoarece se merge pe recursivitate înainte).

Să vedem ce se întâmplă la execuție. De menționat că apelul trebuie făcut prin inițializarea la 0 a lungimii (setăm contorul, înainte de aplicarea primului șablon).

```
?- list_length_2([1,2,3,4,5,6,7],0).
yes;
no.
```

Surprinzător! Interpretarea? *yes* ne spune că s-a terminat execuția cu succes, calculându-se lungimea listei. *no* spune că la repetarea întrebării nu există alt răspuns (iarăși răspuns așteptat predicatul are comportament determinist, având o singură soluție). Atunci de ce nu avem valoarea lungimii? S-a calculat ea? Rescriem predicatul, adăugând o tipărire a variabilei *Length* când se ajunge la lista vidă.

```
list_length_2([],Length):-
    write(' Length ='),
    write(Length),nl.
list_length_2([H|T],PartialLength):-
    NewPartialLength is PartialLength +1,
    list_length_2(T,NewPartialLength).
```

Acum apelul va furniza:

```
?- list_length_2([1,2,3,4,5,6,7],0).
Length = 7
yes;
no.
```

Deci într-adevăr, predicatul calculează corect lungimea, și totuși nu o returnează. De ce? Răspunsul tine de specificitatea programării logice: argumentul lungime a fost inițializat la 0. Deci nu există o variabilă în care să returneze rezultatul. Argumentele pot fi în mod nedeterminist de intrare sau de ieșire, dar nu pot fi la aceeași execuție de intrare/ieșire. O variabilă logică odată instanțiată nu-și mai schimbă valoarea, decât pe backtracking. Deci lungimea, inițializată la 0, rămâne la 0. Lungimea diferitelor secvențe există, pentru că ele se calculează; dar se păstrează în niște variabile intermediare

(NewPartialLength) care se pierde la terminarea execuției (similare variabilelor locale ale funcțiilor din programarea imperativă, variabile a căror valoare nu se mai cunoaște la terminarea execuției funcției). Valoarea tipărită de `write(' Length ='), write(Length), nl` este chiar valoarea unei astfel de variabile locale.

Pentru a captura valoarea acelei variabile trebuie să transmitem de-a lungul întregii execuții o variabilă liberă (neinstantiată) care să se instanțieze la sfârșitul execuției cu lungimea parțială a întregii liste, care este lungimea căutată. Modificarea necesită adăugarea unui nou parametru, cu semantica de lungimea întregii liste, în timp ce semantica parametrului existent este de lungimea listei deja parcurse, și deci este un parametru de acumulare care la sfârșitul execuției are semantica lungimea listei parcurse de la primul la ultimul element, deci lungimea întregii liste, și deci parametrul de acumulare devine în acel moment rezultat final, pe care îl instanțiază (de la primul element până la elementul curent).

```
%list_length_2/3
%list_length_2(lista, lungime lista, lungime portiune
parcursa de lista)

list_length_2([],Length, PartialLength):-
    Length=PartialLength.
list_length_2([H|T],Length,PartialLength):-
    NewPartialLength is PartialLength +1,
    list_length_2(T,Length,NewPartialLength) .
```

Se observă că față de varianta anterioară modificarea constă în adăugarea unui nou parametru (pe poziția argumentului 2), și instanțierea acestuia explicită la valoarea parametrului de acumulare în momentul ajungerii la lista vidă (cu semantica nu mai sunt elemente de contorizat, deci valoarea curentă a contorului reprezintă lungimea întregii liste). Dacă se înlocuiește instanțierea explicită cu cea implicită (folosirea aceluiași nume de variabilă pe pozițiile argumentelor de unificat) atunci forma predicatului este:

```
list_length_2([],Length,Length) .
list_length_2([H|T],Length,PartialLength):-
    NewPartialLength is PartialLength +1,
    list_length_2(T,Length,NewPartialLength) .
```

Indiferent de formă, datorită faptului că parametrul de acumulare trebuie inițializat la apel cu lista vidă, este bine să se utilizeze un alt predicat care face această inițializare, și deci:

```
run_list_length_2(List,                                Length):-
list_length_2(List,Length,0).
```

Ca și în cazul `list_length_2/2` rescriem predicatul, cu tipărirea lungimilor porțiunilor de listă, pe măsură ce acestea se calculează.

```
%list_length_2_w /2
%list_length_2_w (lista, lungime lista, lungime lista
parcursa)

list_length_2_w([],Length,Length):-
    write('[]'),                                //tipărește lista vidă
    write(' '),
    write(Length),nl.                            //tipărește lungimea listei vide
list_length_2_w([H|T],Length,PartialLength):-
    write([H|T]),write(' '),                    //tipărește lista curentă
    write(PartialLength),nl,                    //tipărește lungimea listei curente
    NewPartialLength is PartialLength +1,
    list_length_2_w(T,Length,NewPartialLength).

% run_list_length_1_w /2
% run_list_length_1_w (lista, lungime lista)

run_list_length_2_w(List, Length):-
    write('[H|T]='),write(' '),
    write('PartialLength='),nl,
    list_length_2_w(List,Length,0).
```

Execuția care urmează întrebării:

```
?- run_list_length_2_w([1,2,3,4,5,6,7],Length).
[H|T]=          PartialLength=
[1,2,3,4,5,6,7]          0
[2,3,4,5,6,7]           1
[3,4,5,6,7]             2
[4,5,6,7]               3
[5,6,7]                 4
[6,7]                   5
[7]                     6
[]                      7
```

Length = 7 ? ;
no

De remarcat diferențele între cele două variante. Să le punem în paralel:

Recursivitatea înainte			Recursivitatea înapoi		
[H T]=	PartialLength=	T=	TailLength=	Total=	
[1,2,3,4,5,6,7]	0	[1,2,3,4,5,6,7]	7	7	
[2,3,4,5,6,7]	1	[2,3,4,5,6,7]	6	7	
[3,4,5,6,7]	2	[3,4,5,6,7]	5	7	
[4,5,6,7]	3	[4,5,6,7]	4	7	
[5,6,7]	4	[5,6,7]	3	7	
[6,7]	5	[6,7]	2	7	
[7]	6	[7]	1	7	
[]	7	[]	0	7	

Length = 7 ? ;	Length = 7 ? ;
no	no

Acum devine evident cum în cazul recursivității înainte (calculul înainte apelului recursiv) lungimile reprezintă porțiuni de la începutul listei până la poziția curentă (de exemplu pentru lista rămasă [4,5,6,7] parametrul de acumulare a ajuns la 3, specificând faptul că am trecut de 3 elemente înainte de elementul 4, și deci am aplicat până acum de 3 ori șablonul [H|T]), în timp ce în cazul recursivității înapoi (apelul recursiv înaintea calculului) lungimile reprezintă numărul de elemente de la poziția curentă până la sfârșitul listei (pentru același exemplu lungimea este 4). În permanență suma valorilor determinate prin cele două metode este constantă, lungimea întregii liste (în fapt calculează același lucru, pornind de la extremități diferite), așa cum se poate observa în coloana 5. Pe de altă parte trebuie menționat că în cazul recursivității înapoi rezultatele se abțin în ordine inversă decât ordinea menționată în coloanele 3 și 4 mai sus.

4.7 Tipuri de recursivitate

O altă operație necesară pe liste este cea de calculare a sumei tuturor elementelor unei liste. Este absolut evident că abordarea poate fi identică cu cea din problema precedentă (parcurea recursivă, cu incrementarea unui parametru care acumulează valorile), cu modificare minoră asupra operației făcute înafara apelului recursiv (adăugarea cumulativă a valorii elementului).

De asemenea, sunt valabile cele două abordări, pentru care dăm implementările, fără a intra în detalii.

Recursivitatea înainte conduce la implementarea:

```
% sum_1/2
%sum_1(lista de intrare,suma valorilor elementelor din
argumentul1).

sum_1([],0).
sum_1([H|T],Sum):-
    sum_1(T,TailSum),
    Sum is TailSum + H.
```

În timp ce recursivitatea înainte necesită un argument în plus, variabila rezultat instanțiind-o prin unificare implicită în condiția de terminare:

```
% sum_2/2
%sum_2(lista de intrare,rezultat, parametru de acumulare
%reprezentând suma valorilor elementelor parcurse din
argumentul1).

sum_2([],PartialSum,PartialSum).
sum_2([H|T],Sum,PartialSum):-
    NewPartialSum is PartialSum + H,
    sum_2(T,Sum,NewPartialSum).
```

Această variantă necesitând inițializarea parametrului de acumulare, apelul se face prin intermediul unui alt predicat:

```
run_sum_2(List,Sum):-sum_2(List,Sum,0).
```

Pentru a pune în evidență mai bine deosebirile între cele două metode, refacem traseul urmat la problema anterioară, tipărind valorile intermediare, pe măsură ce acestea se calculează. Implementările, împreună cu rulările corespunzătoare sunt date în continuare:

```
%sum_1_w /2
%sum_1_w (lista de intrare,suma valorilor elementelor din
argumentul1).

sum_1_w([],0).
sum_1_w([H|T],Sum):-
```



```

        sum_1_w(T,TailSum),
        write(T),write('      '),
        write(TailSum),nl,
        Sum is TailSum + H.

%run_sum_1_w /2
% run_sum_1_w (lista de intrare,suma valorilor elementelor
din %argumentul1).

run_sum_1_w(List,Sum):-
    write('Tail='),write('      '),
    write('TailSum='),nl,
    sum_1_w(List,Sum),
    write(List),write('      '),
    write(Sum),nl.

?- run_sum_1_w([1,2,3,4,5,6,7],Rezult).
Tail=      TailSum=
[]          0
[7]         7
[6,7]       13
[5,6,7]     18
[4,5,6,7]   22
[3,4,5,6,7] 25
[2,3,4,5,6,7] 27
[1,2,3,4,5,6,7] 28
Rezult = 28 ? ;
no.

```

și respectiv

```

%sum_2_w /2
%sum_2_w (lista de intrare,rezultat, parametru de
acumulare %reprezentând suma valorilor elementelor parcurse
din argumentul1).

sum_2_w([],PartialSum,PartialSum):-
    write('[]'),write('      '),
    write(PartialSum),nl.
sum_2_w([H|T],Sum,PartialSum):-
    NewPartialSum is PartialSum + H,
    write([H|T]),write('      '),
    write(PartialSum),nl,
    sum_2_w(T,Sum,NewPartialSum).

```

```
%run_sum_2_w /2
% run_sum_2_w (lista de intrare,suma valorilor elementelor
din %argumentul1).
```

```
run_sum_2_w(List,Sum):-
    write('List='),write(' '),
    write('PartialSum='),nl,
    sum_2_w(List,Sum,0).

?- run_sum_2_w([1,2,3,4,5,6,7],Rezult).
List=          PartialSum=
[1,2,3,4,5,6,7]      0
[2,3,4,5,6,7]        1
[3,4,5,6,7]          3
[4,5,6,7]             6
[5,6,7]              10
[6,7]                15
[7]                  21
[]                   28
Rezult = 28 ? ;
no.
```

Punând din nou în paralel rezultatele celor 2 metode:

Recursivitatea înainte		Recursivitatea înapoi	
List=	PartialSum=	Tail=	TailSum=
[1,2,3,4,5,6,7]	0	[1,2,3,4,5,6,7]	28
[2,3,4,5,6,7]	1	[2,3,4,5,6,7]	27
[3,4,5,6,7]	3	[3,4,5,6,7]	25
[4,5,6,7]	6	[4,5,6,7]	22
[5,6,7]	10	[5,6,7]	18
[6,7]	15	[6,7]	13
[7]	21	[7]	7
[]	28	[]	0
Rezult = 28 ? ;		Rezult = 28 ?	
no.		no.	

Aceeași mențiune trebuie făcută ca și în cazul lungimii listelor, și anume că în cazul recursivității înapoi rezultatele se obțin în ordine inversă decât ordinea menționată în coloanele 3 și 4 mai sus (adică primul obținut e pentru lista vidă, respectiv ultimul pentru întreaga listă).

Din cele două exemple (lungimea listei și respectiv suma elementelor unei liste) rezultă că principal, orice problemă care are o abordare recursivă poate fi soluționată prin oricare din cele două tehnici, pe care le descriem generic.

```
%forward_recursion/3
%forward_recursion(input argument, final result, partial
result)

forward_recursion([],PartialResult,PartialResult).
forward_recursion([H|T],Result,PartialResult):-
    do(NewPartialResult,H,PartialResult),
    forward_recursion(T,Result,NewPartialResult)

%forward_recursion_call/2
%forward_recursion_call(intrare, iesire)

forward_recursion_call(Input,Output):-
    forward_recursion(Input,Output,InitialValueOfResult)

%backward_recursion/2
%backward_recursion(input argument, output result)

backward_recursion([],InitialValue).
backward_recursion([H|T],PartialResult):-
    backward_recursion(T,NewPartialResult),
    do(PartialResult,H,NewPartialResult).
```

Unde `do/3` realizează operația specifică problemei (incrementarea, respectiv însumarea pentru problemele anterioare).

Analizând cele două metode generice, putem trage următoarele concluzii:

- recursivitatea înainte are avantajul prelucrării informației (și deci efectuării calculelor) pe măsură ce se avansează în structura de date. Aceasta înseamnă că putem obține ca și rezultate intermediare valoarea specifică a structurii de la începutul ei până la poziția curentă. Mai mult, această valoare este cunoscută (și poate fi făcută disponibilă) la momentul prelucrării elementului curent (în inordine). Valoarea poate fi folosită de un al proces care rulează concurent cu procesul care implementează recursivitatea înainte, în cazul prelucrării într-un limbaj logic paralel sau concurent;
- avantajul *last call optimization*;

- recursivitatea înainte are dezavantajul necesitării unui parametru suplimentar care să consemneze valoarea parametrului de acumulare la sfârșitul execuției;
- recursivitatea înainte are dezavantajul necesitării unui apel inițial specializat, în care parametrului de acumulare i se furnizează valoarea inițială. De aceea, o bună practică este aceea de a scrie un alt predicat care realizează apelul, care să nu facă altceva decât să realizeze inițializarea aceluia parametru, de forma:

```
forward_recursion_call(Input,Output):-
    forward_recursion(Input,Output,InitialValueOfResult)
```

- recursivitatea înapoi dimpotrivă, pune la dispoziție rezultatele în postordine, pe substructuri aflate între elementul curent și ultimul element al structurii prelucrate. Nu necesită inițializarea nici unui parametru, și nici un apel inițial specific. Neajunsul este că, procesele rulând concurrent, trebuie să se sincronizeze pe terminarea apelului `backward_recursion` (adică sunt întârziate, neputând folosi rezultate intermediare, decât în cazul expres când acestea trebuie furnizate pe substructuri aflate între elementul curent și ultimul element).

După cum era de așteptat fiecare dintre mecanisme prezintă avantaje și dezavantaje specifice. Decizia de implementare trebuie luată în funcție de specificitatea problemei.

Vom exemplifica tehnica pe încă o problemă. Aceasta ne va ajuta să exersăm elementele subliniate de observațiile anterioare, dar și să tragem concluzii noi.

4.8 Inversarea unei liste

Pentru a se obține lista inversă, argumentul de intrare trebuie descompus după șablonul `[Head|Tail]`. Dacă presupunem că predicatul a fost deja scris, atunci inversa cozii listei se obține simplu prin apelul recursiv al aceluiași predicat. Tot ce avem de făcut apoi este să adăugăm primul element al listei la sfârșitul cozii inversate. Această operație se poate realiza cu ajutorul predicatului `append/3`, primul argument fiind inversa cozii listei, al doilea argument lista conținând un singur element, capul listei de intrare (`[Head]`), iar al treilea rezultatul (inversa întregii liste).

```
%reverse1/2
%reverse1(lista de intrare, lista inversata).

reverse1([Head|Tail],ReversedList):-
    reveverse1(Tail,ReversedTail),
    append(ReversedTail,[Head],ReversedList.
reverse1([],[]).
```

O execuție furnizează rezultatul:

```
?- reverse1([1,2,3],Result).
yes.
```

Dacă se tipăresc rezultatele intermediare:

```
?- reverse1([1,2,3],Result).
lista partiala inversata este=[]
lista partiala inversata este=[3]
lista partiala inversata este=[3,2]
lista partiala inversata este=[3,2,1]
Result=[3,2,1] ;
no
```

Analizând problema observăm că, exact ca și celelalte de până acum, varianta recursivității înainte este posibilă și de această dată, cu prețul unui argument suplimentar, parametrul de acumulare.

```
%reverse2/3
%reverse2( lista de intrare,
           parametrul de acumulare,
           lista inversata).

reverse2([Head|Tail],PlaceholderforResult,Reversed):-
    Reveverse2(Tail,[Head|Reversed]).
reverse2([],Reversed,Reversed).
```

Datorită parametrului de acumulare avem nevoie de un alt predicat care să realizeze apelul:

```
reverse2(List,Reversed):- reverse2(List,[],Reversed).
```

O execuție va furniza (așa cum este de așteptat) rezultat identic cu cel al primei implementări:

```
?- reverse2([1,2,3],Result).
yes.
```

În timp ce tipărirea rezultatelor intermediare:

```
?- reverse1([1,2,3],Result).
lista partiala inversata este=[]
lista partiala inversata este=[1]
lista partiala inversata este=[2,1]
lista partiala inversata este=[3,2,1]
Result=[3,2,1] ;
no
```

Observați cum parametrul de acumulare acționează ca o stivă: valoarea curentă se adaugă înaintea tuturor valorilor existente până la pasul curent. Analizați particularitățile celor două tehnici pe acest exemplu.

Cea de-a doua implementare a inversării unei liste ne permite să facem o analiză mai amănunțită, și să tragem câteva concluzii legate de compunerea/descompunerea argumentelor în Prolog. Pornim de la secvențele de cod ale predicatelor `append/3` și `reverse/3` pe care pentru claritate le reluăm mai jos.

```
append([],List,List).
append([Head|Tail],List,[Head|Rest]):-
    append(Tail,List,Rest).
```

```
reverse([Head|Tail],PlaceholderforResult,Reversed):-
    reveverse (Tail,[Head|Reversed]).
reverse([],Reversed,Reversed).
```

Dacă facem o analiză comparativă a celor două predicate, și facem abstracție de numele lor și a parametrilor, vom observa ca practic nu este o mare deosebire între ele. Pentru a dovedi acest lucru, vom retranscrie cele două predicate asignându-le același nume de predicat, și aceleași nume pentru argumentele de pe poziții corespondente.

În urma acestei transformări predicatul append devine:

```
predicate([], List, List).
predicate([Head|Tail], List, [Head|Rest]) :-
    predicate(Tail, List, Rest).
```

Aceeași transformare aplicată predicatului de inversare a listei, și inversarea ordinii clauzelor, conduce la obținerea:

```
predicate([], List, List).
predicate([Head|Tail], List, Rest) :-
    predicate(Tail, List, [Head|Rest]).
```

Acum dacă facem o suprapunere a celor două variante vom observa că singura diferență care există între cele două este poziția șablonului de construcție a celui de-al treilea argument. Și anume, în prima variantă a predicatului șablonul este poziționat în capul regulii în timp ce în al doilea predicat, în apelul recursiv. Aceasta modificare "minoră" are implicații majore asupra rezultatului.

Apelul primului conduce la o execuție având un rezultat de forma:

```
?-predicate([1,2],[3,4],Result).
Result=[1,2,3,4].
```

În timp ce apelul celui de-al doilea (aduceți-vă aminte că semnificația celui de-al doilea argument la inversarea unei liste este de parametru de acumulare, ceea ce înseamnă că apelul inițial necesită instanțierea acestui argument la lista vidă):

```
?-predicate([1,2,3,4],[],Result).
Result=[4,3,2,1].
```

De unde această diferență majoră? Exclusiv din poziția șablonului în argumentul "de ieșire" (al treilea). Să încercăm o analiză bazată pe evoluția argumentelor, ca lungime (număr de elemente din liste).

Să presupunem că pentru primul predicat (în fapt append) primele două argumente (considerate de intrare în modelul în discuție) au lungimile l_1 și respectiv l_2 . În mod evident ne așteptăm ca rezultatul (al treilea argument) să aibă lungimea $l_3 = l_1 + l_2$ (ceea ce se și întâmplă dacă analizăm rezultatul de mai sus). Pe de altă parte este și ceea ce ne așteptăm, având în vedere

scopul în care a fost construit predicatul: reprezintă concatenarea a două liste, și deci, toate elementele din argumentele 1 și 2 vor trebui să fie regăsite în argumentul 3, și ca atare, lungimea celui din urmă trebuie să însumeze lungimile primelor 2. Dacă trecem la descompunerea primului argument în primul element și restul listei, în mod evident restul listei (Tail) va avea lungimea $l1-1$. Simetric, dacă argumentul 3 are lungimea $l1+l2$, înseamnă că lungimea lui `[Head|Rest]` este $l1+l2$, și deci `Rest` are lungimea $l1+l2-1$. În mod corespunzător, apelul recursiv se realizează pe liste cu 1 mai scurte (prima intrare și ieșirea). Adică, dacă în capul regulii, lungimile listelor furnizate ca și argumente sunt $l1$, $l2$ și respectiv $l3=l1+l2$, în apelul recursiv, argumentele au lungimi: $l1-1$, $l2$ și respectiv $l3-1=l1-1+l2$, ceea ce înseamnă că proporția se păstrează. Această afirmație ne permite să facem observația mai generală că, atunci când în construcția unei liste rezultat șablonul se aplică în capul regulii, lista rezultat are aceeași monotonie (în termeni de lungime a listei) cu argumentul de intrare (adică crește sau scade deodată cu argumentul de intrare).

Aceeași observație repetată pentru cel de-al doilea predicat ne conduce la concluzia următoare: argumentul al treilea (de ieșire) crește pe măsură ce primul argument (de intrare) descrește. Creșterea și descreșterea se referă strict la lungimi de liste. Să reanalizăm semantica argumentelor: primul este intrarea (lista ce trebuie inversată). Cel de-al doilea (parametrul de acumulare) conține în permanență lista inversată a elementelor peste care am trecut deja din lista de intrare (elementele din intrare care au fost puse "pe stivă" și ca atare s-a obținut implicit efectul de inversare). Acum, dacă la momentul inițial lista de intrare avea lungimea $l1$, și parametrul de acumulare lungime 0 (este listă vidă la apel) ne așteptăm ca rezultatul să aibă tot lungimea $l1$ având în vedere că trebuie să conțină exact aceleași elemente ca și prima listă, doar că în ordine inversă (și acest lucru poate fi verificat pe rezultatul final, când ieșirea are aceeași lungime cu intrarea). Analizând ca și în cazul primului predicat clauza recursivă la un moment oarecare de timp, și plecând de la lungimea $l1$ a primului argument (`[Head|Tail]`), rezultă imediat că `Tail` are lungimea $l1-1$. În apelul recursiv, primul argument, `Tail`, are în mod evident lungimea $l1-1$, argumentul al treilea, `[Head|Rest]` are în mod necesar lungimea $l1+1$, deoarece s-a adăugat un nou element prin șablon. Deci, argumentul de ieșire are monotonie inversă cu argumentul de intrare.

Luând în considerare și al doilea parametru, o reanalizare conduce la:
 $|[Head|Tail]| = l1, |List| = l2, |[Head|Rest]| = l3$, cu relația

$l2=l1+l3$. În timp ce apelul recursiv descrește primul argument, îl crește pe al treilea, menținând adevărată relația $l2=l1+l3$, deoarece $l2=(l1-1)+(l3-1)$, iar parametrul 2 rămâne neschimbat pe toată durata execuției (este doar o variabilă liberă, care se transmite în arborele de execuției de la rădăcină până la frunză cu scopul de a captura rezultatul final. Instanțierea la nivelul frunzei va produce instanțierea tuturor variabilelor "de pe traseu" și deci și a variabilei din apelul inițial).

Analiza pe acest exemplu ne permite să facem o observație cu un grad de generalitate mai mare, și anume: dacă argumentul de ieșire are aceeași monotonie cu cel de intrare (prin analiza semantică, se poate afirma că cele două argumente cresc, respectiv descresc deodată), șablonul (de compunere a rezultatului) se aplică în capul regulii. Dacă dimpotrivă, ieșirea este semantic în "opoziție" cu intrarea (conținutul ei crește în timp ce al intrării scade), atunci șablonul se aplică în corpul regulii.

4.9 Sublistă

Să se verifice dacă pentru două liste furnizate ca argumente una dintre ele este sublistă în cealaltă (apare compact în cealaltă).

Exemple de rulare:

```
?-sublist([1,2,3],[0,1,2,3,4]).
yes.
?-sublist([1,2,3],[0,1,5,2,6,3,4]).
no.
?-sublist([1,2,3],[0,1,5,1,2,6,1,2,3,4]).
yes.
```

```
%sublist_1/2
%sublist_1(sublista, Lista)

sublist_1([H|T],[H|L]):-
    is_sublist(T,L),!.

sublist_1(Sublist,[_|T]):-
    sublist_1(Sublist,T).
sublist_1([],_).
```

```
%is_sublist/2
%is_sublist(sublista, lista)
```

```
is_sublist([H|T],[H|L]):-
    is_sublist(T,L).
is_sublist([],_).
```

La execuție, pentru apelurile:

```
run1_1:-sublist_1([1,2,3],[0,1,2,3,4]).
run1_2:-sublist_1([1,2,3],[0,1,5,2,6,3,4]).
run1_3:-sublist_1([1,2,3],[0,1,5,1,2,6,1,2,3,4]).
```

Se obțin rezultatele așteptate:

```
| ?- run1_1.
yes
| ?- run1_2.
no
| ?- run1_3.
yes
```

Această primă variantă are următoarea abordare: atât timp cât primul element din listă este diferit de cel din sublistă se avansează în listă, renunțând la elementul curent (neputându-se realiza matching pe prima clauză, primele elemente fiind distincte, se intră în clauza 2). În primul moment în care primul element din sublistă coincide cu elementul curent al listei, se trece la avansarea simultană în cele două liste. Avansul simultan se realizează prin intermediul unui alt predicat, `is_sublist/2` pentru ca în cazul nefavorabil în care după câteva elemente care coincid apare unul distinct (exemplul 3) să se poată reveni de la începutul sublistei. Revenirea este asigurată de eșecul `is_sublist/2` (care nu are altă condiție de terminare decât ajungerea la sublista vidă), eșec care permite avansul peste elementul curent al listei și reluarea sublistei (clauza 2 în `sublist_1/2`). Pentru cazul în care se ajunge la lista vidă iar sublista nu a fost parcursă în întregime nu trebuie menționată specific o clauză. Aceasta ar avea forma:

```
is_sublist(_,[]):-fail,!.
```

dar simpla ei absență are același efect (neexistând o clauză peste care să se suprapună, o întrebare de acest fel va eșua implicit).

```
sublist_2(Sublist,List):-
    sublist_2_1(Sublist,Sublist,List).
```

```
%sublist_2_1/2
%sublist_2_1(sublista, copie sublista, lista)

sublist_2_1([H|T], Sublist, [H|R]) :-
    sublist_2_1(T, Sublist, R).
sublist_2_1([H1|T], Sublist, [H2|R]) :-
    sublist_2_1(Sublist, Sublist, R).
sublist_2_1([], _, _).
```

Cu rezultatele:

```
run2_1:-sublist_2([1,2,3],[0,1,2,3,4]).
run2_2:-sublist_2([1,2,3],[0,1,5,2,6,3,4]).
run2_3:-sublist_2([1,2,3],[0,1,5,1,2,6,1,2,3,4]).

| ?- run2_1.
yes
| ?- run2_2.
no
| ?- run2_3.
yes
```

Principial metoda este similară cu cea anterioară. Deosebirea constă în aceea că, revenirea la începutul sublistei după găsirea câtorva elemente consecutive în listă se realizează prin intermediul unei copii a variabilei sublistă. La începutul execuției se păstrează o mostră a sublistei. Dacă avansul în sublistă nu s-a produs până la sublista vidă, se revine cu căutarea din nou de la începutul sublistei (reinițializând-o din mostra păstrată martor).

```
sublist_3(Sublist, List) :-
    append(X, _, List),
    append(_, Sublist, X).

run3_1:-sublist_3([1,2,3],[0,1,2,3,4]).
run3_2:-sublist_3([1,2,3],[0,1,5,2,6,3,4]).
run3_3:-sublist_3([1,2,3],[0,1,5,1,2,6,1,2,3,4]).
| ?- run3_1.
yes
| ?- run3_2.
no
| ?- run3_3.
yes
```

Semantica: dintre toate descompunerile posibile ale argumentului `List` în două liste, generează-o pe cea care are în componență ca prim argument o listă, care prin descompunerea nedeterministă are ca al doilea argument lista `Sublist`. (acum este momentul să revedem valențele comportamentului nedeterminist ale predicatului `append/3` studiat la începutul acestui capitol).

```
sublist_4(Sublist,List):-
    append3(_,Sublist,_,List).

append3([],[],L,L).
append3([], [H|T], L, [H|R]) :-
    append3([],T,L,R).
append3([H|T],L1,L2,[H|R]) :-
    append3(T,L1,L2,R).

run4_1:-sublist_4([1,2,3],[0,1,2,3,4]).
run4_2:-sublist_4([1,2,3],[0,1,5,2,6,3,4]).
run4_3:-sublist_4([1,2,3],[0,1,5,1,2,6,1,2,3,4]).
| ?- run4_1.
yes
| ?- run4_2.
no
| ?- run4_3.
yes
```

Semantica: dintre toate descompunerile în trei liste posibile ale argumentului `List` generează-o pe cea care o are în componență ca al doilea argument lista `Sublist`. (în acest moment putem revizui răspunsul la problema 11 din acest capitol).

```
sublist_5(X,Y):-
    append(X,_,Y).
sublist_5(X,[_|Y]):-
    sublist_5(X,Y).

run5_1:-sublist_5([1,2,3],[0,1,2,3,4]).
run5_2:-sublist_5([1,2,3],[0,1,5,2,6,3,4]).
run5_3:-sublist_5([1,2,3],[0,1,5,1,2,6,1,2,3,4]).
| ?- run5_1.
yes
| ?- run5_2.
no
| ?- run5_3.
```

```
yes
| ?
```

Soluția a 5-a nu este decât o variantă pe jumătate explicitată iterativ a ultimelor două (cu condiția ca predicatul `append/3` să aibă clauza recursivă prima clauză și faptul clauza a doua): argumentul prim este sublistă a argumentului secund dacă descompunerea argumentului secund începe chiar cu primul argument; în caz contrar, se avansează peste un element în lista mare, (se "sare" peste prima descompunere din varianta 3) și se încearcă din nou descompunerea argumentului 2, astfel încât să înceapă cu argumentul 1.

O altă utilizare interesantă pentru predicatul `append` este cea în care se determină toate sublistele având o monotonie precizată (crescătoare în exemplul implementat de noi):

```
ascending(L,R) :-
    append3_2(_,R,_,L),
    is_ascending(R).

%is_ascending/1
%is_ascending(lista)

is_ascending([X,Y|T]) :-
    X<Y,
    is_ascending([Y|T]).
is_ascending([_]).

?- ascending([1,5,2,4,8,-1,7], Res).
Res = [1] ? ;
Res = [1,5] ? ;
Res = [5] ? ;
Res = [2] ? ;
Res = [2,4] ? ;
Res = [2,4,8] ? ;
Res = [4] ? ;
Res = [4,8] ? ;
Res = [8] ? ;
Res = [-1] ? ;
Res = [-1,7] ? ;
Res = [7] ? ;
no
```

4.10 Exerciții și probleme propuse

1. În ce măsură este afectată semantica predicatului `member/2` de schimbarea ordinii clauzelor, adică:

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

2. Dacă ordinea clauzelor `member/2` este cea din exercițiul 1, câte răspunsuri, și care este ordinea de obținere a lor la întrebarea:

```
?-member(X, [1, 2, 3]) .
```

3. Ce răspuns are întrebarea:

```
?-append(X, Y, Z) .
```

4. Câte soluții se obțin la întrebarea

```
?-append3_1(X, Y, Z, [1, 2, 3]) .
```

În ce ordine se obțin?

5. Câte soluții se obțin la întrebarea

```
?-append3_2(X, Y, Z, [1, 2, 3]) .
```

În ce ordine se obțin?

6. Câte soluții se obțin la întrebarea

```
?-append3_3(X, Y, Z, [1, 2, 3]) .
```

În ce ordine se obțin?

7. Predicatul `append3_3` este similar `append3_1` cu sau cu `append3_2`? Cum explicați?

8. Modificăm ordinea clauzelor predicatului `append3_3` după cum urmează:

```
append3_3([], [], L, L) .
append3_3([H|T], L1, L2, [H|R]) :-
    append3_3(T, L1, L2, R) .
append3_3([], [H|T], L, [H|R]) :-
    append3_3([], T, L, R) .
```

Se schimbă eficiența (numărul de șabloane aplicate pentru obținerea rezultatului)? Câte soluții, care și în ce ordine se obțin pentru întrebarea:

```
?-append3_3(X,Y,Z,[1,2,3]).
```

9. Modificăm ordinea clauzelor predicatului `append3_3` după cum urmează:

```
append3_3([H|T],L1,L2,[H|R]):-
    append3_3(T,L1,L2,R).
append3_3([], [H|T],L,[H|R]):-
    append3_3([],T,L,R).
append3_3([],[],L,L).
```

Se schimbă eficiența (numărul de șabloane aplicate pentru obținerea rezultatului)? Câte soluții, care și în ce ordine se obțin pentru întrebarea:

```
?-append3_3(X,Y,Z,[1,2,3]).
```

10. Modificăm ordinea clauzelor predicatului `append3_3` după cum urmează:

```
append3_3([], [H|T],L,[H|R]):-
    append3_3([],T,L,R).
append3_3([H|T],L1,L2,[H|R]):-
    append3_3(T,L1,L2,R).
append3_3([],[],L,L).
```

Se schimbă eficiența (numărul de șabloane aplicate pentru obținerea rezultatului)? Câte soluții, care și în ce ordine se obțin pentru întrebarea:

```
?-append3_3(X,Y,Z,[1,2,3]).
```

11. Pentru întrebarea:

```
?-append3_1(_,X,_,[1,2,...,n]).
```

câte răspunsuri există?

12. Pentru întrebarea:

```
?-append3_2(_,_,[H|_],[1,2,...,n]).
```

câte răspunsuri există?

13. Pentru întrebarea:

```
?-append3_2([_|T],_,_,[1,2,...,n]).
```

câte răspunsuri există?

14. Ce răspuns se obține la întrebarea:

```
| ?- delete_all(X,[1,2,1,3,1],R).
```

15. Câte răspunsuri se obțin pentru întrebarea

```
| ?- delete_all(1, [1, 2, 1, 3, 1], R) .
```

Justificați.

16. Transformați predicatul `delete_all/3` într-unul determinist.

17. Câte răspunsuri, în ce ordine, și carte sunt acestea, pentru întrebarea:

```
| ?- subst(X, 4, [1, 2, 1, 3, 1], R) .
```

18. Câte răspunsuri, în ce ordine, și carte sunt acestea, pentru întrebarea:

```
| ?- subst_all(1, 4, [1, 2, 1, 3, 1], R) .
```

19. Câte răspunsuri, în ce ordine, și carte sunt acestea, pentru întrebarea:

```
| ?- subst_all(X, 4, [1, 2, 1, 3, 1], R) .
```

20. Transformați predicatul `subst_all/4` într-unul determinist.

21. Desenați arborele de deducție pentru `list_length_2/2` și întrebarea

```
?- list_length_2([1, 2, 3, 4, 5, 6, 7], 0) .
```

cu instanțierile la nivelul fiecărui nod. Evidențiați în arbore nodul în care lungimea întregii liste s-a calculat. De ce acest rezultat nu este disponibil (nu este returnat)?

22. Care este efectul mutării șablonului unui argument de ieșire de forma `[H|T]` din capul regulii în corpul său (în apelul recursiv al aceluiași predicat)?

23. Care este semnificația unui argument de ieșire de forma `[H|T]` în capul unei reguli?

24. Care este semnificația unui argument de ieșire de forma `[H|T]` în apelul recursiv al unui predicat (corpul caluzei aceluși predicat)?

25. Desenați arborele de deducție pentru `list_length_2/3` și întrebarea

```
?- list_length_3([1, 2, 3, 4, 5, 6, 7], Length, 0) .
```

cu instanțierile la nivelul fiecărui nod. Observați cum la terminarea execuției instanțierea din ultimul nod al arborelui produce instanțierea variabilelor pe întreg lanțul înapoi.

26. De ce pentru `list_length_2` este nevoie de o variabilă liberă pentru memorarea rezultatului final în timp ce pentru `list_length_1` această variabilă nu e necesară?

27. Dați câte un exemplu de aplicație în care fiecare din lungimile parțiale ale unei liste determinate prin cele două metode (înainte și respectiv înapoi) să fie necesare.

28. Dați un exemplu de calcul al sumei elementelor unei liste în care metoda recursivității înainte să fie mai avantajoasă.

29. Propuneți o problemă a cărei soluție recursivă să permită atât abordarea înainte cât și cea înapoi. Realizați implemetarea în cele două tehnici, și realizați o analiză comparativă a lor. Justificați necesitatea fiecăreia dintre ele, imaginând câte o situație reală în care fiecare dintre ele (pe rând) să prezinte avantaje.

30. Comparați cele patru implementări ale `sublist`.

31. Care din cele patru `sublist` implementări este cea mai eficientă? Cea mai ineficientă?

32. Găsiți o altă soluție a problemei `sublist`. Comparați-o cu soluțiile prezentate.

33. La soluția `sublist_1/2` se propune înlocuirea predicatului `is_sublist/2` cu următorul predicat

```
is_sublist_new_version(Sublist,List):-
    append(_,Sublist,List).
```

Este aceasta o soluție corectă?

34. Comparați soluția propusă în problema 23 cu `sublist_3/2`. Ce se observă?

35. Presupunem că în `sublist_1/2` facem următoarea modificare:

```
is_sublist([H|T],[H|L]):-!,
    is_sublist(T,L).
```

Rezultatul este unul corect? Justificați. Care dintre întrebările specificate la exemplificare problemei va avea de suferit?

36. Presupunem că în `sublist_2_1/3` facem următoarea modificare:

```
sublist_2_1([H|T],Sublist,[H|R]):-!,  
    sublist_2_1(T,Sublist,R).
```

Rezultatul este unul corect? Justificați. Care dintre întrebările specificate la exemplificare problemei va avea de suferit?

37. Pentru întrebarea:

```
?-sublist([1,2],[1,3,1,2,1,3,2,3,1,2,3,1,3,1,2,3])
```

câte răspunsuri afirmative se obțin la fiecare din cele patru metode de implementare menționate în text?

5. Sortarea

Una din cele mai utilizate probleme atunci când se prezintă un limbaj de programare este cea a ordonării (nedescrescătoare) a unui șir de elemente. Vom considera și noi această problemă atât pentru naturalețea exprimării în limbaj logic a diferitelor metode, cât și pentru eleganța acestora. De asemenea problema reprezintă un foarte bun suport pentru prezentarea în același timp a unor mecanisme caracteristice limbajelor logice, și a unei varietăți de implementări ale aceleiași metode principale, precum și analiza comparativă a implementărilor.

Metodele directe de sortare reprezintă o primă abordare pentru că nu fac apel la nici o tehnică de programare, ci realizează implementarea direct pe baza specificațiilor. Dar vom porni de la o implementare care pleacă direct de la definiția ordonării unei structuri.

5.1 Sortarea prin generarea permutărilor

Dacă pornim de la observația că ordonarea unei liste nu este altceva decât găsirea permutării structurii de intrare care are proprietatea că elementele sale sunt ordonate, și transcriind "logic" afirmația anterioară, rezultă:

```
%sort_perm/2
% sort_perm(intrare, iesire)

sort_perm(In,Out):-
    permutation(In,Out),
    is_ordered(Out),!.
```

Surprinzător de natural! Ce putem spera mai mult de la un limbaj de programare decât să exprimăm imediat în sintaxa limbajului observațiile făcute (axiome – transpuse în fapte, raționamente – transpuse în clauze).

Dacă predicatul `is_ordered/1` este absolut banal (necesitând parcurgerea listei, cu testare relației de ordine între elemente succesive):

```
is_ordered ([_]).
is_ordered ([H1,H2|T]):-
    order(H1,H2),
    is_ordered ([H2|T]).
```

`order(A,B) :- A < B.`

nu același lucru putem afirma despre generarea tuturor permutărilor. Totuși, și această problemă se rezolvă surprinzător de simplu.

Vom porni de la numărul permutărilor unei mulțimi de n elemente, și de la definiția acestui număr. Vom avea $n!$ permutări, și pornind de la relația de recurență a factorialului

$$n! = (n-1)! \cdot n$$

vom genera chiar permutările. Permutările diferă prin natura și ordinea elementelor. Deci, pentru a genera toate permutările, trebuie DOAR să selectăm 1 element (aleatoriu; dacă selecția asigură faptul că mai devreme sau mai târziu TOATE elementele vor fi selectate, succesul este asigurat!), să-l fixăm pe prima poziție în lista de ieșire, după care, să asigurăm generarea (după aceeași tehnică) a permutărilor structurii din care elementul selectat (aleatoriu) lipsește. Selecția (aleatorie) ne asigură factorul n al relației recursive de mai sus, în timp ce factorul $(n-1)!$ este asigurat de generarea după aceeași tehnică a permutărilor structurii din care elementul selectat aleatoriu lipsește. Transcrierea gândirii umane folosind sintaxa limbajelor logice ar fi:

```
permutation(In,Out) :-
    select(One,In),
    remove(One,In,Int),
    permutation(Int,Res),
    compose(One,Res,Out).
```

Nu trebuie decât să identificăm cum putem realiza fiecare din pașii menționați. Considerând aspectul *aleatoriu* necesar pentru selecție (rând pe rând să fie selectat fiecare element din intrare), pentru select imediat se identifică predicatul `member`, varianta nedeterministă a sa. Deci `select(One,In) ≡ member(One,In)`

În același timp, eliminarea unui element se realizează cu predicatul deja discutat `delete_el`, și deci `remove(One,In,Int) ≡ delete_el(One,In,Int)`. Singura dificultate a rămas compunerea rezultatului. Cum ordinea permutărilor generate nu interesează, avem varianta imediată a șablonului de compunere/descompunere a listelor. Și deci `compose(One,Res,Out) ≡ Out=[One|Res]`.

Cu aceasta, putem scrie predicatul de generare a permutărilor:

```
%permutation1/2
%permutation1(input, output)

permutation1([], []).
permutation1(In, [One|Res]):-
    member(One, In),
    delete_el(One, In, Int),
    permutation1(Int, Res).

| ?- permutation1([1,2,3], Permutation).
Permutation = [1,2,3] ? ;
Permutation = [1,3,2] ? ;
Permutation = [2,1,3] ? ;
Permutation = [2,3,1] ? ;
Permutation = [3,1,2] ? ;
Permutation = [3,2,1] ? ;
no
```

Soluția se bazează pe caracterul nedeterminist al predicatului `member/2`, care, rînd pe rînd, va selecta toate elementele din intrare, ca în rularea:

```
| ?- member(X, [1,2,3]).
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
no
```

Secvența `member(One, In), delete_el(One, In, Int)`, selectează nedeterminist un element oarecare `One` pe care îl șterge, furnizând lista `Int` din care `One` lipsește. Adică:

```
| ?- member(X, [1,2,3]), delete_el(X, [1,2,3], Y).
X = 1, Y = [2,3] ? ;
X = 2, Y = [1,3] ? ;
X = 3, Y = [1,2] ? ;
no
```

Și deci apelul recursiv este posibil a se realiza, contribuind la realizarea factorului $(n+1)!$.

O analiză mai atentă ne va permite să facem observația că în timp ce apelul `member` selectează nedeterminist un element, apelul `delete_el` produce eliminarea deterministă a elementului precizat. Dar dacă rolul lui `member` este doar de selecție nedeterministă, nu poate juca acest rol `delete_el`? Cu alte cuvinte, care este răspunsul la o întrebare de forma:

```
| ?- delete_el(X, [1,2,3],Y) .
X = 1, Y = [2,3]? ;
X = 2, Y = [1,3]? ;
X = 3, Y = [1,2]? ;
no
```

Efectul fiind același, nu merită (eficiența!) să "irosim" o parcurgere cu selecția nedeterministă, și apoi ștergerea deterministă. Predicatul de generare a permutărilor se rescrie:

```
%permutation1/2
%permutation1(input, output)

permutation1([], []).
permutation1(In, [One|Res]) :-
    delete_el(One, In, Int),
    permutation1(Int, Res).

| ?- permutation1([1,2,3],Permutation) .

Permutation = [1,2,3] ? ;
Permutation = [1,3,2] ? ;
Permutation = [2,1,3] ? ;
Permutation = [2,3,1] ? ;
Permutation = [3,1,2] ? ;
Permutation = [3,2,1] ? ;
no
```

Pornind de la aceeași definiție a permutărilor de mai sus, dar identificând o altă variantă pentru selecția aleatoare și eliminarea elementului selectat, vor obține o altă soluție, specifică limbajelor logice.

Predicatul de concatenare `append` a fost construit cu scopul declarat de a concatena două liste, dar cum am văzut, deoarece în limbajele logice semantica de argument de intrare sau ieșire nu este foarte strictă (în realitate, o variabilă putând juca, în funcție de context, oricare dintre roluri) rolul predicatului poate fi și acela de descompunere nedeterministă a unei liste

(cea de ieșire din accepțiunea clasică) în listele componente ale sale. Deci apelul:

```
| ?- append(X,Y,[1,2,3,4]).
```

Furnizând răspunsurile:

```
X = [], Y = [1,2,3,4] ? ;
X = [1], Y = [2,3,4] ? ;
X = [1,2], Y = [3,4] ? ;
X = [1,2,3], Y = [4] ? ;
X = [1,2,3,4], Y = [] ? ;
no
```

avem doar un pas pentru a selecta nedeterminist un element, și anume să-i "spunem" apelului să ne selecteze elementul A, adică:

```
| ?- append(_, [A|_], [1,2,3,4]).
```

va furniza:

```
A = 1 ? ;
A = 2 ? ;
A = 3 ? ;
A = 4 ? ;
no
```

Acum putem să-i precizăm că dorim să ne și returneze lista din care elementul selectat lipsește, adică:

```
| ?- append(A, [X|B], [1,2,3,4]), append(A,B,Y).
```

va furniza:

```
X = 1, Y = [2,3,4] ? ;
X = 2, Y = [1,3,4] ? ;
X = 3, Y = [1,2,4] ? ;
X = 4, Y = [1,2,3] ? ;
no
```

Intenționat am omis transcrierea valorilor pentru A și B care nu ne interesează în fapt. Acum nu putem decât să remarcăm faptul că secvența:

```
run_append(X,Y):-
    append(A, [X|B], [1,2,3,4]), append(A,B,Y).
```

```

| ?- run_append(X,Y) .
X = 1,
Y = [2,3,4] ? ;
X = 2,
Y = [1,3,4] ? ;
X = 3,
Y = [1,2,4] ? ;
X = 4,
Y = [1,2,3] ? ;
no

```

realizează EXACT același lucru ca și secvența anterioară:

```

run_nondeterm(X,Y):-
    member(X,[1,2,3,4]), delete_el(X,[1,2,3,4],Y) .

| ?- run_nondeterm(X,Y) .
X = 1,
Y = [2,3,4] ? ;
X = 2,
Y = [1,3,4] ? ;
X = 3,
Y = [1,2,4] ? ;
X = 4,
Y = [1,2,3] ? ;
no

```

Cu această observație, generarea permutărilor devine:

```

%permutation2/2
%permutation2(input, output)

permutation2([],[]).
permutation2(In,[One|Res]):-
    append(A,[One|B],In),
    append(A,B,Int),
    permutation2(Int,Res) .

```

unde primul apel de append este nedeterminist (de descompunere a listei de intrare în toate variantele posibile, selectând astfel, la fiecare descompunere un alt element One de poziționat pe prima poziție în rezultat), și acesta joacă rolul apelului de member/2 din permutation1/2, iar cel de-al doilea determinist, de refacere a structurii Int, pentru a fi pasată ca argument de

intrare apelului recursiv, iar acesta joacă rolul lui `delete_el/3` din aceeași implementare. Surprinzătoare valențe pot fi evidențiate pentru un predicat simplu cum este `append/3` dacă se evidențiază mecanisme specifice ale programării logice. Din păcate, deși rezultatele rulării, așa cum așteptam sunt aceleași:

```
| ?- permutation1([1,2,3],Permutation).
Permutation = [1,2,3] ? ;
Permutation = [1,3,2] ? ;
Permutation = [2,1,3] ? ;
Permutation = [2,3,1] ? ;
Permutation = [3,1,2] ? ;
Permutation = [3,2,1] ? ;
no
```

nu putem renunța la nici unul din cele 2 apeluri, așa cum am făcut la `permutation1/2`. Dacă însă facem o foarte scurtă analiză de eficiență, vom observa, așa cum ne așteptam, $O(n!)$, ceea ce ne face să omitem efectul constantei multiplicative induse de 2 apeluri consecutive pe structura liniară, în loc de unul.

5.2 Sortarea prin selecție

Este o primă opțiune care se bazează pe alegerea (selectarea, de unde și numele metodei) unui element cu o anumită proprietate (minim sau maxim) și așezarea acestuia în poziția corespunzătoare a șirului ordonat. Având în vedere că reprezentarea la îndemână este cea a listelor, și că accesul rapid în liste se realizează la capul acestora (compunerea-descompunerea listelor după primul element necesitând $O(1)$), înseamnă că soluția naturală va fi cea prin care selectăm minimul pe care îl poziționăm în capul listei rezultat.

Deci pentru rezolvarea problemei avem mai întâi nevoie de cel mai mic (minimul) element al listei. O abordare naturală (provenind chiar direct din gândirea umană, făcând abstracție de existența limbajelor de programare) este următoarea: parcurgem toate elementele din zona de analizat, și comparăm elementul curent cu cel mai mic element găsit până în acel moment. Dacă elementul curent este mai mare sau egal cu cel mai mic element găsit până la pasul curent, atunci minimul rămâne neschimbat. În caz contrar (elementul curent e mai mic decât minimul găsit până la pasul

curent) elementul curent devine noul minim. Procesul continuă până când se termină șirul de intrare.

Având în vedere că metoda necesită o valoare inițială pentru minim, este natural ca aceasta să fie (la primul pas) considerată primul element al listei.

```
min([Head|Tail],Minim):-min(Tail,Head,Minim)
```

Unde semnificația argumentelor este:

```
%min/2
min(lista căreia se calculează minimul, minimul listei
furnizată ca prim argument)
```

```
%min/3
min(lista căreia se calculează minimul, minimul
parțial, minimul listei furnizată ca prim argument)
```

Pornind de la această analiză, o primă specificație în limbaj logic a metodei este:

```
min1([],M,M).
min1([H|T],M,Min):-
    H<M,
    min1(T,H,Min).
min1([H|T],M,Min):-
    H>=M,
    min1(T,M,Min).

min1([H|T],Min):-min1(T,H,Min).
```

O execuție cu tipărirea rezultatelor parțiale (valorile minimului calculate pe măsură ce se avansează în listă) ne indică:

```
| ?- min1([8,5,7,4,9,1,3,2,6],Minim).
minimul partial s-a initializat la MinP=8
minimul partial s-a schimbat la MinP=5
minimul partial s-a schimbat la MinP=4
minimul partial s-a schimbat la MinP=1
minimul este Minim=1
Minim = 1 ? ;
no
```

Dacă facem un pas înainte și realizăm reprezentarea matematică a acestei specificații, atunci:

$$\min(M) = \min(C \cup M') = \begin{cases} C & \text{dacă } C \leq \min(M') \\ \min(M') & \text{dacă } C > \min(M') \end{cases}$$

și trecând la specificația logică a reprezentării matematice de mai sus:

```
min2([H], H).
min2([H|T], H) :-
    min2(T, H, Min),
    H < M.
min2([H|T], Min) :-
    min2(T, M, Min),
    H >= M.
```

O primă observație este aceea că, datorită faptului că se face comparația la revenirea din recursivitate, parametrul de acumulare nu mai este necesar, deci am redus argumentele la două. Aceasta este o caracteristică generală dacă se face o analiză comparativă între metodele de rezolvare a unei probleme cu recursivitate înainte (luarea deciziei înainte de apelul recursiv) respectiv înapoi (luarea deciziei doar la revenirea din recursivitate). Afirmăm deocamdată că este o primă caracteristică, favorabilă recursivității înapoi. Vom vedea imediat și alte diferențe favorabile acestui tip de recursivitate. Trebuie însă remarcat faptul că se pierde avantajul major al recursivității înainte: *last call optimisation*. Prin faptul că în recursivitatea înainte deciziile se iau înaintea apelului recursiv, este posibil ca apelul în sine să fie ultimul subscop în conjuncția de subscopuri din corpul clauzei. Datorită faptului că apelul este ultimul se realizează o creștere a eficienței la implementare (în timp și spațiu), datorită manierei în care starea programului este salvată/refăcută. Un alt avantaj îl constituie obținerea de rezultate parțiale ale structurii de date pe măsură ce se parcurge. Acest fapt are semnificație atât de sine stătător, prin valoare intrinsecă a acestui rezultat, dar și pentru că un astfel de rezultat poate fi oferit unor procese care rulează concurrent.

O observație imediată care se poate face legat de predicatul `min2` este aceea că dacă luăm în considerare a 2-a și a 3-a clauză, ele sunt disjuncte, ceea ce înseamnă că backtracking-ul (predicatul fiind determinist) nu ar provoca decât o irosire de timp. Deci, ieșind din zona logicii pure, prin adăugarea

tăierii de backtracking, comparația din clauza 3 nu mai este necesară. Deci predicatul se rescrie:

```
min2([H],H).
min2([H|T],H):-
    min2(T,H,Min),
    H<M,! .
min2([H|T],Min):-
    min2(T,M,Min).
```

Câștigul nu este prea însemnat, dacă avem în vedere faptul că timp considerabil se folosește pentru apelul recursiv, și nu pentru comparație (o operație elementară). Putem însă mai bine de atât. Dacă ne uităm atent la această formă, realizăm că în clauza 2 apelul recursiv e necesar pentru a furniza termenul comparației $H < M$ iar în clauza 3 pentru a furniza rezultatul însuși. Putem combina cele două necesități (furnizarea termenului comparației și a rezultatului) dacă această clauză (a 3a) devine cea în care comparația se produce. Deci, dacă în forma inițială a predicatului schimbăm ordinea clauzelor 2 și 3 obținem:

```
min2([H],H).
min2([H|T],Min):-
    min2(T,M,Min),
    H>=M.
min2([H|T],H):-
    min2(T,H,Min),
    H<M.
```

Dacă acestei forme îi adăugăm tăierea de backtracking în clauza 2, devine evident că nu mai este necesară comparația din clauza 3, deci predicatul se rescrie sub forma:

```
min2([H],H).
min2([H|T],Min):-min2(T,M,Min),H>=M,! .
min2([H|T],H):-min2(T,H,Min).
```

Dar un lucru interesant! Minimul Min calculat prin apel recursiv în clauza 3 NU mai este necesar, deoarece minimul este chiar primul element al listei H . Deci apelul recursiv nu mai trebuie efectuat, ținând cont de faptul că unica rațiune de a-l efectua era aceea de a calcula minimul din restul listei. Deci predicatul se rescrie:

```

min2([H],H) .
min2([H|T],Min):-min2(T,M,Min),H>=M,! .
min2([H|T],H) .

```

Foarte interesant! Și eficient, ținând cont că am eliminat apelul recursiv de pe o ramură. Dar mai departe, observăm că în ultima clauză coada listei T nu se mai folosește nicăieri, ceea ce înseamnă că pe poziția corespunzătoare putem folosi variabila universală, deci:

```

min2([H],H) .
min2([H|T],Min):-
    min2(T,M,Min),
    H>=M,! .
min2([H|_],H) .

```

Acum mai putem observa faptul că prima clauză este doar un caz particular al celei de-a treia (când coada listei e vidă, deci $[H]=[H|_]$ pentru cazul $_=[H]$), și ca atare prima clauză poate lipsi (fiind suplinită de a treia). Cu această observație ajungem la forma finală a predicatului:

```

min2([H|T],Min):-
    min2(T,Min),
    Min<H,! .
min2([H|_],H) .

```

O trasare a execuției acestui predicat poate pune mai bine în evidență modalitatea de funcționare a predicatului.

```

?- min2([2,9,11,-3,5],R).
R = -3 ?
yes

```

```

?- min2([6,-2,1,-3,4],R).
1   1 Call: min2([6,-2,1,-3,4],_514) ?
2   2 Call: min2([-2,1,-3,4],_514) ?
3   3 Call: min2([1,-3,4],_514) ?
4   4 Call: min2([-3,4],_514) ?
5   5 Call: min2([4],_514) ?
6   6 Call: min2([],_514) ?
6   6 Fail: min2([],_514) ?
5   5 Exit: min2([4],4) ?
7   5 Call: 4<-3 ?
7   5 Fail: 4<-3 ?

```

```

4 4 Exit: min2([-3,4],-3) ?
8 4 Call: -3<1 ?
8 4 Exit: -3<1 ?
3 3 Exit: min2([1,-3,4],-3) ?
9 3 Call: -3<-2 ?
9 3 Exit: -3<-2 ?
2 2 Exit: min2([-2,1,-3,4],-3) ?
10 2 Call: -3<6 ?
10 2 Exit: -3<6 ?
1 1 Exit: min2([6,-2,1,-3,4],-3) ?
R = -3 ?
yes
% trace

```

Execuții cu tipărirea valorilor intermediare ale minimului (urmăriți sensul în care se produc instanțierile, și comparați cu rezultatele produse de `min1/3`, prima execuție fiind chiar pentru aceleași date de intrare):

```

| ?- min2([8,5,7,4,9,1,3,2,6],Minim) .
minimul partial s-a schimbat la MinP=6
minimul partial s-a schimbat la MinP=2
minimul partial s-a schimbat la MinP=1
Minim = 1 ? ;
no

| ?- min2([8,9,1,3,2,5,4,7,6],Minim) .
minimul partial s-a schimbat la MinP=6
minimul partial s-a schimbat la MinP=4
minimul partial s-a schimbat la MinP=2
minimul partial s-a schimbat la MinP=1
Minim = 1 ? ;
no

| ?- min2([2,9,11,-3,5],Minim) .
minimul partial s-a schimbat la MinP=5
minimul partial s-a schimbat la MinP=-3
Minim = -3 ? ;
no

```

Sunt remarcabile atât concizia cât și eficiența acestei implementări. O analiză imediată dovedește că timpul de execuție este $O(n)$ deci implementarea algoritmului este optimală.

Implementarea în limbaj logic a sortării prin selecție cu primul predicat de minim este:

```
sort_sel1([], []).
sort_sel1(List, [H|T]) :-
    min1(List, H),
    delete_el(H, List, Res),
    sort_sel1(Res, T).

| ?- sort_sel1([3,8,5,9,2], Result).
Result = [2,3,5,8,9] ?
yes
```

Iar prin tipărirea la execuție a rezultatelor intermediare, se observă că elementele se ordonează începând cu cele mari.

```
| ?- sort_sel1([3,8,5,9,2], Result).
lista partiala sortata este=[]
lista partiala sortata este=[9]
lista partiala sortata este=[8,9]
lista partiala sortata este=[5,8,9]
lista partiala sortata este=[3,5,8,9]
Result = [2,3,5,8,9] ? ;
no
```

Implementarea urmează imediat specificațiile de mai sus, și anume, minimul întregii liste este poziționat la începutul listei de ieșire, după care se continuă procesul cu restul listei de intrare (adică lista inițială din care se elimină minimul, eliminare obținută prin apelarea `delete_el`, predicat de ștergere a elementului specificat ca prim argument din lista specificată ca al doilea argument). Implementarea acestuia este imediată, și nu mai facem nici o analiză a lui, decât că se șterge doar prima apariție a elementului.

```
delete_el(X, [X|T], T).
delete_el(X, [H|T], [H|R]) :-
    delete_el(X, T, R).
%delete_el(_, [], []).
```

De remarcat că ultima clauză nu este necesară, deoarece în contextul sortării apelul va cere ștergerea unui element existent în lista de intrare.

Folosind cea de-a doua metodă de selecție a minimului, vom avea:

```

sort_sel2([], []).
sort_sel2(List, [H|T]) :-
    min2(List, H),
    delete_el(H, List, Res),
    sort_sel2(Res, T).

| ?- sort_sel2([3,8,5,9,2], Result).
Result = [2,3,5,8,9] ?
yes

```

Evident că soluția este aceeași, și evident că eficiența este superioară metodei anterioare, deoarece predicatul de minim (unica deosebire între cele două) este mai eficient (optimal în acest caz). Vestea bună este că această metodă poate fi încă îmbunătățită. O trecere rapidă în revistă va consemna faptul că predicatele de minim și ștergere a minimumului parcurg fiecare lista doar pentru a returna minimumul, respectiv lista din care acesta lipsește. O întrebare firească este: nu putem obține ambele efecte printr-o singură parcurgere? Răspunsul sperat este favorabil. Prin simpla adăugare a unui nou argument predicatului de minim, și anume:

```

min2([H|T], Min, [H|Result]) :-
    min2(T, Min, Result),
    H >= Min, !.
min2([H|T], H, T).

```

Singura modificare aplicată, în afară de adăugarea noului parametru, este înlocuirea variabilei universale din coada listei argument 1, deoarece de data aceasta ne interesează această listă, ea reprezentând lista din care minimumul lipsește. Cu aceasta, predicatul de sortare devine:

```

sort_sel2([], []).
sort_sel2(List, [H|T]) :-
    min2(List, H, Rest),
    sort_sel2(Rest, T).

| ?- sort_sel2([3,8,5,9,2], Result).
Result = [2,3,5,8,9] ?
yes

```

În timp ce rularea cu tipărirea rezultatelor intermediare ne indică o ordonare intermediară ca și în metoda anterioară. De remarcat că cele două metode diferă prin modalitatea de alegere a minimumului, valorile intermediare ale acestuia diferind.


```
| ?- sort_sel2([3,8,5,9,2],Result).
lista partiala sortata este=[]
lista partiala sortata este=[9]
lista partiala sortata este=[8,9]
lista partiala sortata este=[5,8,9]
lista partiala sortata este=[3,5,8,9]
Result = [2,3,5,8,9] ? ;
no
```

O analiză a eficienței pune în evidență faptul că, deși această variantă este o îmbunătățire a celei precedente, clasa algoritmului rămâne aceeași, $O(n^2)$, îmbunătățirea referindu-se la o constantă multiplicativă inferioară, datorată evitării parcurgerii repetate pentru selecție minim și ștergere.

Se poate deci observa că, pe lângă avantajele prezentate mai sus ale metodei cu recursivitate înainte (avantaje care implicit reprezintă dezavantaje ale celei de-a doua metode), recursivitatea înapoi are două avantaje majore, ambele legate de eficiență: eliminarea unui argument (nu mai avem nevoie de minim parțial și de minim global - variabilă liberă care să se instanțieze la sfârșitul execuției cu minimul parțial - ci de o singură variabilă, eficientizare în spațiul de memorie, pe de o parte, și respectiv evitarea parcurgerii duble a listei, eficientizare în timp).

5.3 Sortarea prin inserție

Cea de-a doua metodă de sortare directă are același principiu de bază ca și prima, și anume: spațiul de prelucrat (șirul de sortat, reprezentat ca listă în implementările logice) este împărțit din punct de vedere logic (împărțirea este virtuală, nu există un proces real de separare fizică) în două zone, sursa și destinația. Sursa este porțiunea (logică) din intrare care conține elementele de ordonat, iar destinația zona elementelor care deja au fost ordonate. Această împărțire logică este valabilă atât pe întreaga perioadă de procesare (în timpul sortării), dar reprezintă și o condiție respectiv postcondiție a sortării. Mai mult, în fiecare pas, un element din zona neordonată (sursă) este preluat și mutat în zona deja ordonată (destinație). Aceasta înseamnă ca la fiecare pas destinația se incrementează cu 1 iar sursa se decrementează cu 1, ceea ce asigură corectitudinea totală (presupunând corectitudinea parțială dovedită, elementul menționat mai sus reprezintă cheia inducției pentru terminarea programului). Justificarea

faptului că reprezintă și o precondiție respectiv postcondiție a sortării este foarte simplă: la pasul inițial sursa este întregul șir de intrare, destinația fiind vidă, iar la pasul final destinația este șirul deja ordonat sursa fiind vidă. Condiția ca cele două zone logice să reprezinte o partiție a șirului este asigurată (pe toată durata execuției, dar și la început respectiv la sfârșit cele două zone sunt disjuncte, iar reuniunea lor constituie întregul șir).

Principiul enunțat mai sus este valabil atât pentru sortarea prin selecție cât și pentru cea prin inserție. Deosebirea dintre ele o constituie modalitatea (efortul computațional) de preluare a elementului din sursă și adăugarea lui în destinație. La sortarea prin selecție întregul efort se focalizează pe faza preluării unui element din sursă, și anume, se caută un anume element al sursei, care are proprietatea că poate fi adăugat în zona destinație fără nici un efort (computațional). Aceasta înseamnă că la pasul următor se preia elementul din sursă următor având aceeași proprietate, și se adaugă după cel anterior adăugat, ele fiind totodată ordonate (atât relativ cât și absolut). Proprietatea care asigură această relație este cea de minim, și în selecția minimului se depune întreg efortul ($O(\text{lungimea listei})$), adăugarea lui în zona destinație necesitând doar $O(1)$ (compunerea cu ajutorul șablonului $[H|T]$). La sortarea prin inserție rolul se schimbă, se preia un element din zona sursă cu efort computațional minim, și se adaugă la locul potrivit în sursă, efortul întreg depunându-se în această direcție. Dacă privim așa lucrurile, preluarea unui element din sursă fără efort presupune considerarea unui element de la începutul listei (descompunerea după șablon necesită timp unitar); acest element va trebui potrivit în destinație. Pentru această potrivire trebuie baleată destinația (deja ordonată), iar elementul preluat din sursă adăugat acolo unde relația de ordine o indică. Costul acestei operații este $O(1)$ în cazul cel mai favorabil (se inserează un element mai mic decât cel mai mic element al destinației, adică primul) și respectiv $O(\text{lungimea listei})$ în cazul cel mai defavorabil (se inserează un element mai mare decât cel mai mare element al destinației, adică ultimul).

Predicatul de inserare:

```
% insert(element de inserat, lista in care se
insereaza, lista rezultat).
```

```
insert(E1, [H|T1], [H|T2]) :-
    order(H, E1), !,
    insert(E1, T1, T2) .
insert(X, L, [X|L]) .
```

```
order(N,M):-N<=M.
```

Unde `order` este predicatul reprezentând relația de ordine care se dorește a fi impusă. Dacă se dorește o mai mare lizibilitate a codului, se poate transcrie direct:

```
insert(E1,[H|T1],[H|T2]):-
    H<E1,!,
    insert(E1,T1,T2).
insert(X,L,[X|L]).
```

Sortarea utilizând inserarea de mai sus se scrie imediat:

```
%sort_ins1/2
%sort_ins1(lista de ordonat, lista ordonata)
sort_ins1(List,Result):-sort_ins1(List,[],Result).

%sort_ins1/3
%sort_ins1(lista de ordonat, parametru de
acumulare,lista ordonata)

sort_ins1([H|T],PartialResult,Result):-
    insert(H,PartialResult,NewPartialResult),
    sort_ins1(T, NewPartialResult,Result).
sort_ins1([],Result,Result).

| ?- sort_ins1([3,8,5,9,2],Result).
Result = [2,3,5,8,9] ?
yes
```

Recunoaștem în secvența de mai sus recursivitatea înainte: se aplică întâi inserarea elementului curent într-o structură existentă (un rezultat parțial, constituind de fapt destinația) urmată de apelul recursiv pe ceea ce a rămas din listă (sursa decrementată) și noul rezultat parțial (destinația incrementată). O altă caracteristică identificată este necesitatea existenței unui parametru suplimentar (parametru de acumulare, cu semantica rezultat parțial, reprezentând porțiunea de listă parcursă până la pasul curent și sortată), care se adaugă ca un nou parametru (parametrul 2) printr-un predicat special, `sort_ins1/2`. Acest parametru este lista vidă la început (corespunzător precondiției = destinație vidă) și respectiv rezultatul (corespunzător postcondiției = destinație întreaga listă). Inițializarea parametrului de acumulare se realizează prin apelul lui `sort_ins1/3` de

către `sort_ins1/2`, iar rezultatul se instanțiază, preluând valoarea parametrului de acumulare în ultimul pas, prin instanțierea implicită din cadrul faptului (clauzei a doua) a predicatului `sort_ins1/3`. De menționat de asemenea avantajul metodei directe, care realizează ordonarea listei de la începutul său, așa cum se poate observa pe execuția de mai jos, care tipărește rezultatele intermediare (lista obținută după fiecare inserare).

```
| ?- sort_ins([3,8,5,9,2],Result).
PartialResult=[]
PartialResult=[3]
PartialResult=[3,8]
PartialResult=[3,5,8]
PartialResult=[3,5,8,9]
PartialResult=[2,3,5,8,9]
Result = [2,3,5,8,9] ? ;
no
```

Recurсивitatea înapoi având ca principiu “rezolvarea problemei” începând cu zone de la sfârșitul structurii de date, începe cu apelul recursiv. Acesta furnizează destinația pentru inserare, ceea ce permite renunțarea la parametrul de acumulare. Metoda este implementată mai jos:

```
%sort_ins2/2
%sort_ins2(lista de ordonat, lista ordonată)

sort_ins2([H|T],Result):-
    sort_ins2(T,PRes),
    insert(H,PRes,Result).
sort_ins2([], []).
```

Unde `insert` are exact aceeași semantică și implementare ca și în cazul primei metode.

```
| ?- sort_ins([7,2,4,9,3,6],Result).
Result = [2,3,4,6,7,9] ?
Yes
```

Metoda cu recursivitate înainte prezintă același avantaj al reducerii spațiului folosit, neobservându-se nici o îmbunătățire în timp (nici măcar în cadrul aceleiași clase). La execuție cu tipărirea listelor intermediare se observă că rezultatul se formează dinspre celălalt capăt.

```
| ?- sort_ins2([3,8,5,9,2],Result).
PartialResult=[]
PartialResult=[2]
PartialResult=[2,9]
PartialResult=[2,5,9]
PartialResult=[2,5,8,9]
Result = [2,3,5,8,9] ? ;
no

| ?- sort_ins2([3,8,5,9,27,2,4,9,3,6],Result).
PartialResult=[]
PartialResult=[6]
PartialResult=[3,6]
PartialResult=[3,6,9]
PartialResult=[3,4,6,9]
PartialResult=[2,3,4,6,9]
PartialResult=[2,3,4,6,9,27]
PartialResult=[2,3,4,6,9,9,27]
PartialResult=[2,3,4,5,6,9,9,27]
PartialResult=[2,3,4,5,6,8,9,9,27]
Result = [2,3,3,4,5,6,8,9,9,27] ? ;
no
```

5.4 Sortarea prin interschimbare

Principiul este diferit de cel al metodelor anterior descrise, filosofia metodei constă în compararea fiecărei perechi de elemente adiacente, și schimbarea poziției relative a acestora în condițiile în care nu se află în relația de ordine corespunzătoare. La o trecere pentru realizarea acestor comparații a tuturor perechilor de elemente adiacente, se compară exclusiv elemente consecutive, care eventual se pot interschimba, în acest fel, elementul cel mai mare (dacă relația de ordine este $<$) ajungând pe ultima poziție a structurii prelucrate (demonstrația este simplă, prin reducere la absurd). Costul unei astfel de parcurgeri este proporțional cu lungimea listei, iar implementarea, la parcurgerea listei de intrare, poziționează elementul mai mic la ieșire, și continuă recursiv cu elementul mai mare și restul intrării.

```
%bubble1/2
%bubble1 (lista de intrare, lista de iesire)

bubble1([], []).
bubble1([H], [H]).
```

```

bubble1([H1,H2|T],[H1|R]):-
    order(H1,H2),!,
    bubble1([H2|T],R).
bubble1([H1,H2|T],[H2|R]):-
    bubble1([H1|T],R).

```

De remarcat că, datorită faptului că șablonul din intrare preia primele două elemente, condiția de terminare în lista vidă nu e suficientă: de la 2 elemente, apelul recursiv se face pe 1 element, care nu ar avea matching. Mai mult, se remarcă faptul că putem renunța la terminarea pe listă vidă, cea pe lista de 1 element fiind suficientă. Luând în considerare aceste precizări, și explicitând predicatul `order` cu `<`, predicatul se rescrie:

```

bubble1([H],[H]).
bubble1([H1,H2|T],[H1|R]):-
    H1<H2,!,
    bubble1([H2|T],R).
bubble1([H1,H2|T],[H2|R]):-
    bubble1([H1|T],R).

```

Dacă o parcurgere asigură poziționarea elementului maxim în locul corect (poziție finală), înseamnă că o nouă parcurgere îl aranjează pe al doilea cel mai mare element, și așa mai departe. Deci un număr de parcurgeri egal cu lungimea listei vor asigura ordonarea acestuia; în fapt avem nevoie de $n-1$ parcurgeri, deoarece, la a n -la parcurgere, al $n-1$ -lea cel mai mare element (în fapt al doilea cel mai mic) se poziționează corect, prin eventuala interschimbare cu cel mai mic element, care implicit ajunge și el în poziția corectă. Deci tot ce ne trebuie pentru sortare e să determinăm lungimea intrării, și să aplicăm parcurgerea un număr de ori egal cu lungimea -1. Predicatul `length/2` calculează lungimea listei de ordonat, pentru a o trimite ca parametru (valoarea decrementată cu 1) predicatului de sortare propriu zis `sort_bubble1/3`.

```

%sort_bubble1/2
%sort_bubble1(lista intrare, lista iesire)

sort_bubble1(Lin,Lout):-
    length(Lin,Length),
    actualNoOfSteps is Lenth-1,
    sort_bubble1(Lin,Lout,ActualNoOfSteps).

```

```
%sort_bubble1/3
%sort_bubble1(lista intrare, lista iesire, numar de
iteratii necesare)

sort_bubble1(Lin,Lin,0):-!.
sort_bubble1(Lin,Lout,Length):-
    NewLength is Length-1,
    bubble1(Lin,Lint),
    sort_bubble1(Lint,Lout,NewLength).
```

Cu execuția:

```
| ?- sort_bubble1([3,8,5,9,2],Result).
lista dupa inca o trecere este=[3,5,8,2,9]
lista dupa inca o trecere este=[3,5,2,8,9]
lista dupa inca o trecere este=[3,2,5,8,9]
lista dupa inca o trecere este=[2,3,5,8,9]
Result = [2,3,5,8,9] ? ;
no
```

În acest moment putem face o remarcă importantă: deși limbajele logice nu dețin ca și structuri specifice limbajelor instrucțiunile repetitive, ele dețin toate componentele necesare implementării acestora. În fapt, metoda sortării prin interschimbare prezentată mai sus se realizează prin intermediul repetării unei secvențe de un număr fix, cunoscut dinainte, de ori. Ceea ce în fapt reprezintă o buclă for, cu limita superioară lungimea listei-1, pasul decremental 1, și limita inferioară 0. Ceea ce ne permite să furnizăm șablonul general al buclei cu număr specificat de repetiții:

```
%for/3
%for (intrare, iesire, iterator)

for(In,In,0):-!.
for(In,Out,I):-
    NewI is I-1,
    do(In,Intermediate),
    for(Intermediate,Out,NewI).
```

În secvența de mai sus, do reprezintă secvența de instrucțiuni de executat în buclă (în cazul nostru parcurgerea cu interschimbare, bubble1), dar poate fi interpretat șablonul mai general, pentru orice buclă for. Condiția de terminare (clauza 1, adică faptul) trebuie să fie prima clauză, pentru a împiedica trecerea iteratorului la valori negative.

Prima implementare a sortării prin interschimbare se bazează pe observația că un număr de parcurgeri cu interschimbare egal cu lungimea listei-1 vor asigura ordonarea listei. Această observație (demonstrația riguroasă prin inducție este destul de simplă) poate fi rafinată. Am arătat că la fiecare pas (cel puțin) un element ajunge pe poziția finală (notă: faceți analogie cu sortarea prin selecție. Care sunt asemănările și deosebirile; cum comentați?). În realitate, în majoritatea cazurilor, mai mult decât un element ajunge în poziția finală. De exemplu dacă intrarea este [2,1,9,4,6,7] atunci nu doar elementul 9 dar și 6 și 7 ajung pe poziție finală. Înseamnă că în practică, după acest pas numărul de iterații rămas necesar de efectuat ar putea fi decrementat cu 3 și nu doar cu 1 așa cum precizează metodologia.

Întrebarea evidentă care urmează este: cum putem contoriza și cu cât se poate decrementa iteratorul? Mai mult, cum ne putem da seama dacă suntem într-o situație în care să-l decrementăm mai mult de 1? Răspunsul este deosebit de liniștitor: nu ne interesează! Pentru că vom aplica un alt criteriu de terminare a iterațiilor. De data aceasta nu vom mai efectua un număr fix (cunoscut apriori) de iterații, ci acest număr va fi cunoscut doar la execuție. Putem spune că vom efectua parcurgerile cu interschimbare atât timp cât va fi nevoie (deci `while` sau `repeat_until`!). De câte iterații (parcurgeri cu interschimbare) avem nevoie? Răspunsul e extrem de simplu: până când șirul e ordonat! Când e șirul ordonat? Cum putem testa asta? Această testare nu va induce un cost suplimentar (ceea ce ar face inutil câștigul obținut prin reducerea numărului de iterații)? Răspunsul la întrebarea “Când e șirul ordonat?” este surprinzător de simplu. Fiecare parcurgere realizează interschimbarea elementelor adiacente a căror poziție relativă este incorectă. Presupunând că o astfel de parcurgere NU face nici o interschimbare, rezultă imediat că TOATE perechile de elemente adiacente se află în poziție relativ corectă, de unde rezultă că șirul este ordonat! Deci avem sursa identificării terminării în chiar secvența care se execută în cadrul buclei.

Dacă în `bubble1/2` nu am efectuat nici o interschimbare, înseamnă că întreaga execuție s-a încheiat fără a se intra pe clauza 3. Cum putem captura această informație? Adăugând un nou parametru, care semnalizează trecerea prin clauzele 1 și 2 diferit de trecerea prin clauza 3 (de fapt ne interesează să capturăm trecerea cel puțin o dată prin clauza 3), vom avea practic implementată condiția (discriminantul) pentru bucla repetitivă.


```
%bubble2/3
%bubble2(lista intrare, lista iesire, conditie de
testat)

bubble2([H],[H],Test).
bubble2([H1,H2|T],[H1|R],Test):-
    H1<=H2,!,
    bubble2([H2|T],R,Test).
bubble2([H1,H2|T],[H2|R],Test):-
    Test==0,
    bubble2([H1|T],R,Test).
```

Variabila `Test` în apelul inițial este o variabilă liberă, dacă la terminarea execuției predicatului `bubble1/3` variabila este tot liberă, atunci nu s-a trecut prin clauza 3 (care ar fi instanțiat variabila), și deci lista este ordonată. Dacă la terminare are valoare 0 (singura valoare pe care o poate lua), înseamnă că cel puțin o dată s-a intrat prin clauza 3, care a produs legarea `Test=0` și deci lista nu e ordonată, cerând cel puțin o nouă parcurgere. Dacă vom captura această informație și folosi corespunzător, sortarea devine:

```
%sort_bubble2/3
%sort_bubble2(lista intrare, lista iesire, variabila de
control a iteratiilor)

sort_bubble2(Lin,Lout,Test):-
    bubble2(Lin,Lint,Test),
    nonvar(Test),!,
    sort_bubble2(Lint,Lout,NewTest).
sort_bubble2(Lin,Lin,_).
```

Predicatul funcționează astfel: atât timp cât o parcurgere a efectuat minim o interschimbare (`nonvar(Test)`), se rămâne în bucla de interschimbări (clauza 1). Altfel, execuția se încheie, prin instanțierea listei de ieșire la cea de intrare care este deja ordonată (clauza 2).

Dacă dorim ca metoda să fie o implementare pură de buclă repetitivă cu test final (de tip `repeat_until`), atunci putem modifica predicatul `bubble2/3` astfel încât să seteze simetric variabila `Test`: pe 0 în cazul cel puțin a unei interschimbări, și pe 1 în cazul în care parcurgerea nu a efectuat nici o interschimbare. În felul acesta, predicatului de sortare îi devine transparent mecanismul, semantica predicatului apelat devenind mai precisă.

În aceste condiții, tot ceea ce trebuie făcut, este să se dubleze prima clauză a predicatului `bubble2/3`, și în condițiile în care execuția s-a încheiat, variabila `Test` fiind păstrată pur variabilă logică (variabilă liberă) să o instanțieze la 1, semnalând conform convenției că "nu s-a efectuat nici o interschimbare". Cu alte cuvinte, dacă în implementarea anterioară aveam la sfârșitul execuției unui apel `bubble2/3`:

```
Test= {
    0          dacă s-a realizat minim o interschimbare
    \variabilă liberă    dacă nu s-a realizat nici o interschimbare
```

în implementarea de mai jos semantica devine:

```
Test= {
    0          dacă s-a realizat minim o interschimbare
    \1          dacă nu s-a realizat nici o interschimbare
```

```
%bubble2/3
%bubble2(lista intrare, lista iesire, conditie de
testat)
```

```
bubble2([H],[H],Test):-
    var(Test),!,
    Test=1.
bubble2([H],[H],0):-!
bubble2([H1,H2|T],[H1|R],Test):-
    H1=<H2,!,
    bubble2([H2|T],R,Test).
bubble2([H1,H2|T],[H2|R],Test):-
    Test=0,
    bubble2([H1|T],R,Test).
```

Cu acestea, noua formă pentru sortarea prin interschimbare de tip `repeat_until` devine:

```
%sort_bubble2/3
%sort_bubble2(lista intrare, lista iesire, variabila de
control a iteratiilor)

sort_bubble2(Lin,Lout,Test):-
    bubble2(Lin,Lint,Test),
    Test=0,!,
    sort_bubble2(Lint,Lout,NewTest).
sort_bubble2(Lin,Lin,1).
```

Cu o execuție care produce rezultate intermediare de forma:

```
| ?- sort_bubble2([3,8,5,9,2],Result,Flag).
lista dupa inca o trecere este=[3,5,8,2,9]
lista dupa inca o trecere este=[3,5,2,8,9]
lista dupa inca o trecere este=[3,2,5,8,9]
lista dupa inca o trecere este=[2,3,5,8,9]
lista dupa inca o trecere este=[2,3,5,8,9]
Flag = 0,
Result = [2,3,5,8,9] ? ;
no
```

Iar șablonul buclei repetitive cu test final va fi:

```
% repeat_until/3
% repeat_until(intrare, iesire, iterator)

repeat_until(In,Out,Test):-
    do(In,Intermediate,Test),
    Test=0,!,
    repeat_until (Lint,Lout,NewTest).
repeat_until(In,In,1).
```

În secvența de mai sus, `do` reprezintă secvența de instrucțiuni de executat în buclă (în cazul nostru parcurgerea cu interschimbare, `bubble2`), dar poate fi interpretat șablonul mai general, pentru orice buclă `for`.

Remarcă importantă: dacă pentru bucla `for/3` condiția de terminare (faptul) trebuia să fie pe prima poziție (clauza 1) pentru `repeat_until/3` această condiție trebuie să fie ultima (clauza a doua), în caz contrar execuția terminându-se chiar înainte de prima parcurgere, instanțiind ieșirea la intrare și variabila de test la 1.

Se știe că o buclă cu test final (de forma `repeat_until`) poate fi transformată într-una cu test inițial (de forma `while_do`). Cât de ușor se produce această transformare pentru sortarea prin interschimbare? Rămâne la latitudinea cititorului să o implementeze.

Până acum am reprezentat diferite metode, toate accesibile și limbajelor imperative. Ceea ce înseamnă că de la specificațiile Prolog, după testare, implementarea într-un limbaj imperativ ar fi imediată, nemaissolicitând demonstrarea de corectitudine (în cazul unei implementări 1 la 1).

În continuare vom prezenta o metodă caracteristică limbajelor logice, prin aceea că, rezolvarea problemei face apel la unul din mecanismele specifice programării logice, și anume nedeterminismul.

Haideți să ne reamintim predicatul de concatenare a două liste, și anume varianta sa nedeterministă.

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

Unei întrebări nedeterminate, variantele de răspuns vor fi:

```
| ?- append(X, Y, [1,2,3,4]) .
X = [], Y = [1,2,3,4] ? ;
X = [1], Y = [2,3,4] ? ;
X = [1,2], Y = [3,4] ? ;
X = [1,2,3], Y = [4] ? ;
X = [1,2,3,4], Y = [] ? ;
no
```

Cum putem folosi?

Ce se întâmplă dacă argumentul instanțiat este lista de ordonat, și mai adăugăm o comparație?

```
| ?- append(X, [A,B|Y], [3,8,5,9,2]), A>B, ! .
```

Apelul va selecta în fapt prima pereche (de la stânga la dreapta) de elemente care nu se află în relație de ordine nedescrescătoare. În cazul nostru va produce instanțierea $X = [3]$, $Y = [9, 2]$, $A=8$, $B=5$.

Dacă în acest moment reconcatenăm listele, și interschimbând A cu B , se obține:

```
| ?- append(X, [B,A|Y], L)
L=[3,5,8,9,2]
```

Legând cele două apeluri:

```
| ?- append(X, [A,B|Y], [3,8,5,9,2]),
    A>B, !, append(X, [B,A|Y], L) .
```

Obținem un efect similar cu cel obținut de parcurgerea cu interschimbare, doar că de data aceasta, parcurgerea explicită este înlocuită de cea implicită, obținută prin nedeterminism. Și deci o altă variantă a sortării va fi:

```
%bubble2/2
%bubble2(lista intrare, lista iesire)
```

```
sort_bubble3(Lin,Lout):-
    append(X,[A,B|Y],Lin),
    A>B,
    append(X,[B,A|Y],Lint),!,
    sort_bubble3(Lint,Lout).
sort_bubble3(Lin,Lin).
```

Eleganța și simplitatea metodei sunt remarcabile! Efortul programatorului este minim. Totul este lăsat în seama mecanismelor programării logice. Rămâne pentru studiul cititorului calculul efortului computațional pe care-l presupune implementarea.

```
| ?- sort_bubble3([3,8,5,9,2],Result).
lista dupa inca o interschimbare este=[3,5,8,9,2]
lista dupa inca o interschimbare este=[3,5,8,2,9]
lista dupa inca o interschimbare este=[3,5,2,8,9]
lista dupa inca o interschimbare este=[3,2,5,8,9]
lista dupa inca o interschimbare este=[2,3,5,8,9]
Result = [2,3,5,8,9] ? ;
no
|
```

O proprietate importantă, a tuturor metodelor analizate până în acest moment, și deseori cerută prin specificația problemei, este *stabilitatea* algoritmilor. Prin stabilitate se înțelege păstrarea aceleiași poziții relative pentru elementele de valori egale în lista de ieșire ca și în lista de intrare.

5.5 Exerciții și probleme propuse

1. Ce soluții și în ce ordine se obțin pentru predicatul `permutation1/2` și întrebarea:

```
| ?- permutation1([1,2,3],Permutation).
```

în cazul în care se inversează ordinea clauzelor (faptul devine clauza a doua).

2. Ce soluții și în ce ordine se obțin pentru predicatul `append/3` și întrebarea:

```
| ?- append(X,Z,[1,2,3,4]).
```

3. Ce soluții și în ce ordine se obțin pentru predicatul `append/3` și întrebarea:

```
| ?- append(_,[A|_],[1,2,3,4]).
```

4. Ce soluții și în ce ordine se obțin pentru predicatul `append/3` și întrebarea:

```
| ?- append(X,[A|Y],[1,2,3,4]).
```

5. Ce soluții și în ce ordine se obțin pentru predicatul `permutation2/2` și întrebarea:

```
| ?- permutation2([1,2,3],Permutation).
```

în cazul în care se inversează ordinea clauzelor (faptul devine clauza a doua).

6. Ce implicații are adăugarea tăierii de backtracking după comparația `H<M` în a doua clauză a predicatului `min1/3`. Justificați.

7. Ce implicații are trecerea faptului ca ultimă clauză a predicatului `min1/3`. Justificați. Cum este afectată execuția de această modificare?

8. Cum afectează execuția schimbarea ordinii clauzelor în forma finală a predicatului `min2/2`. Justificați.

9. Precizați rezultatul obținut pentru predicatul `sort_sel1/2` și întrebarea

```
| ?- sort_sel1([4,7,9,3,5,2,1],Result).
```

Care sunt rezultatele intermediare obținute?

10. Precizați rezultatul obținut pentru predicatul `sort_sel2/2` și întrebarea

```
| ?- sort_sel2([4,7,9,3,5,2,1],Result).
```

Care sunt rezultatele intermediare obținute?

11. Precizați rezultatul obținut pentru predicatul `sort_ins1/2` și întrebarea

```
| ?- sort_ins1([4,7,9,3,5,2,1],Result).
```

Care sunt rezultatele intermediare obținute?

12. Precizați rezultatul obținut pentru predicatul `sort_ins2/2` și întrebarea

```
| ?- sort_ins2([4,7,9,3,5,2,1],Result).
```

Care sunt rezultatele intermediare obținute.

13. Precizați rezultatul obținut pentru predicatul `sort_bubble1/2` și întrebarea

```
| ?- sort_bubble1([4,7,9,3,5,2,1],Result).
```

Care sunt rezultatele intermediare obținute?

14. Precizați rezultatul obținut pentru predicatul `sort_bubble2/3` și întrebarea

```
| ?- sort_bubble2([4,7,9,3,5,2,1],Result,Flag).
```

Care sunt rezultatele intermediare obținute.

15. Precizați rezultatul obținut pentru predicatul `sort_bubble3/2` și întrebarea

```
| ?- sort_bubble3([4,7,9,3,5,2,1],Result).
```

Care sunt rezultatele intermediare obținute?

16. Analizați asemănările și deosebirile dintre implementările sortării prin interschimbare cu buclă `for`, respectiv `repeat_until`. Care se pretează mai bine la problema în cauză?

17. Implementați în Prolog o buclă de tip `while`. Care este diferența față de `repeat_until`? Cum s-ar implementa sortarea prin interschimbare cu acest nou tip de buclă?

6. Structuri de date în Prolog

6.1 Arbori

Structurile de date disponibile în limbajele imperative pot fi manipulate și în limbajele logice. Acest lucru este important dacă avem în vedere că problemele complexe necesită în majoritatea situațiilor elaborarea și manevrarea unor structuri complexe. La primul nivel de complexitate s-ar situa arborii, a căror implementare este relativ simplă. Arborii ca obiecte compuse și recursive, iar informația minimală prezentă la nivelul unui nod este cheia de identificare a nodului și legătura (oricare ar fi reprezentarea acesteia) către nodurile fiu (oricâți fii ar avea). Vom începe, și de fapt ne vom concentra majoritatea atenției asupra arborilor binari (cu doi fii). Deoarece Prologul e limbaj netipizat, modalitatea de specificare a arborilor ține strict de programator. Vom trece în revistă operațiile elementare pe arbori (inserare, ștergere, căutare, traversare), și vom discuta aceste aspecte pe diferite tipuri particulare de arbori binari, după care vom sublinia aspectele definitorii în cazul trecerii de la arbori binari la arbori ternari sau multicăi. De asemenea, aceste aspecte vor lua în considerare atât arborii compleți cât și arborii terminați în variabile (incompleți).

Câteva remarci preliminare:

- deși nu există un cuvânt rezervat, se obișnuiește ca arborele vid să se reprezinte ca `nil` (deși nu este singura notăție utilizată, de-a lungul acestui text o vom păstra cu consecvență),
- pentru reprezentarea unui nod vom folosi reprezentarea în inordine a elementelor unui nod: `t(fiu stânga, rădăcină, fiu dreapta)`.

Arbori binari de căutare

Vom începe prin prezentarea operațiilor elementare aplicate asupra arborilor binari de căutare. Arborii binari de căutare sunt arbori binari care au proprietatea că orice cheie din subarborele stâng este mai mică decât cheia din nodul rădăcină, care la rândul său este mai mică decât orice cheie din subarborele drept. Proprietatea este recursivă (orice subarbore a unui arbore binar de căutare e un arbore binar de căutare), ceea ce face ca aceste structuri

să fie ordonate. Termenul în limba engleză pentru aceste structuri este "Binary Search Tree", de aceea, acronimul încetățenit în literatura de specialitate pentru aceste structuri este de *BST*.

6.1.1 Căutarea

```
%search_bst/2
%search_bst(cheie_de_cautat,arbore_sursa).

search_bst(Key,nil):-!,
    write("nu exista in arbore").
search_bst(Key,t(Left,Key,Right)):-!,
    write("am ajuns la cheia cautata").
search_bst(Key,t(Left,Key1,Right)):-
    Key<Key1,!,search_bst(Key,Left).
search_bst(Key,t(Left,Key1,Right)):-
    search_bst(Key,Right).
```

Predicatul are două condiții de terminare: cea în care găsește cheia (clauza 2), și eventual în acest moment o prelucrare (a nodului, sau a unuia din subarbori ar putea fi necesară) ar putea fi solicitată (prin apelul unui alt predicat care să facă transformarea substructurii de interes), și cea care ajunge la *nil* fără să găsească cheia (clauza 1), caz în care mesajul corespunzător este semnalat. În funcție de specificațiile concrete ale problemei, această clauză s-ar putea termina cu eșec (cazul în care se dorește ca negăsirea cheii să provoace backtracking în secvența apelantă a *search_bst*). În acest caz, clauza ar avea forma:

```
search_bst(Key,nil):-!,
    write("nu exista in arbore"),!,fail.
```

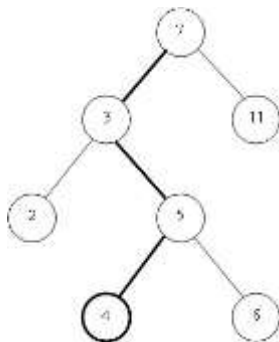


Fig. 6.1

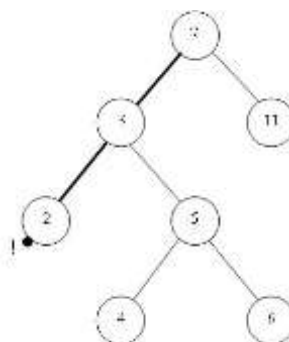


Fig. 6.2

Celelalte două clauze decid continuarea căutării cheii fie în subarboarele stânga (clauza 3) respectiv dreapta (clauza 4), în funcție de rezultatul (determinist) al comparației $Key < Key1, !$.

Așa cum ne așteptam, căutarea în BST se face pe o singură ramură, ordinul de mărime al operației fiind $O(h)$, unde h este înălțimea BST. Această înălțime are o valoare $\lg n \leq h \leq n$ unde n reprezintă numărul de noduri din arbore. Justificați ($\lg n \leq h \leq n$) această afirmație.

6.1.2 Inserarea

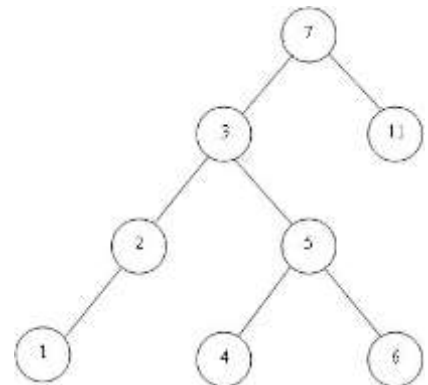
```
%insert_bst/3
%insert_bst(cheie_de_inserat,
            arbore_sursa,
            arbore_rezultat).

insert_bst(Key, nil, t(nil, Key, nil)) :- !.
insert_bst(Key, t(Left, Key, Right), t(Left, Key, Right)) :- !,
    write("exista in arbore").
insert_bst(Key, t(Left, Key1, Right), t(NewLeft, Key, Right)) :-
    Key < Key1, !, insert_bst(Key, Left, NewLeft).
insert_bst(Key, t(Left, Key1, Right), t(Left, Key, NewRight)) :-
    insert_bst(Key, Right, NewRight).
```

Fig. 6.3

Interpretarea clauzelor este următoarea:

- inserarea unui nod într-un arbore vid reprezintă frunza având cheia dată (clauza primă);
- inserarea unui nod într-un arbore ce conține în nodul rădăcină cheia de inserat returnează la ieșire arborele de intrare, cu transmiterea mesajului că nu s-a făcut inserare propriu-zisă (clauza secundă);
- inserarea unui nod într-un arbore ce are în rădăcină o cheie mai mare decât cheia de inserat este un arbore rezultat în urma inserării nodului în subarboarele stâng al arborelui sursă;



- Dacă nici una din alternativele anterioare nu este îndeplinită (existența tăierii în corpul clauzelor 1, 2, 3 ne scutește de a mai verifica $Key > Key1$) arborele rezultat se obține prin inserarea în subarborele drept și copierea rădăcinii și a subarborului stâng.

Observații:

1. Prin specificitatea limbajelor logice (odată ce o variabilă a fost instanțiată, valoarea ei poate fi schimbată doar prin provocarea eșecului, care conduce implicit și la pierderea valorii) la operațiile de inserare/ștergere în structuri complete (vom vedea excepții, la structurile terminate în variabile, la capitolul corespunzător) structura de ieșire trebuie să fie una diferită de cea de intrare (nu avem la dispoziție argumentele de tip intrare/ieșire). Aceasta înseamnă că trebuie folosit un parametru suplimentar pentru arborele de ieșire.
2. Se observă că operația de inserare poate fi descompusă în două secvențe de sine stătătoare: căutarea poziției de inserat, și respectiv inserarea propriu-zisă.
3. Faza de căutare urmează identic strategia din operația de căutare. Singura deosebire față de operația propriu-zisă de căutare este aceea că se mai crează o structură (arborele de ieșire). Arborele de ieșire are structura identică cu cel de intrare cu excepția ramurii pe care se produce inserarea. Se observă că în fiecare clauză cel puțin un subarbor de ieșire este chiar cel de intrare (unificare implicită prin utilizarea aceluiași nume de variabilă). Ramura care se modifică are un nume diferit, modificarea producându-se în momentul adăugării frunzei, în clauza 1.
4. Inserarea se face **totdeauna** ca frunză. De altfel proprietatea e chiar mai generală, în orice fel de arbori, crearea unui nou nod presupune totdeauna formarea unei noi frunze, chiar dacă într-o fază următoare se realizează transformări care schimbă poziția nodului)
5. Timpul necesar efectuării acestei operații este tot $O(h)$.

6.1.3 Ștergerea

Operația este (aparent) mai dificilă, impunând abordarea unei strategii. Dificultatea constă în aceea că dacă la un BST inserarea o putem face totdeauna ca frunză (orice cheie dorim să adăugăm, se găsește ramura de urmat pentru a ajunge la poziția-frunză în care se adaugă elementul) ștergerea trebuie să elimine chiar elementul specificat, indiferent de poziția acestuia. Înainte de a intra în detalii de implementare, să analizăm problema.

Presupunând că există în arbore un nod conținând cheia pe care dorim s-o ștergem, acesta s-ar putea afla într-una din următoarele situații:

- frunză,
- nod cu un singur succesor (nil pe stânga sau pe dreapta, dar nu în ambele părți),
- nod cu doi succesori (vom denumi acest caz eliminarea nodului rădăcină; vom justifica imediat faptul că ștergerea unui nod cu doi succesori se reduce la ștergerea nodului rădăcină).

Vom trata cazurile de la simplu la complex:

Eliminare *frunză*: (Fig. 6.4, 6.5) deoarece nici un nod nu este legat de o frunză, aceasta se poate pur și simplu elimina, fără a afecta structura în nici un fel (în arborele de ieșire, locul frunzei din intrare este luat de un nod nil). Operația (după ajungerea la nod) durează $O(1)$.

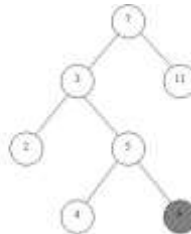
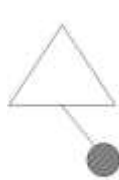


Fig. 6.4

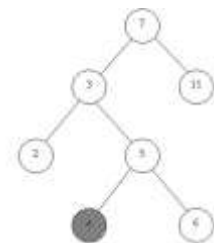


Fig. 6.5

Eliminare *nod cu un singur succesor*: (Fig. 6.6, 6.7) un astfel de nod are o singură legătură spre descendenți. Nodul poate fi eliminat prin șuntarea nodului, adică singurul său descendent este redirecționat spre nodul părinte al nodului eliminat. Prin această operație relația de căutare din BST se menține consistentă. Operația (după ajungerea la nod) durează $O(1)$.

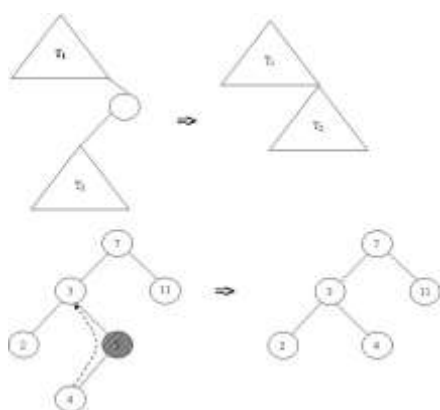


Fig. 6.6

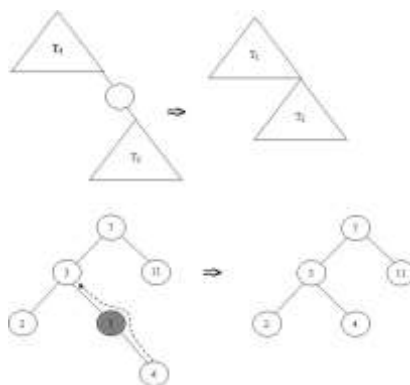


Fig. 6.7

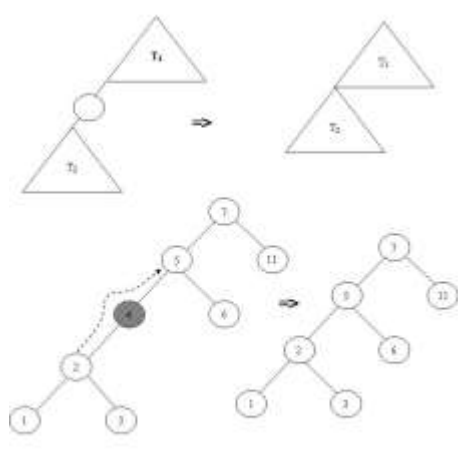


Fig. 6.8

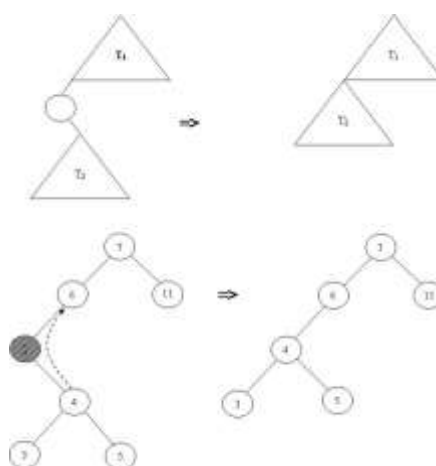


Fig. 6.9

Eliminare nod cu doi succesori: (Fig. 6.8, 6.9) deși aparent dificilă, vom arăta că operația se reduce la una din cele anterioare. Pentru ajungerea la nodul cu doi succesori de șters se face mai întâi o căutare a nodului. După ce s-a ajuns la nod, acesta este rădăcina unui întreg subarboare (în cazul cel mai simplu având doar alte două noduri, fiii săi). De aceea vom considera problema ștergerii rădăcinii unui BST. Dificultatea problemei constă în aceea că arborele trebuie menținut BST. Vom identifica în continuare opțiunile disponibile, pornind de la următoarea analiză:

La parcurgerea în inordine a arborelui, cheile apar ordonat crescător (în paranteză se specifică un subarbore, conferind semantica recursivă relației de inegalitate).

$(Left) < Key < (Right)$

Dacă se dorește ștergerea Key , parcurgerea în inordine a arborelui rezultat va avea forma:

$(Left) < (Right)$

Variantele posibile sunt:

- unul dintre subarbori devine arbore de ieșire, de care se "agață" celălalt arbore;
- se elimină rădăcina, așa cum sugerează relația de inegalitate.

Fiecare din cele două variante are avantaje și dezavantaje. Prima, simplă și intuitivă (deși vom vedea că nu mai puțin costisitoare) are dezavantajul degenerării arborelui (ștergeri succesive pe aceeași ramură apropie arborele de o listă, ducând la creșterea timpului de căutare, și implicit a tuturor celorlalte operații în arbore). A doua, este (aparent) mai complexă. Are avantajul păstrării structurii de arbore.

Fiecare din cele 2 variante au la rândul lor 2 moduri de realizare (datorate dualității stânga/dreapta).

Varianta 1:

- $Left$ devine arborele principal: (Fig. 6.10) trebuie adăugat $Right$ păstrând structura BST. Aceasta se face inserând (de fapt legând) rădăcina $Right$ pe poziția corespunzătoare BST în arborele $Left$. Datorită relației $(Left) < (Right)$ avem că $\forall Key1 \in Left, \text{și } \forall Key2 \in Right, Key1 < Key2$, în particular, $RootRight$ are aceeași proprietate (unde $RootRight$ este rădăcina subarborelui dreapta), și deci: $\forall Key1 \in Left, Key1 < RootRight$, ceea ce înseamnă că rădăcina subarborelui stânga este mai mare decât orice cheie din subarborele dreapta. Deci adăugarea rădăcinii (împreună cu întregul subarbore) presupune adăugarea ca fiu al celui mai mare nod

din stânga. Acesta (fiind cel mai mare) este ultimul la parcurgerea în inordine a subarborelui stâng, și deci are proprietatea că e cel mai din dreapta nod din subarboarele stânga (deci are maxim 1 succesor – spre stânga – iar `RootRight` se va adăuga spre dreapta, deci respectă condiția de inserare frunză!). Metoda este simplă, dar (paradoxal) din pacate crește înălțimea arborelui. Dacă înainte de ștergere aveam:

$$h = \max(h_{\text{Left}}, h_{\text{Right}}) + 1$$

după ștergere, înălțimea devine:

$$h = h_{\text{Left}} + h_{\text{Right}}$$

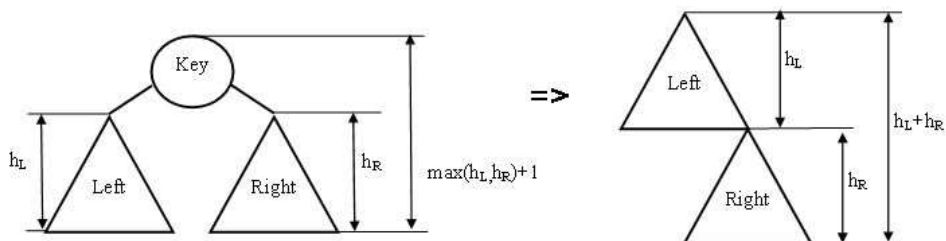


Fig. 6.10

- simetric, `Right` (Fig. 6.11) devine arborele principal: trebuie adăugat `Left` păstrând structura BST. Aceasta se face inserând (de fapt legând) rădăcina `Left` pe poziția corespunzătoare BST în arborele `Right`. Adăugarea rădăcinii (împreună cu întregul subarbor) presupune adăugarea ca fiu al celui mai mic nod din dreapta. Acesta (fiind cel mai mic) este primul la parcurgerea în inordine a subarborelui drept, și deci are proprietatea că e cel mai din stânga nod din subarboarele dreapta (deci are maxim 1 succesor – spre dreapta – iar `RootLeft` se va adăuga spre stânga, deci respectă condiția de inserare frunză!). Tehnica fiind simetrică prezintă același dezavantaj al creșterii înălțimii.

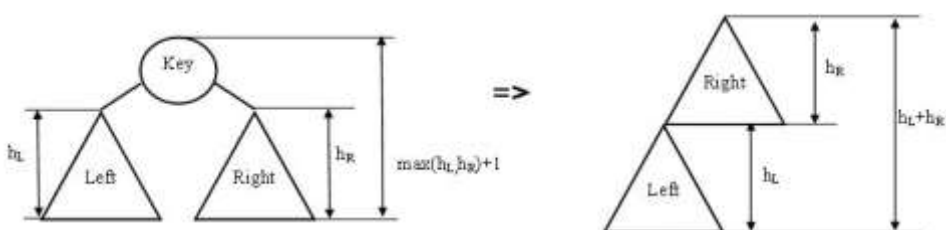


Fig. 6.11

Varianta 2: datorită simetriei structurii ne așteptăm să avem și de această dată două versiuni. Vom prezenta în detaliu una dintre ele, celaltă rămânând la latitudinea cititorului să o realizeze.

Relația de ordine în arborele din care trebuie eliminată rădăcina este:

$$(\text{Left}) < \text{Key} < (\text{Right})$$

Descompunem subarboarele stâng și drept evidențiind cea mai mare cheie din subarboarele stâng (notată MaxLeft) și respectiv cea mai mică cheie din subarboarele drept (notată MinRight) (Fig. 6.12, 6.13). Aceste chei au proprietatea că preced, respectiv succed cheia din rădăcină la parcurgerea în inordine. Dacă notăm:

$$\text{Left}' = \text{Left} \setminus \{\text{MaxLeft}\} \text{ și } \text{Right}' = \text{Right} \setminus \{\text{MinRight}\}$$

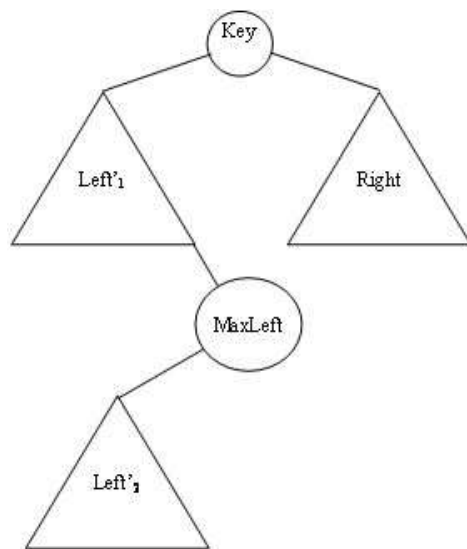


Fig. 6.12

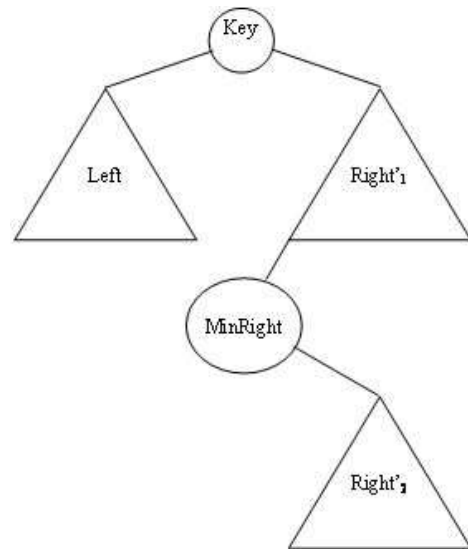


Fig. 6.13

Relația de ordine se rescrie:

$$(\text{Left}') < \text{MaxLeft} < \text{Key} < \text{MinRight} < (\text{Right}')$$

Eliminarea rădăcinii presupune obținerea relației:

$$(\text{Left}') < \text{MaxLeft} < \text{MinRight} < (\text{Right}')$$

Ceea ce înseamnă că lipsește rădăcina. Dar am putea schimba strategia ștergerii rădăcinii cu înlocuirea sa cu elementul predecesor sau succesor, și eliminarea fizică (ca nod din arbore) a respectivului nod. Se simplifică problema? Cu alte cuvinte: este eliminarea fizică a nodului care ocupă poziția în care se află cheia `MaxLeft` sau `MinRight` într-o situație mai favorabilă decât nodul rădăcină `Key` ?

`MaxLeft` este cheia care se află în ultimul nod la parcurgerea în inordine a subarborelui `Left`. Dacă aceasta este situația, atunci `MaxLeft` NU are succesor spre dreapta (sau de loc), ceea ce înseamnă că e fie nod cu un singur succesor, fie o frunză (ambele aceste situații au fost analizate, și se rezolvă mai ușor). Simetric `MinRight` este fie frunză, fie nod cu un singur succesor, spre stânga.

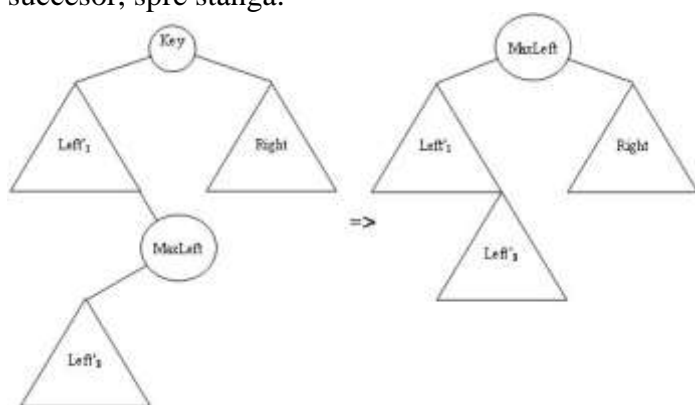


Fig. 6.14

Cu aceste observații, strategia în cazul ștergerii nodului rădăcină este următoarea (Fig. 6.14):

- se determină nodul cel mai din dreapta (respectiv stânga) din subarborele stâng (respectiv drept). Cheia acestuia este `MaxLeft` (respectiv `MinRight`);
- se elimină fizic nodul conținând cheia `MaxLeft` (respectiv `MinRight`). Eliminarea se face în conformitate cu strategia de la eliminarea frunzelor sau a nodurilor cu un singur succesor;
- cheia din nodul eliminat fizic (eventual împreună cu toate informațiile suplimentare memorate în nod) se mută în rădăcină, în locul cheii `Key` (și eventual în locul tuturor informațiilor suplimentare memorate în nod).

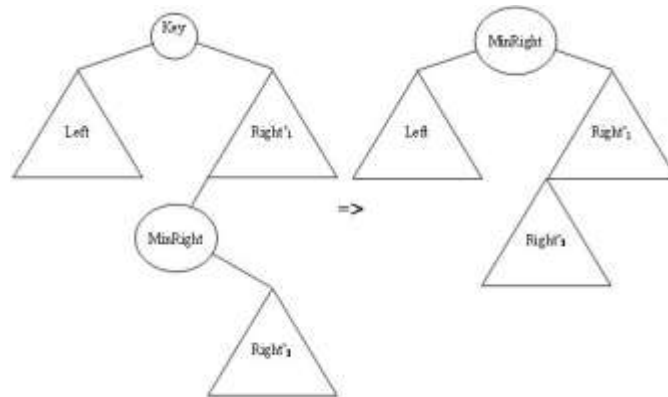


Fig. 6.15

Se observă că în această situație timpul necesar efectuării ștergerii este $O(h)$ și este determinat de timpul necesar căutării maximumului pe dreapta sau minimumului pe stânga. Restul operațiilor (înlocuire nod, re poziționare legături tată-fiu) se efectuează în $O(h)$. Un alt avantaj al acestei metode este că arborele nu degenerază (nu crește în înălțime). În urma ștergerii unei chei, înălțimea arborelui de ieșire este fie egală cu a celui de intrare, fie mai mică cu 1 unitate.

Acum putem trece la secvența Prolog care realizează transformările menționate. Ștergerea presupune două faze: identificarea cheii de șters (secvență de căutare) și ștergerea propriu-zisă, în conformitate cu una din cele 3 situații posibile (frunză, un succesor, rădăcină). Am menționat deja că pentru ștergerea rădăcinii avem patru alternative. Noi vom implementa varianta cu înlocuirea rădăcinii cu `MaxLeft` rămânând la latitudinea cititorului implementarea celorlalte trei.

```
% delete_bst/3
% delete_bst(cheie_de_sters, arbore_sursa,
arbore_rezultat)

delete_bst(Key,nil,nil):-!,
    write(Key),
    write(' nu exista in arbore').
delete_bst(Key,t(nil,Key,nil),nil):-!.
delete_bst(Key,t(nil,Key,Right),Right):-!.
delete_bst(Key,t(Left,Key,nil),Left):-!.
delete_bst(Key,t(Left,Key,Right),t(NewLeft,MaxLeft,Right)):-!,
    deleteMaxFromTree(Left,MaxLeft,NewLeft).
delete_bst(Key,t(Left,Key1,Right),t(NewLeft,Key1,Right)):-
```

```

    Key<Key1,!,
    delete_bst(Key,Left,NewLeft).
delete_bst(Key,t(Left,Key1,Right),t(Left,Key1,NewRight)):-
    delete_bst(Key,Right,NewRight).

%deleteMaxFromTree/3
%deleteMaxFromTree(arbore_sursa,
                    cheie_maxima_din_arbore,
                    arbore_rezultat)

deleteMaxFromTree(t(Left,MaxKey,nil),MaxKey,Left):-!.
deleteMaxFromTree(t(Left,Key,Right),MaxKey,t(Left,Key,NewRight)):-
    deleteMaxFromTree(Right,MaxKey,NewRight).

```

Predicatul de ștergere `delete_bst/3` realizează prin ultimele 2 clauze (6 și 7) căutarea nodului conținând cheia de șters (și aceste clauze urmează întocmai tipicul din `search_bst/2`). Celelalte clauze sunt specifice ștergerii proprii-zise, după cum urmează:

- clauza 1 corespunde situației în care se ajunge (în faza de căutare) la `nil` fără a găsi cheia căutată, situație în care se returnează mesajul de eroare, iar arborele de ieșire rămâne cel de intrare nemodificat (în funcție de interes, anumite variante ar putea provoca eșecul în acest moment);
- clauza 2 corespunde situației de ștergere a unei frunze, caz în care frunza din intrare este înlocuită cu `nil` la ieșire;
- clauza 3 (respectiv 4) corespunde situației în care nodul de șters are doar succesori spre dreapta (respectiv stânga). Ștergerea nodului se realizează prin redirecționarea unicului fiu din dreapta (respectiv stânga) spre nodul părinte al nodului de șters. Acest lucru este implementat prin setarea ieșirii la `Right` (respectiv `Left`);
- clauza 5 corespunde ștergerii rădăcinii. Se realizează prin înlocuirea rădăcinii cu cheia cea mai mare, mai mică decât rădăcina (micșorând subarborele stâng, dreptul nemodificat). Acest mecanism este implementat prin `deleteMaxFromTree/3`.

Predicatul `deleteMaxFromTree/3` produce o "alunecare" (clauza 2) pe ramura cea mai din dreapta a subarborului stâng, până la cheia maximă din subarbore. Această clauză implementează în fapt o buclă `while` (așa cum am arătat și la sortarea prin interschimbare), de forma:

```

while(subarbore dreapta nevid)
    do avansează spre dreapta;

```

Prima clauză a predicatului (de terminare) corespunde identificării cheii maxime, returnării acesteia (parametrul 2) și a întregului subarbore din care nodul șuntat (cu fiu doar spre stânga) lipsește.

La o analiză mai atentă a predicatului `delete_bst/3` se poate observa că a doua clauză este un caz particular al clauzelor 3 și 4 (o frunză reprezintă în fapt și un nod cu un succesor, acesta fiind `nil`). Deci această clauză poate lipsi, `nil` suprapunându-se cu succes peste `Right` (respectiv `Left`) în clauzele 3 (sau 4). Renunțând la această clauză predicatul devine:

```
delete_bst(Key, nil, nil) :- !,
    write(Key),
    write(' nu exista in arbore').
delete_bst(Key, t(nil, Key, Right), Right) :- !.
delete_bst(Key, t(Left, Key, nil), Left) :- !.
delete_bst(Key, t(Left, Key, Right), t(NewLeft, MaxLeft, Right)) :-
    !, deleteMaxFromTree(Left, MaxLeft, NewLeft).
delete_bst(Key, t(Left, Key1, Right), t(NewLeft, Key1, Right)) :-
    Key < Key1, !,
    delete_bst(Key, Left, NewLeft).
delete_bst(Key, t(Left, Key1, Right), t(Left, Key1, NewRight)) :-
    delete_bst(Key, Right, NewRight).
```

Operația de ștergere are timpul de execuție $O(h)$: secvența de căutare are maxim $O(h)$ iar ștergerea propriu zisă în cazul cel mai defavorabil este tot $O(h)$. Aparent avem o constantă multiplicativă 2 (clasa algoritmului rămânând oricum liniară); în realitate nu este 2 deoarece nu pot apărea simultan cazul cel mai defavorabil atât în secvența de căutare cât și în cea de ștergere. Dacă căutarea durează $O(h)$ înseamnă că am găsit o frunză și atunci ștergerea durează $O(1)$. Dacă dimpotrivă, ștergerea durează $O(h)$ înseamnă că ștergem chiar rădăcina arborelui, și atunci căutarea durează $O(1)$. Într-un caz intermediar, când căutarea durează $O(h_1)$, cu $h_1 < h$, și ștergem un nod cu doi succesori, ștergerea propriu-zisă pornește la căutarea nodului care înlocuiește rădăcina chiar de la nivelul h_1 , și deci numărul de pași maxim va fi $h - h_1$. Deci în total $O(h_1)$ căutarea și $O(h - h_1)$ ștergerea, adică exact $O(h)$.

Putem concluziona că toate trei operațiile (căutare, inserare, ștergere) într-un arbore binar de căutare durează $O(h)$, iar $\lg n \leq h \leq n$.

Urmărim în continuare câteva operații succesive de inserare și ștergere, cu tipărirea arborilor rezultați, folosind următoarele predicate auxiliare pentru tipărire:

```
writeTree(Tree):-
    writeTree(0,Tree).
```

Tipărește un arbore, cu nivelul de indentare inițial precizat ca prim argument.

```
%writeTree/2
%writeTree(nivel,arbore)

writeTree(_,nil):-!.
writeTree(Level,t(Left,Key, Right)):-
    LevelPlus1 is Level + 1,
    writeTree(LevelPlus1,Right),nl,
    writeSpaces(Level),write('['),
    write(Key),write(']'),
    writeTree(LevelPlus1,Left).
```

Tipărește N·4 caractere spațiu.

```
%writeSpaces/1
%writeSpaces(integer)

writeSpaces(0):-!.
writeSpaces(N):-
    write('    '),
    NMinus1 is N - 1,
    writeSpaces(NMinus1).

| ?- nl,write(`Insert 7 :'),
    insert_bst(7,nil,T1),writeTree(T1),nl,
    write(`Insert 11 :'),
    insert_bst(11,T1,T2),writeTree(T2),nl,
    write(`Insert 3 :'),
    insert_bst(3,T2,T3),writeTree(T3),nl,
    write(`Insert 5 :'),
    insert_bst(5,T3,T4),writeTree(T4),nl,
    write(`Insert 2 :'),
    insert_bst(2,T4,T5),writeTree(T5),nl,
    write(`Insert 4 :'),
    insert_bst(4,T5,T6),writeTree(T6),nl,
```

```

write('Insert 3 :'),
insert_bst(3,T6,T7),writeTree(T7),nl,
write('Insert 6 :'),
insert_bst(6,T7,T8),writeTree(T8),nl,
write('Delete 7 :'),
delete_bst(7,T8,T9),writeTree(T9),nl,
write('Delete 13 :'),
delete_bst(13,T9,T10),writeTree(T10),nl,
write('Delete 5 :'),
delete_bst(5,T10,T11),writeTree(T11),nl,
write('Delete 11 :'),
delete_bst(11,T11,T12),writeTree(T12).

```

```

Insert 7 :
[7]
Insert 11 :
  [11]
[7]
Insert 3 :
  [11]
[7]
  [3]
Insert 5 :
  [11]
[7]
    [5]
    [3]
Insert 2 :
  [11]
[7]
    [5]
    [3]
    [2]
Insert 4 :
  [11]
[7]
    [5]
      [4]
    [3]
      [2]
Insert 3 :3 exista in arbore
  [11]
[7]
    [5]
      [4]

```

```

    [3]
      [2]
Insert 6 :
    [11]
  [7]
    [6]
      [5]
        [4]
          [3]
            [2]
Delete 7 :
    [11]
  [6]
    [5]
      [4]
        [3]
          [2]
Delete 13 :13 nu exista in arbore'
    [11]
  [6]
    [5]
      [4]
        [3]
          [2]
Delete 5 :
    [11]
  [6]
    [4]
      [3]
        [2]
Delete 11 :
    [6]
      [4]
        [3]
          [2]
T1 = t(nil,7,nil),
T2 = t(nil,7,t(nil,11,nil)),
T3 = t(t(nil,3,nil), 7,t(nil,11,nil)),
T4 = t(t(nil,3,t(nil,5,nil)),7,t(nil,11,nil)),
T5 = t(t(t(nil,2,nil),3,t(nil,5,nil)),
7,t(nil,11,nil)),
T6=t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,nil)),7,t(nil,11,nil)),
T7=t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,nil)),7,t(nil,11,nil)),

```

```

T8 =
t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),7,t(nil,11,
nil)),
T9=t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,nil)),6,t(nil,11,nil)),
T10=t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,nil)),7,t(nil,11,nil)),
T11 = t(t(t(nil,2,nil),3,t(nil,4,nil)),6,t(nil,11,nil)),
T12 = t(t(t(nil,2,nil),3,t(nil,4,nil)),6,nil) ?
yes

```

6.1.4 Traversare

```

%traverse/1
%traverse(arbore_sursa)

traverse(nil).
traverse(t(Left,Key,Right)):-
    write("Key"),
    traverse(Left),
    write("Key"),
    traverse(Right),
    write("Key").

```

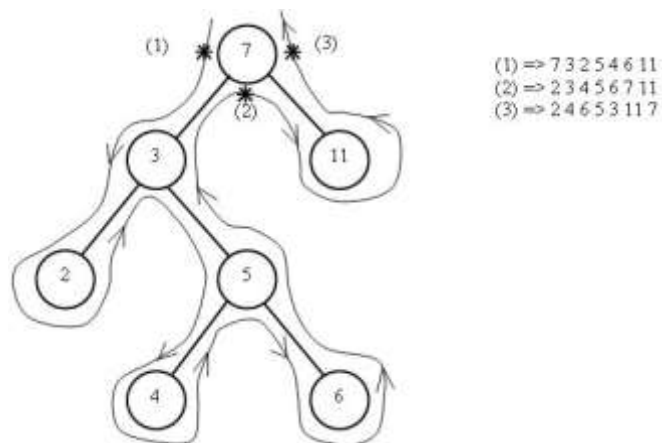


Fig. 6.16

Predicatul prezentat mai sus este unul generic, de traversare (Fig. 6.16) a unui arbore binar. De menționat că în funcție de tipul traversării (pre, in sau postordine) doar una din cele trei secvențe de tipărire a cheii din nodul rădăcină este prezentă. Mai mult, în funcție de problema concretă, tipărirea poate fi înlocuită cu o operație complexă, de prelucrare a cheii (sau a unei

structuri din arbore). Prezintă secvența generică pentru fiecare tip de traversare:

```

traverse_pre(nil).
traverse_pre(t(Left,Key,Right)):-
    do(Key),
    traverse_pre(Left),
    traverse_pre(Right).

traverse_in(nil).
traverse_in(t(Left,Key,Right)):-
    traverse_in(Left),
    do(Key),
    traverse_in(Right).

traverse_post(nil).
traverse_post(t(Left,Key,Right)):-
    traverse_post(Left),
    traverse_post(Right),
    do(Key).

```

Operația de traversare este liniară în numărul de noduri din arbore, $O(n)$ (cu condiția ca durata lui `do` să fie constantă). Exemplificăm în continuare traversarea pe o problemă concretă, justificând tipul traversării ales, și încercând să menținem limite liniare pentru ordinul de mărime.

Dorim să scriem un predicat de calculare a înălțimii unui BST (faptul că arborele este de căutare nu are nici o relevanță în această problemă).

Calcularea înălțimii (Fig. 6.17) trebuie să implementeze relația:

$$h = \max(h_{\text{Left}}, h_{\text{Right}}) + 1$$

și după cum însăși relația sugerează, postordinea pare a fi maniera de traversare necesară (e necesară o traversare pentru că pentru a calcula înălțimea arborelui, trebuie calculată înălțimea fiecărui nod din arbore, și acest lucru se poate realiza doar prin traversare).

```

%height/2
%height(arbore, inaltime)

height(nil,0).
height(t(Left,_,Right),H):-

```

```

height(Left,HLeft),
height(Right,HRight),
HLeft>HRight,!,
H is HLeft+1.
height(t(Left,_,Right),H):-
height(Left,HLeft),
height(Right,HRight),
H is HRight+1.

```

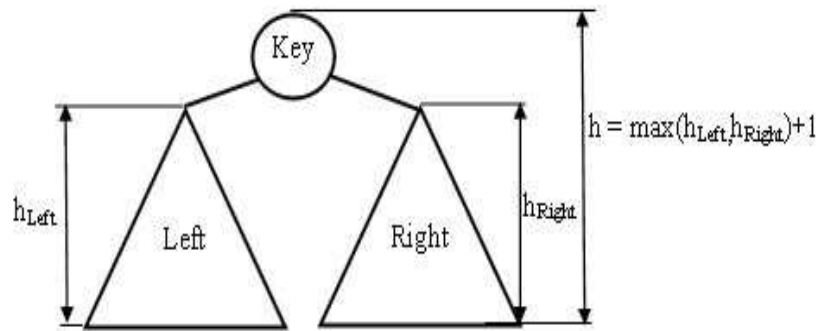


Fig. 6.17

O analiză de eficiență surprinde un aspect neplăcut: deși problema are limită teoretică $\Omega(n)$, și algoritmul surgerează aceeași limită superioară ($O(n)$), datorită specificității limbajelor logice, traversarea postordine introduce o constantă multiplicativă supraunitară. Astfel încât, în loc de o relație de forma $t(n) = 2t(n/2) + 1$ avem o relație de forma $t(n) = 4t(n/2) + 1$ pentru calculul timpului de execuție la implemetarea Prolog. Modificarea constantei conduce la schimbarea clasei algoritmului de la $O(n)$ la $O(n^2)$. Concret, datorită faptului că, după un eventual eșec în clauza a 2a al comparației $HLeft > HRight$, deși $HLeft$ și $HRight$ s-au calculat, prin eșecul produs ele se pierd, și datorită backtrackingului implicit execuția se reia pe clauza a 3a, recalculând valori anterior calculate dar pierdute. Practic în cazul cel mai defavorabil (când în oricare din subarborii arborelui de intrare ramura dreaptă este mai înaltă decât cea stângă) postordinea este afectată ca eficiență. Menționăm că este vorba doar de postordine, unde decizia se ia ulterior apelurilor recursive, ceea ce înseamnă că o decizie greșită necesită reluarea apelurilor.

Problema în cauză are o particularitate favorabilă implementării: putem observa că deși se recalculează în clauza 3, $HLeft$ nu mai este necesară. Aceasta înseamnă că apelul respectiv poate fi omis, reducând relația din teorema master la $t(n) = 3t(n/2) + 1$:

```

height(nil,0).
height(t(Left,_,Right),H):-
    height(Left,HLeft),
    height(Right,HRight),
    HLeft>HRight,!,
    H is HLeft+1.
height(t(Left,_,Right),H):-
    height(Right,HRight),
    H is HRight+1.

```

În felul acesta, clasa algoritmului s-a apropiat de clasa algoritmilor liniari ($O(n) < O(n^{\log_2 3}) < O(n^2)$). E bine, dar nu suficient, din două motive: în primul rând că factorul de multiplicare din apelurile recursive nu s-a redus (încă) la 2, așa cum ne-am aștepta, și, chiar mai grav, soluția găsită este una particulară, pentru problema în cauză. Doar pentru faptul că rezultatul unui apel recursiv nu este necesar ne putem permite să renunțăm la el. Dar nu ne putem baza pe așa ceva în general, pentru o problemă oarecare care necesită postordine. Să găsim soluția generală.

```

height(nil,0).
height(t(Left,_,Right),H):-
    height(Left,HLeft),
    height(Right,HRight),
    max(HLeft,HRight,Max),
    H is Max+1.

%max/3
%max(valoare_intrare,
    valoare_intrare,
    maximul_dintre_cele_doua)

max(A,B,A):-A>=B,!.
max(A,B,B).

```

Soluția în acest caz este să calculăm maximul valorilor înălțimilor celor doi subarbori separat, într-un predicat care primește ca și argumente cele două mărimi, returnând maximul dintre ele. În acest fel discriminarea (în două clauze distincte) revine predicatului de maxim, care ia decizia în funcție de valorile primite, nemaifiind necesară recalcularea lor. În acest fel, formula de calcul a timpului de execuție este: $t(n)=2t(n/2)+1$, adică exact ce s-a dorit. Vestea și mai bună este că această tehnică este aplicabilă în general, indiferent de particularitățile problemei. Codul generic va fi de forma:

```
% compute_post/2
% compute_post(arbore, valoare_calculata)

compute_post(nil, TreeValue) .
compute_post (t(Left, Key, Right)) :-
    compute_post(Left, LeftValue),
    compute_post(Right, RightValue),
    do(Key, TreeValue, LeftValue, RightValue) .
```

Avantajul constă în aceea că se evită duplicarea calculelor prin transmiterea parametrilor predicatului `do/4`.

6.1.5 Exerciții și probleme propuse

1. Scrieți secvența Prolog pentru implementarea ștergerii unui nod din BST conform tehnicii de "agățare" a subarborelui dreapta ca frunză la cel din stânga (varianta 1, prima versiune).

2. Scrieți secvența Prolog pentru implementarea ștergerii unui nod din BST conform tehnicii de "agățare" a subarborelui stânga ca frunză la cel din dreapta (varianta 1, a doua versiune).

3. Scrieți secvența Prolog pentru implementarea ștergerii unui nod din BST prin înlocuirea cu cheia minimă a subarborelui drept. Apoi rulați secvența de ștergeri consecutive prezentată anterior, anticipând forma arborelui de ieșire la fiecare pas.

4. Pentru arborele de intrare:

```
T1=t(t(t(nil,2,nil),3,t(nil,4,nil)),6,t(nil,11,nil))
```

și secvența:

```
?-delete_bst(6,T1,T2),insert_bst(6,T2,T3)
```

care este structura arborilor `T2` și `T3`?

5. Este posibil ca în urma unei secvențe de forma

```
?-delete_bst(Key,T1,T2),insert_bst(Key,T2,T3)
```

arborii `T1` și `T2` să fie identici? În ce condiții? Justificați.

6. Pentru arborele:

```
T=t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
7,t(nil,11,nil))
```

Să se execute predicatul de traversare în pre, în și postordine. Precizați în avans în ce ordine vor fi tipărite cheile.

7. Calculați înălțimea arborelui

$T = t(t(t(nil, 2, nil), 3, t(t(nil, 4, nil), 5, nil)), 6, t(nil, 11, nil))$

8. Să se calculeze adâncimea fiecărui nod dintr-un arbore binar.

Remarcă: adâncimea este diferită de înălțime. Într-un arbore frunzele au înălțimea 0, dar rădăcina are adâncime 0!

9. Să se scrie o secvență Prolog liniară care calculează diametrul unui arbore binar. Diametrul se definește ca cea mai lungă dintre cele mai scurte distanțe între oricare două noduri din arbore.

Sugestie: diametrul unui arbore este un drum între două frunze ale arborelui care nu trebuie în mod necesar să treacă prin rădăcină!

- 10. Ce tip de parcurgere este necesară pentru calcularea numărului de frunze dintr-un arbore binar?**
- 11. Calculați numărul de noduri cu un singur succesor dintr-un arbore binar. Ce strategie abordați?**
- 12. Care este eficiența pentru calcularea numărului de noduri dintr-un arbore binar?**
- 13. Scrieți predicatul Prolog care tipărește toate cheile aflate în noduri de pe nivele pare (rădăcina se consideră pe nivel 0). Care este eficiența?**
- 14. Să se genereze lista tuturor nodurilor având doi succesori dintr-un arbore binar.**
- 15. Se dau doi arbori binari de căutare, de înălțimi h_1 și h_2 . Să se scrie secvența Prolog care generează arborele binar de căutare care conține toate cheile din cei doi arbori, și având înălțimea $h \leq h_1 + h_2$. Calculați eficiența secvenței.**

Remarcă: problema nu necesită transformarea structurilor de intrare sau a vreunei structuri intermediare sau a celei finale în arbori AVL sau arbori perfect echilibrați.

6.2 Structuri incomplete (terminate în variabile)

Dacă până acum am reușit să demonstrăm expresivitatea limbajelor logice, demonstrație susținută de numeroasele exemple, este deja limpede de ce un limbaj ca Prologul este un candidat excepțional pentru specificarea problemelor. Eleganța și simplitatea exprimării (prin trecerea naturală de la gândirea logică și limbajul natural la limbajul predicatelor de ordinul întâi) fac oportună utilizarea limbajului pentru implementarea imediată și testarea unor noi metode sau algoritmi. Mai mult, am arătat că multe probleme cunoscute beneficiază de implementări logice eficiente sau optime (acolo unde problema beneficiază de un algoritm optimal cunoscut) dacă ne referim la timpul de execuție. Din păcate dacă ne referim la dimensiunea spațială, lucrurile sunt diferite: am observat că, spre deosebire de limbajele imperative, în programarea logică e necesar a se folosi un argument de intrare, și un alt argument pentru ieșire, oricât de minore ar fi diferențele între cele două argumente (chiar dacă structura de date este una voluminoasă, iar implementarea în cauză produce modificări minore argumentului de intrare, în cazuri particulare lăsându-o chiar nemodificată, cum este exemplul inserării unui nod în BST, pentru situația în care cheia deja există în arbore).

Putem să influențăm acest aspect? Putem reduce spațiul ocupat? Acest lucru ar fi posibil dacă structura de date care reprezintă variabila de intrare în care se produce transformarea nu ar fi o constantă (notă: un argument de intrare odată instanțiat, indiferent de dimensiunea structurii la care instanțierea se produce devine o constantă, a cărei valoare nu poate fi schimbată decât prin provocarea intenționată a eșecului, caz în care întreaga structură se pierde). Putem reprezenta structura altfel decât ca pe o constantă, punând ca ultim element al structurii o variabilă liberă. În continuare vom exemplifica acest concept pentru liste și arbori.

6.2.1 Liste incomplete (terminate în variabilă)

Dacă listele în mod natural se termină în lista vidă, adică lista $L=[1, 2, 3]$ având 3 elemente, după aplicarea de 3 ori a șablonului `|de` computere/descompunere, se ajunge la structura `[]`, structura nou propusă, înlocuiește terminația vidă `[]` în terminația variabilă indiferentă `_`. În acest fel, lista completă cu 3 elemente $L=[1, 2, 3]$ va avea echivalentul său în listă terminată în variabilă cu 3 elemente: $L=[1, 2, 3|_]$. În timp ce condiția de terminare pe liste complete verifică ajungerea la `[]` (apelul recursiv pe restul listei T a ajuns la `[]`), pentru listele terminate în variabile condiția va testa ajungerea la variabila finală (apelul recursiv pe restul listei T a ajuns la o variabilă, adică `var(T)`).

Deci o listă terminată în variabile are coada listei variabilă liberă, și nu un element final lista vidă. Atragem atenția la diferența între listele:

```
L1=[1, 2, 3|A]
și
L2=[1, 2, 3, B]
```

Lista L_1 este o listă incompletă (terminată în variabilă), conținând 3 elemente, 1, 2, 3 după care urmează variabila finală (semantica lui A fiind de *listă*, deoarece apare după șablonul `|`), în timp ce L_2 este o listă completă (terminată în lista vidă), care are 4 elemente, dintre care 3 sunt precizate (1, 2, și 3) iar al 4lea este o variabilă, (semantica lui B este de *element al listei*, deoarece este separat prin `,` de restul elementelor, și nu prin șablon). Atenție deci: o listă NU este terminată în variabilă doar pentru că apare o variabilă în listă (chiar în cazul în care variabila apare pe poziția ultimului element), ci este terminată în variabilă doar dacă variabila finală are semantica de listă (reprezentând coada listei).

Să analizăm acum comportamentul câtorva predicate elementare (discutate deja pentru listele complete) în cazul listelor terminate în variabile).

Apartenența la o listă: `member/2`

```
%member/2
%member(element, listă)

member(H, [H|_]) .
```

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

Presupunem că avem întrebarea:

```
?- L=[1,2,3|_], member(2,L).
Yes, L=[1,2,3|_]
```

răspunsul este cel așteptat. Neașteptat este însă răspunsul la întrebarea:

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

Deși 4 nu este un element al listei L, predicatul confirmă existența lui. Mai mult, chiar îl adaugă în listă, în fapt răspunsul la întrebare fiind:

```
Yes, L=[1,2,3,4|_]
```

Răspunsul este surprinzător (pentru moment), și aparent, INCORECT! (în fapt vom vedea imediat că e corect; incorectitudinea ni se datorează exclusiv, și are o explicație simplă. Am folosit un predicat `member/2` pentru altă structură de date decât pentru care a fost proiectat).

Hai-deți să facem analiza execuției ultimei întrebări: în mod evident, la primul pas, matchingul întrebării peste clauza 1 eșuează (prin eșecul unificării lui 4 cu 1 prin intermediul lui H), iar matching-ul (reușit) cu clauza 2 produce apelul recursiv pe coada listei, adică: `member(4, [2,3|_])`. Prin același raționament, încă 2 apeluri eșuează în tentativa de unificare cu clauza 1 (prin eșecul unificării lui 4 cu 2 respectiv 4 cu 2 prin intermediul lui H), fiecare unificându-se cu clauza 2 și producând apelurile recursive `member(4, [3|_])` și respectiv `member(4, _)` (în ultimul apel variabila argumentului 2 având semantica de listă). Să privim în detaliu acest ultim apel și să- analizăm.

`member(4, _)` se suprapune peste `member(H, [H|_])`

notăm

```
member(4, _) # member(H, [H|_])
```

în urma unificării, se formează sistemul de ecuații (corespunzător unificării perechilor de parametri):

$$\begin{cases} H = 4 \\ _ = [H|_] \end{cases} \Rightarrow _ = [4|_]$$

Acum, dacă revenim și observăm că variabila liberă $_$ care s-a unificat în sistemul de mai sus este chiar variabila liberă din terminația $L=[1,2,3|_]$, devine evident, că înlocuind $_$ din $L=[1,2,3|_]$ cu $[4|_]$ cu care aceasta s-a instanțiat obținem:

$$L=[1,2,3|[4|_]]= [1,2,3,4|_].$$

Deci semantica predicatului `member/2` scris pentru liste complete, în cazul utilizării pentru liste terminate în variabilă este diferită, și anume: verifică apartenența unui element la listă; în cazul în care există, returnează adevărat, lăsând lista nemodificată, în timp ce în cazul în care elementul lipsește, îl adaugă ca ultim element al listei. Pentru a păstra semantica avută la listele complete, predicatul trebuie rescris sub forma:

```
%member_LTV/2
%member_LTV(element, listă)

member_LTV(H,L):-
    var(L),!,fail.
member_LTV(H,[H|_]).
member_LTV(H,[_|T]):-
    member_LTV(H,T).
```

Interpretarea clauzelor 2 și 3 este identică cu interpretarea avută pentru listele terminate în listă vidă: identificarea elementului căutat (și oprirea cu succes), respectiv avansul recursiv. Se observă că însele clauzele au forma identică cu varianta anterioară. Singura modificare este adăugarea unei noi clauze, pe prima poziție (și de data aceasta poziționarea ca primă clauză este importantă și obligatorie), condiția de terminare cu eșec.

Analiza clauzei: dacă în urma apelurilor recursive în lista terminată în variabilă s-a ajuns la variabila finală (`var(L)` returnează `true`), atunci înseamnă că elementul nu a fost identificat în listă (în caz contrar s-ar fi oprit pe clauza 2), și deci acest fapt trebuie menționat de predicat, prin eșec. Combinația `var(L),!,fail` asigură testarea ajungerii la variabila finală, și provocarea ireversibilă (datorită `!`) a eșecului, fără posibilitatea de revenire,

ceea ce este în conformitate cu specificațiile; dacă elementul nu este prezent în LTV, atunci predicatul trebuie să eșueze.

Prin analiza comparativă a predicatelor `%member_LTV/2` și `%member/2` se naște o întrebare absolut firească: de ce, pentru cazul de terminare cu insucces LTV au nevoie de clauză explicită, în timp ce listele normale nu. Cu alte cuvinte, se pune întrebarea de ce nu este în vigoare unul din următoarele cazuri:

1. Ambele tipuri de liste au condiție explicită de terminare cu insucces, deci clauzele:

```
member_LTV(H,L):-var(L),!,fail.
```

și

```
member(H,[]):-!,fail.
```

sunt ambele prezente în predicatele corespunzătoare.

2. Niciuna din cele două tipuri de liste nu au condiție explicită de terminare cu insucces: deci niciuna din clauzele mai sus menționate nu există în predicatele corespunzătoare.

Este evident că varianta a doua este din start greșită: lipsa clauzei `member_LTV(H,L):-var(L),!,fail.` va produce inserarea elementului la sfârșitul LTV, în cazul în care el nu există la apel (această afirmație a fost dovedită anterior).

Cum varianta a doua nu e posibilă, sugerăm adăugarea clauzei la `member(H,[]):-!,fail.`, și deci, retranscrierea lui `member/2` sub forma:

```
%member/2
%member(element, listă)
```

```
member(H,[]):-!,fail.
member(H,[H|_]).
member(H,[_|T]):-
    member(H,T).
```

Propunerea pare corectă, și după analiză, se observă că adăugarea nu afectează comportamentul. Atunci cum apărea eșecul în cazul căutării unui

element într-o listă completă în care elementul nu există, în lipsa primei clauze din forma actuală, adică, la întrebarea:

```
?- member(4, [1,2,3]).  
no.
```

Cum se produce eșecul? După avansul pe clauza recursivă de 3 ori, apelul ajunge la `?- member(4, [])`, care neavând nici o clauză peste care să se suprapună (`[]` nu poate face matching cu niciuna din variantele `[H|_]`, `[_|T]` din clauzele existente, care impun existența a cel puțin unui element) eșuează. Deci, pentru `%member/2` simpla lipsă a clauzei `member(H, []) :- !, fail.` are efectul prezenței sale. Acest lucru este valabil pentru orice structuri de date terminate în structura vidă (structuri complete): simpla lipsă a unei clauze terminate cu eșec pentru structura vidă are același efect ca și prezența sa (Atenție: observația nu mai rămâne valabilă dacă înainte de tăierea de backtracking se produc efecte laterale acest mecanism va fi detaliat la prezentarea logicii implicite). În cazul lipsei unei astfel de clauze, se produce eșecul implicit, structura vidă nesuprapunându-se peste structurile nevide. Mai mult, lipsa unei astfel de clauze este în slujba eficienței: la execuție sunt mai puține clauze de testat. (În fapt limbajele actuale realizează indexarea după primul argument, ceea ce înseamnă că se realizează o partiționare a clauzelor în funcție de cazurile argumentului. În cazul de față, cazurile ar fi `[]`, respectiv `[H|T]` care ar fi tratate eficient în cazul în care acesta ar fi primul argument).

Remarca NU este valabilă pentru structurile terminate în variabilă, unde condiția de eșec trebuie să fie explicit prezentă. Mai mult această condiție trebuie să fie prima testată.

De reținut:

1. Pentru listele complete (terminate în lista vidă) ordinea clauzelor nu este esențială (obligatorie o anumită ordine pentru corecta funcționare), putându-se realiza o ordonare sau alta în funcție de interes (ordinea de obținere a soluțiilor în cazul nedeterminist, eficiența, etc.).
2. Pentru listele complete absența condiției de terminare cu eșec pe lista vidă este suplinită de eșecul care se produce implicit.

3. Pentru listele terminate în variabilă (LTV) ordinea clauzelor este esențială, o proastă ordonare putând conduce la intrarea în buclă infinită.
4. Pentru listele terminate în variabilă (LTV) condiția de terminare cu eșec este obligatoriu să existe, și ea TREBUIE să fie prima clauză. Poziționarea ei oriunde altundeva este inefectivă.

Acum vom rescrie inserarea într-o LTV: scriem predicatul care să verifice dacă un element este membru într-o listă; dacă da, răspunde afirmativ, dacă nu, îl inserează fără o nouă parcurgere și fără a genera o nouă listă (inserarea în aceeași listă). Varianta pentru liste complete ar necesita în mod obligatoriu trei argumente.

```
%inser/3
%inser(element, lista_sursa, lista_destinatie)

inser(H, [], [H]) .
inser(H, [H|T], [H|T]) :- ! .
inser(E, [H|T], [H,R]) :-
    inser(E, T, R) .
```

LTV ne permit să renunțăm la al treilea argument, folosind lista ca argument de intrare/ieșire.

```
%inser_LTV/2
%inser_LTV(element, lista)

inser_LTV(H, L) :-
    var(L) , ! ,
    L=[H|_] .
inser_LTV(H, [H|_]) :- ! .
inser_LTV(H, [_|T]) :-
    inser_LTV(H, T) .
```

Clauza 1 se referă la situația în care, parcurgând întreaga listă s-a ajuns la variabila finală fără să fi găsit elementul în listă. În acest caz, variabila finală `L` "crește" la `L=[H|_]`, argumentul rămânând o LTV. Clauzele 2 și 3 sunt identice cu clauzele corespondente pentru listele complete, cu excepția argumentului 3 care nu mai este necesar.

Analizând soluția observăm că în prima clauză, variabila din coada listei `L` se unifică cu `[H|_]`, ceea ce face ca rezultatul să devină identic cu clauza 2.

Deci practic cele două clauze sunt identice (de fapt prima devine identică cu a doua DUPĂ unificarea $L=[H|_]$), ceea ce face posibilă renunțarea la prima clauză, și retranscrierea predicatului sub forma:

```
inser_LTV(H, [H|_]) :- !.
inser_LTV(H, [_|T]) :-
    inser_LTV(H, T).
```

În acest caz clauza 1 are două semantici distincte:

- elementul care se dorește a fi inserat este identificat ca fiind primul element din LTV (capul LTV, fie chiar primul element al său, fie un element interior, la care s-a ajuns după apelurile recursive aplicate prin intermediul clauzei a doua),
- elementul de inserat nu există în listă și se adaugă acum. Ajungându-se la variabila finală, aceasta se unifică cu șablonul $[H|_]$ producând astfel adăugarea elementului la sfârșitul listei, și păstrarea listei ca LTV, datorită variabilei $_$ din structura $[H|_]$.

La `inser_LTV /2` se observă nu doar reducerea spațiului utilizat (prin reducerea numărului de argumente, doar două, față de trei argumente la `inser/3`), dar și o îmbunătățire a performanței timp. Reducerea numărului de clauze înseamnă mai puține încercări, și deci obținerea soluției mai repede.

Pentru a evidenția diferența între LTV și lista care conține o variabilă finală, furnizăm răspunsul la următoarele două întrebări aparent similare:

```
| ?- L=[1,3|_], inser_LTV (5,L).
L = [1,3,5|_A] ?
yes
```

```
| ?- L=[1,5,_], inser_LTV (7,L).
L = [1,5,7] ?
```

Se remarcă faptul că în cazul listei $L=[1,3|_]$, la inserare ea rămâne o LTV. Pentru $L=[1,5,_]$, deși predicatul apelat este cel corespunzător LTV, variabila din coadă se instanțiază, și lista rezultată NU este o LTV

6.2.2 Arbori incompleți (terminați în variabilă)

Exact ca pentru liste, varianta LTV adăugând variabilă în locul listei vide, pentru arborii binari, terminația `nil` este înlocuită cu variabila liberă. Căutarea într-un binar de căutare terminat în variabile va fi:

```
%search_ATV/2
%search_ATV(cheie_de_cautat,arbore_sursa).

search_ATV(Key,T):-
    var(T),!,
    write('cheia cautata nu se afla in arbore'),
    fail.
search_ATV(Key,t(Left,Key,Right)):-!,
    write('am ajuns la cheia cautata').
search_ATV(Key,t(Left,Key1,Right)):-
    Key<Key1,!,search_ATV(Key,Left).
search_ATV(Key,t(Left,Key1,Right)):-
    search_ATV(Key,Right).
```

Singura modificare (în afară de schimbare numelui) fiind în fapt eșecul explicit în cazul ajungerii la variabila finală.

Execuția căutării produce rezultatele scontate, după cum se poate urmări mai jos, pe câteva exemple:

```
gen(T):-    inser_ATV(7,T),inser_ATV(3,T),
            inser_ATV(5,T),inser_ATV(9,T),
            inser_ATV(8,T),inser_ATV(2,T).
gen1(T):-   inser_ATV(7,T),inser_ATV(3,T).

run_search(Key,T):-gen(T),search_ATV(Key,T).
run_search1(Key,T):-gen1(T),search_ATV(Key,T).

| ?- run_search1(3,T).
am ajuns la cheia cautata
T = t(t(_A,3,_B),7,_C) ? ;
no
| ?- run_search1(4,T).
cheia cautata nu se afla in arbore
no
| ?- run_search(5,T).
am ajuns la cheia cautata
```

```

T = t(t(t(_A,2,_B),3,t(_C,5,_D)),7,t(t(_E,8,_F),9,_G))
? ;
no
| ?- run_search(4,T).
cheia cautata nu se afla in arbore
no

```

Inserarea într-un arbore terminat în variabile (ATV) devine:

```

%inser_ATV/2
%inser_ATV(cheie_de_cautat_sau_inserat, arbore_TV).

inser_ATV(Key,T):-
    var(T),!,
    T=t(_ ,Key,_ ).
inser_ATV(Key,t(_ ,Key_)):-!.
inser_ATV(Key,t(Left,Key1,_)):-
    Key<Key1,!,
    inser_ATV(Key,Left).
inser_ATV(Key,t(_ ,Key1,Right)):-
    inser_ATV(Key,Right).

```

Ca și în cazul listelor, semantica predicatului de inserare este: predicatul verifică dacă un element este prezent în arbore; dacă da, răspunde afirmativ, dacă nu, îl inserează fără o nouă parcurgere și fără a genera un nou arbore (inserarea în același arbore). La fel ca și în cazul listelor, observăm clauza 1 soluționând situația în care, prin recursivitate, ajungând la variabila finală în arborele de căutare terminat în variabilă, acesta este înlocuită cu un nod, conținând două variabile libere pe poziția succesorilor ($T=t(_,Key,_)$), în timp ce clauza 2 răspunde situației în care elementul de căutat/inserat este identificat ca fiind deja prezent în arbore. Tot ca și în cazul listelor, cele două pot fi unificate într-o singură clauză (a doua), predicatul devenind:

```

inser_ATV(Key,t(_ ,Key,_)):-!.
inser_ATV(Key,t(Left,Key1,_)):-
    Key<Key1,!,
    inser_ATV(Key,Left).
inser_ATV(Key,t(_ ,Key1,Right)):-
    inser_ATV(Key,Right).

```

La o întrebare de forma:

```

?-inser(7,T),inser(3,T),inser(5,T).

```

unde T este variabilă liberă la apel, arborele va avea formele intermediare:

```
1.t(_,7,_).
2.t(t(_,3,_),7,_).
3.t(t(_,3,t(_,5,_)),7,_).
```

Dacă urmărim o execuție pe codul de mai sus:

```
run_insert(T):- inser_ATV(7,T),
                inser_ATV(3,T),
                inser_ATV(5,T).
| ?- run_insert(T).
T = t(t(_A,3,t(_B,5,_C)),7,_D) ? ;
no
```

În timp ce o execuție cu tipărirea variabilelor de tip arbore după fiecare inserare în parte:

```
run_insert_wr(T):- inser_ATV(7,T),
                   write('T='),write(T),nl,
                   inser_ATV(3,T),
                   write('T='),write(T),nl,
                   inser_ATV(5,T),
                   write('T='),write(T),nl.
| ?- run_insert_wr(T).
T=t(_671,7,_673)
T=t(t(_836,3,_838),7,_673)
T=t(t(t(_836,3,t(_1028,5,_1030)),7,_673)
T = t(t(_A,3,t(_B,5,_C)),7,_D) ? ;
no
```

Ștergerea dintr-un ATV de căutare:

```
%delete_ATV/3
delete_ATV(cheie_de_sters,arbore_sursa,arbore_rezultat)

delete_ATV(Key,Tree,Tree):-
    var(Tree),!,
    write(Key),
    write(' nu exista in
arbore').delete_ATV(Key,t(Left,Key,Right),Right):-
    var(Left),!.
delete_ATV(Key,t(Left,Key,Right),Left):-
```



```

    var(Right),!.
delete_ATV(Key,t(Left,Key,Right),t(NewLeft,NewKey,Right)):-!,
    deleteMaxFromITree(Left,NewKey,NewLeft).
delete_ATV(Key,t(Left,Key1,Right),t(NewLeft,Key1,Right)):-
    Key<Key1,!,
    delete_ATV(Key,Left,NewLeft).
delete_ATV(Key,t(Left,Key1,Right),t(Left,Key1,NewRight)):-
    delete_ATV(Key,Right,NewRight).

%deleteMaxFromTree/3
%deleteMaxFromTree(arbore_sursa,
    cheie_maxima_din_arbore,
    arbore_rezultat)

deleteMaxFromITree(t(Left,MaxKey,Right),MaxKey,Left):-
    var(Right),!.
deleteMaxFromITree(t(Left,Key,Right),MaxKey,t(Left,Key,NewRight)):-
    deleteMaxFromITree(Right,MaxKey,NewRight).

```

Ambele predicate păstrează structura din cazul BST, incluzând arborii de ieșire, deoarece structura acestora se schimbă. Specifice sunt condițiile de terminare, care testează ajungerea la variabila finală nu doar pentru structura în ansamblu (`var(Tree)`) ci și pentru fiecare subarbore în parte (`var(Left)` respectiv `var(Right)`), condiție specifică ATV.

Iar o rulare pe aceiași arbori ca la inserare și căutare:

```

run_del(Key,T):-gen(Input),delete_ATV(Key,Input,T).
run_dell(Key,T):-gen1(Input),delete_ATV(Key,Input,T).
gen(T):-    inser_ATV(7,T),inser_ATV(3,T),
            inser_ATV(5,T),inser_ATV(9,T),
            inser_ATV(8,T),inser_ATV(2,T).
gen1(T):-    inser_ATV(7,T),inser_ATV(3,T).

| ?- run_dell(4,T).
4 nu exista in arbore
T = t(t(_A,3,_B),7,_C) ? ;
no
| ?- run_dell(3,T).
T = t(_A,7,_B) ? ;
no
| ?- run_del(3,T).
T = t(t(_A,2,t(_B,5,_C)),7,t(t(_D,8,_E),9,_F)) ? ;
no
| ?- run_del(4,T).

```

```

4 nu exista in arbore
T = t(t(t(_A,2,_B),3,t(_C,5,_D)),7,t(t(_E,8,_F),9,_G))
? ;
no

```

Observațiile generale de la LTV se mențin și în cazul ATV:

- ordinea clauzelor este esențială, o proastă ordonare putând conduce la intrarea în buclă infinită.
- condiția de terminare cu eșec este obligatoriu să existe (la căutare), și ea trebuie să fie prima clauză. Poziționarea ei oriunde altundeva este inefectivă.

Urmărind câteva operații succesive de inserare și ștergere în ATV, cu tipărirea arborilor rezultați, vom folosi următoarele predicate auxiliare pentru tipărire:

```

write_ATV(Tree):-
    write_ATV(0,Tree).

%write_ATV /2
%write_ATV (nivel,arbore)

write_ATV(_,Tree):-var(Tree),!.
write_ATV(Level,t(Key,Left,Right)):-
    LevelPlus1 is Level + 1,
    write_ATV(LevelPlus1,Right),
    nl,writeSpaces(Level),
    write('['),write(Key),write(']'),
    write_ATV(LevelPlus1,Left).

```

Tipărește 4N caractere spațiu.

```

%writeSpaces/1
%writeSpaces(integer)

writeSpaces(0):-!.
writeSpaces(N):-
    write(' '),
    NMinus1 is N - 1,
    writeSpaces(NMinus1).

| ?- nl,write(`Insert 7
:'),insert_ATV(7,T),write_ATV(T),
nl,write(`Insert 11 :'),insert_ATV(11,T),write_ATV(T),
nl,write(`Insert 3 :'),insert_ATV(3,T),write_ATV(T),

```

```

nl,write('Insert 5 :'),insert_ATV(5,T),write_ATV(T),
nl,write('Insert 2 :'),insert_ATV(2,T),write_ATV(T),
nl,write('Insert 4 :'),insert_ATV(4,T),write_ATV(T),
nl,write('Insert 3 :'),insert_ATV(3,T),write_ATV(T),nl,
write('Insert 6 :'), insert_ATV (6,T),write_ATV(T),nl,
write('Delete 7 :'),delete_ATV(7,T,T1),write_ATV(T1), nl,
write('Delete 13 :'),delete_ATV(13,T1,T2),write_ATV(T2),nl,
write('Delete 5 :'),delete_ATV(5,T2,T3),write_ATV(T3),
nl,write('Delete 11 :'),delete_ATV(11,T3,T4),write_ATV(T4).

```

```

Insert 7 :
[7]
Insert 11 :
[11]
[7]
Insert 3 :
[11]
[7]
[3]
Insert 5 :
[11]
[7]
[5]
[3]
Insert 2 :
[11]
[7]
[5]
[3]
[2]
Insert 4 :
[11]
[7]
[5]
[4]
[3]
[2]
Insert 3 :
[11]
[7]
[5]
[4]
[3]
[2]
Insert 6 :

```

```

[11]
[7]
[6]
[5]
[4]
[3]
[2]
Delete 7 :
[11]
[6]
[5]
[4]
[3]
[2]
Delete 13 :13 nu exista in arbore
[11]
[6]
[5]
[4]
[3]
[2]
Delete 5 :
[11]
[6]
[4]
[3]
[2]
Delete 11 :
[6]
[4]
[3]
[2]
T=t(t(t(_A,2,_B),3,t(t(_C,4,_D),5,t(_E,6,_F))),7,t(_G,11,_H)),
T1 = t(t(t(_A,2,_B),3,t(t(_C,4,_D),5,_E)),6,t(_G,11,_H)),
T2 = t(t(t(_A,2,_B),3,t(t(_C,4,_D),5,_E)),6,t(_G,11,_H)),
T3 = t(t(t(_A,2,_B),3,t(_C,4,_D)),6,t(_G,11,_H)),
T4 = t(t(t(_A,2,_B),3,t(_C,4,_D)),6,_H) ?
yes

```

6.2.3 Exerciții și probleme propuse

1. Este corectă soluția pentru `%member_LTV/2` având următoarea ordine a clauzelor? Justificați.

```
member_LTV(H, [_|T]) :-
    member_LTV(H, T) .
member_LTV(H, [H|_]) .
member_LTV(H, L) :- var(L) , ! , fail .
```

Dar dacă reordonăm:

```
member_LTV(H, [H|_]) .
member_LTV(H, [_|T]) :-
    member_LTV(H, T) .
member_LTV(H, L) :- var(L) , ! , fail .
```

Dar:

```
member_LTV(H, L) :- var(L) , ! , fail .
member_LTV(H, [_|T]) :-
    member_LTV(H, T) .
member_LTV(H, [H|_]) .
```

Găsiți răspunsul la întrebarea:

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

La fiecare din cele trei variante menționate.

2. Ce se întâmplă dacă rescriem %member_LTV/2 sub forma:

```
member_LTV(H, [_|T]) :-
    member_LTV(H, T) .
member_LTV(H, [H|_]) .
member_LTV(H, L) :-
    var(L) , ! , fail .
```

și avem întrebarea:

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

3. Ce se întâmplă dacă rescriem %member_LTV/2 sub forma:

```
member_LTV(H, L) :- var(L) , ! , fail .
member_LTV(H, [_|T]) :-
    member_LTV(H, T) .
member_LTV(H, [H|_]) .
```

și avem întrebarea:

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

Justificați.

4. Justificați răspunsul primit la repetarea întrebării:

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

în cazul în care se rescrie predicatul sub forma:

```
inser_LTV(H,L):-
    var(L),
    L=[H|_].
inser_LTV(H,[H|_]):-!.
inser_LTV(H,[_|T]):-
    inser_LTV(H,T).
```

5. Cum se rescrie inser_ATV în cazul arborilor oarecare (nu de căutare) terminați în variabile.

6. Pentru arborele de intrare:

```
T1=t(t(t(_,2,_),3,t(_,4,_)),6,t(_,11,_))
```

și secvența:

```
?-delete_AVT(6,T1,T2),insert_AVT(6,T2)
```

care este structura arborelui T2 după fiecare dintre cele două operații?

7. Scrieți secvența Prolog pentru implementarea ștergerii unui nod din AVT prin înlocuirea cu cheia minimă a subarborelui drept. Apoi rulați secvența de ștergeri consecutive prezentată anterior, anticipând forma arborelui de ieșire la fiecare pas.

8. Să se scrie predicatele pentru parcurgerile în pre, în in și postordine în ATV.

9. Rescrieți predicatul height/2 de calculare a înălțimii unui arbore, pentru cazul ATV.

10. Să se genereze lista tuturor nodurilor având cu un singur succesor dintr-un ATV. Ce strategie abordați?

- 11.** Calculați numărul de noduri cu doi succesori dintr-un ATV.
- 12.** Ce tip de parcurgere este necesară pentru calcularea numărului de frunze dintr-un ATV?

7. Liste diferență

7.1 Transformarea arborilor de căutare în liste

Numeroase probleme necesită transformări în structuri de date; între acestea este și cea a trecerii de la forma de arbore la cea de listă, în particular de la BST la listă ordonată. Parcurgerea în inordine asigură traversarea (și deci obținerea elementelor) în ordine crescătoare. Pe de altă parte construirea listei rezultat este posibil să se realizeze doar în momentul în care subarborii au fost deja transformați. Rezultă că elementele din structura de ieșire vor corespunde parcurgerii în inordine, în timp ce procesul propriu-zis de traversare are loc în postordine. Să ne reamintim predicatul generic de traversare în postordine:

```

traverse_post(nil) .
traverse_post(t(Left,Key,Right)) :-
    traverse_post(Left),
    traverse_post(Right),
    do(Key) .

```

În cazul nostru, elementele din arbore vor trebui transferate într-o listă ordonată, deci predicatul de traversare va trebui să mai aibă un argument de ieșire, iar predicatul `do/1` din secvența generică va trebui să combine rezultatele parțiale (obținute prin apelurile recursive) într-unul singur.

```

%generate_ordered_list/2
%generate_ordered_list(arbore de intrare, lisa noduri
inordine)

generate_ordered_list(nil, []).
generate_ordered_list(t(Left,Key,Right),List) :-
    generate_ordered_list(Left,ListLeft),
    generate_ordered_list(Right,ListRight),
    do(Key,ListLeft,ListRight,List) .

```

Este evident că specificațiile predicatului `do/4` menționat mai sus sunt perfect îndeplinite de `append3/4`.

```

do(Key,ListLeft,ListRight,List) :-
    append3(ListLeft,[Key],ListRight,List) .

```


Având în vedere că lista a doua conține exact un element de fiecare dată putem evita apelul inutil pe acea listă, aplicând șablonul, și deci:

```
do (Key, ListLeft, ListRight, List) :-
    append(ListLeft, [Key|ListRight], List) .
```

Cu acestea, predicatul devine:

```
generate_ordered_list(nil, []).
generate_ordered_list (t(Left, Key, Right), List) :-
    generate_ordered_list(Left, ListLeft),
    generate_ordered_list(Right, ListRight),
    append(ListLeft, [Key|ListRight], List) .
```

O analiză de eficiență ne arată că după cele două apelurilor recursive (de mărime $t(n/2)$ pentru cazul mediu) secvența liniară (din afara apelurilor, introdusă de `append`) induce un ordin de mărime al algoritmului supraliniar ($O(n \lg n)$ în caz mediu).

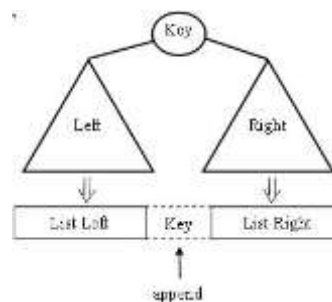
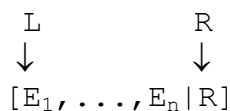


Fig. 7.1

7.1.1 Inordinea

Creșterea eficienței este posibilă dacă se Fig. 7.1 consideră un tip particular de structuri, și anume listele diferență. Noțiunea de listă diferență este ilustrată în continuare, R fiind sublistă a lui L .



Lista diferență D , notată $L \setminus R$, este:

$$D = L \setminus R = [E_1, \dots, E_n]$$

Dacă se dă o listă L ('pointer' la începutul listei L) și o sublistă R ('pointer' la un sfârșitul diferenței), atunci lista diferență $D = L \setminus R$, constă din elementele din

L care nu sunt în R (elementele dintre cei doi 'pointeri' L și R). Listele diferență mai sunt numite și liste cu prim și ultim element (deoarece se precizează începutul = primul element, și respectiv sfârșitul = ultimul element al listei). Putem folosi această tehnică pentru a elimina operația de concatenare, după cum urmează:

```
%generate_ordered_list/3
%generate_ordered_list(arbore, inceput lista diferenta,
sfarsit lista diferenta)

generate_ordered_list(nil,L,L).
generate_ordered_list (t (Left,Key,Right),First,Last):-
    generate_ordered_list(Left,FirstListLeft,LastListLeft),
    generate_ordered_list(Right,FirstListRight,LastListRight),
    First= FirstListLeft,
    LastListLeft=[Key|FirstListRight],
    Last= LastListRight.
```

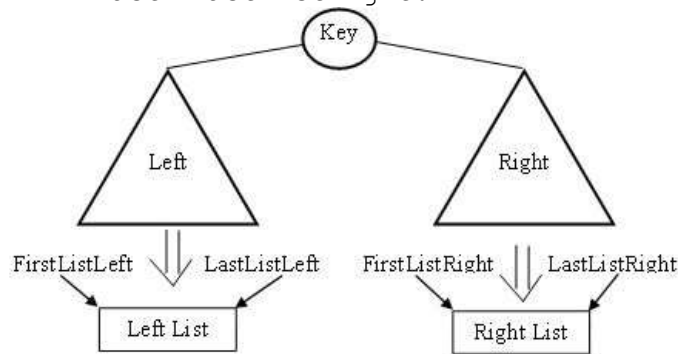


Fig. 7.2

Varianta de mai sus este cea cu unificare explicită, prin faptul că argumentele listei diferență se instanțiază explicit, prin operații separate, specificate. Ce menționează predicatul: după cele două apeluri recursive, care realizează transformarea subarborilor stânga și dreapta în perechi de liste diferență ($\text{FirstListLeft}, \text{LastListLeft}$ și respectiv $\text{FirstListRight}, \text{LastListRight}$) se realizează unificarea explicită a argumentelor, care în fapt rezolvă aceeași problemă pe care `append` o realiza: adaugă `Key` la sfârșitul primei liste (rezultatul apelului pe stânga), și rezultatul apelului pe dreapta la acest rezultat intermediar. În fapt operația se produce puțin diferit. Deoarece a spune că adaugă cheia `Key` la sfârșitul primei liste este același lucru (din punctul de vedere al rezultatului) cu aceea că adaugă cheia `Key` la începutul celei de-a doua liste) în fapt acest lucru se realizează; din punct de vedere al eficienței, această versiune este mai

rapidă. În prima versiune prin `[Key|ListRight]`, iar în cea de-a doua prin `[Key|FirstListRight]`.

Practic interpretarea unificărilor explicite arată:

`First= FirstListLeft` rezultatul întregii structuri (liste) de ieșire începe acolo unde începe rezultatul apelului pe subarborele stânga,

`Last= LastListRight` rezultatul întregii structuri (liste) de ieșire se termină acolo unde se termină rezultatul apelului pe subarborele dreapta,

`LastListLeft=[Key|FirstListRight]` rezultatul listă obținut la apelul recursiv pe stânga va avea imediat după ultimul său element rezultatul listă (începutul listei) obținut la apelul recursiv pe dreapta, în fața căreia se adaugă cheia nodului rădăcină.

Condiția de terminare specifică faptul că transformarea arborelui vid este o listă vidă, obținută prin diferența a două liste identice ($L \setminus L = []$).

Predicatul necesită și un tip particular de apel inițial:

```
?-generate_ordered_list(Tree,List,[]).
```

Transcrierea soluției de mai sus cu unificare implicită conduce la:

```
generate_ordered_list(nil,L,L).
generate_ordered_list(t(Left,Key,Right),
    FirstListLeft,LastListRight):-
    generate_ordered_list(Left,FirstListLeft,[Key|Intermediate]),
    generate_ordered_list(Right,Intermediate,LastListRight).
```

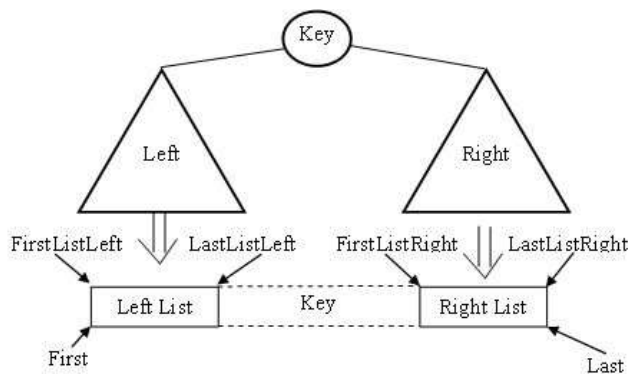


Fig.7.3

Rularea codului anterior pe diferiți arbori de intrare:

```

run1(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
    7,
    t(nil,11,nil)),
    generate_ordered_list(Tree,List,[]).

run2(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(nil,4,nil)),6,t(nil,11,nil)),
    generate_ordered_list(Tree,List,[]).

run3(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,nil)),6,t(nil,11,nil)),
    generate_ordered_list(Tree,List,[]).

run4(List):-
    Tree=
    t(t(t(t(nil,1,nil),2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
    7,
    t(t(nil,8,t(nil,10,nil)),11,nil)),
    generate_ordered_list(Tree,List,[]).

run5(List):-
    Tree=
    t(t(t(t(nil,1,nil),2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
    7,
    t(t(nil,8,t(t(nil,9,nil),10,nil)),11,t(nil,12,nil))),
    generate_ordered_list(Tree,List,[]).

```

Conduce la obținerea listelor ordonate:

```

| ?- run1(List).
List = [2,3,4,5,6,7,11] ? ;
no
| ?- run2(List).
List = [2,3,4,6,11] ? ;
no
| ?- run3(List).
List = [2,3,4,5,6,11] ? ;
no
| ?- run4(List).
List = [1,2,3,4,5,6,7,8,10,11] ? ;

```

```
no
| ?- run5(List).
List = [1,2,3,4,5,6,7,8,9,10|...] ? ;
No
```

7.1.2 Preordinea

Pentru diferite probleme lista în pre respectiv postordine ar putea fi necesară. În continuare prezentăm variantele de obținere a acestor liste cu append, și liste diferență.

```
generate_preorder_list(nil, []).
generate_preorder_list(t(Left,Key,Right),List):-
    generate_preorder_list(Left,ListLeft),
    generate_preorder_list(Right,ListRight),
    append([key|ListLeft],ListRight,List).
```

Cu liste diferență, varianta cu unificare explicită:

```
generate_preorder_list(nil,L,L).
generate_preorder_list(t(Left,Key,Right),First,Last):-
    generate_preorder_list(Left,FirstListLeft,LastListLeft),
    generate_preorder_list(Right,FirstListRight,LastListRight),
    First= [Key|FirstListLeft],
    LastListLeft=FirstListRight,
    Last= LastListRight.
```

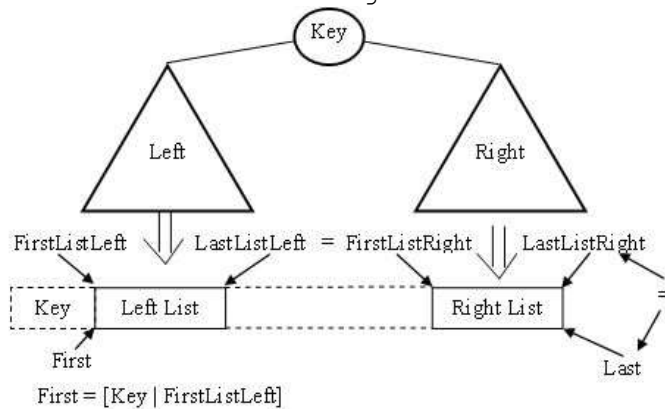


Fig. 7.4

Cu liste diferență, varianta cu unificare implicită:

```

generate_preorder_list(nil,L,L).
generate_preorder_list(t(Left,Key,Right),
                        [Key|FirstListLeft],LastListRight):-
    generate_preorder_list(Left,FirstListLeft,Interm),
    generate_preorder_list(Right,Interm,LastListRight).

```

Și acest predicat necesită un tip particular de apel inițial:

```
?-generate_preorder_list(Tree,List,[]).
```

Iar execuția pe aceiași arbori de intrare

```

runp1(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
    7,
    t(nil,11,nil)),
    generate_preorder_list(Tree,List,[]).

runp2(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(nil,4,nil)),6,t(nil,11,nil)),
    generate_preorder_list(Tree,List,[]).

runp3(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,nil)),6,t(nil,11,nil)),
    generate_preorder_list(Tree,List,[]).

runp4(List):-
    Tree=
    t(t(t(t(nil,1,nil),2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
    7,
    t(t(nil,8,t(nil,10,nil)),11,nil)),
    generate_preorder_list(Tree,List,[]).

runp5(List):-
    Tree=
    t(t(t(t(nil,1,nil),2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
    7,
    t(t(nil,8,t(t(nil,9,nil),10,nil)),11,t(nil,12,nil))),
    generate_preorder_list(Tree,List,[]).

```

Obține lista nodurilor în preordine:

```

| ?- runp1(List).
List = [7,3,2,5,4,6,11] ? ;
no
| ?- runp2(List).
List = [6,3,2,4,11] ? ;
no
| ?- runp3(List).
List = [6,3,2,5,4,11] ? ;
no
| ?- runp4(List).
List = [7,3,2,1,5,4,6,11,8,10] ? ;
no
| ?- runp5(List).
List = [7,3,2,1,5,4,6,11,8,10|...] ? ;
no

```

7.1.3 Postordinea

Efectuând aceleași prelucrări în scopul obținerii listei cheilor în postordine:

```

generate_postorder_list(nil, []).
generate_postorder_list(t(Left,Key,Right),List):-
    generate_postorder_list(Left,ListLeft),
    generate_postorder_list(Right,ListRight),
    append(ListRight,[Key],Interm)
    append(ListLeft,Interm,List).

```

Se observă că acest caz este chiar mai ineficient decât precedentele; adăugarea la sfârșitul listei a unei chei necesită un apel suplimentar de `append`.

Cu liste diferență, varianta cu unificare explicită:

```

generate_postorder_list(nil,L,L).
generate_postorder_list(t(Left,Key,Right),First,Last):-
    generate_postorder_list(Left,FirstListLeft,LastListLeft),
    generate_postorder_list(Right,FirstListRight,LastListRight),
    First= FirstListLeft,
    LastListLeft=FirstListRight,
    LastListRight=[Key|Last].

```

Cu liste diferență, varianta cu unificare implicită:

```

generate_postorder_list(nil,L,L).
generate_postorder_list(t(Left,Key,Right),
                        FirstListLeft,LastListRight):-
    generate_postorder_list(Left,FirstListLeft,Interm),
    generate_postorder_list(Right,Interm,[Key|LastListRight]).

```

Și acest predicatul necesită un tip particular de apel inițial:

```
?-generate_postorder_list(Tree,List,[]).
```

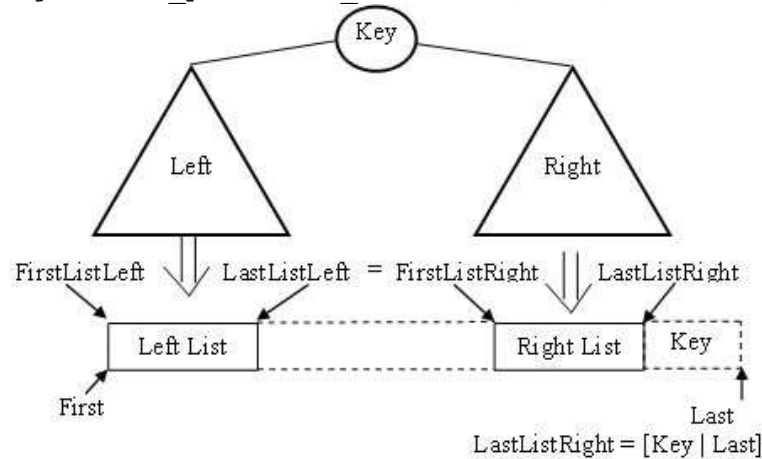


Fig. 7.5

Iar o rulare pe aceiași arbori

```

runpost1(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
    7,
    t(nil,11,nil)),
    generate_postorder_list(Tree,List,[]).

runpost2(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(nil,4,nil)),6,t(nil,11,nil)),
    generate_postorder_list(Tree,List,[]).

runpost3(List):-
    Tree=
    t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,nil)),6,t(nil,11,nil)),
    generate_postorder_list(Tree,List,[]).

runpost4(List):-

```



```

    Tree=
t(t(t(t(nil,1,nil),2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
7,
t(t(nil,8,t(nil,10,nil)),11,nil)),
    generate_postorder_list(Tree,List,[]).

runpost5(List):-
    Tree=
t(t(t(t(nil,1,nil),2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),
7,
t(t(nil,8,t(t(nil,9,nil),10,nil)),11,t(nil,12,nil))),
    generate_postorder_list(Tree,List,[]).

```

Furnizează lista nodurilor în postordine:

```

| ?- runpost1(List).
List = [2,4,6,5,3,11,7] ? ;
no
| ?- runpost2(List).
List = [2,4,3,11,6] ? ;
no
| ?- runpost3(List).
List = [2,4,5,3,11,6] ? ;
no
| ?- runpost4(List).
List = [1,2,4,6,5,3,10,8,11,7] ? ;
no
| ?- runpost5(List).
List = [1,2,4,6,5,3,9,10,8,12|...] ? ;
no

```

7.2 Algoritmi de tip *divide et impera*

O problemă reprezentativă, care este pusă în discuție atunci când se prezintă tehnica de programare *divide et impera* este problema ordonării unui șir de elemente, mai concret algoritmul de ordonare al lui Hoare, sortarea prin partiționare. Îl punem și noi acum în discuție, atât pentru a urmări tehnica *divide et impera* în specificație procedurală, cât și pentru a urmări specificitatea datorată prelucrării în acest model, și nu în cele din urmă avantajele pe care le aduc listele diferență în acest context.

Ne reamintim că reprezentarea structurii de date în varianta procedurală este cea de listă. Sortarea prin partiționare, ca algoritm reprezentativ al tehnicii divide et impera urmărește întocmai linia acestei tehnici:

```
%divide_et_impera/2
%divide_et_impera(intrare, iesire)

divide_et_impera(Source, Destination) :-
    elementary_Dimension(Source), !,
    direct_method(Source, Destination) .
divide_et_impera(Source, Destination) :-
    divide(Source, S1, S2, ..., Sn),
    divide_et_impera(S1, D1),
    divide_et_impera(S2, D2),
    ...,
    divide_et_impera(Sn, Dn),
    combine(D1, D2, ..., Dn, Destination) .
```

Unde predicatul `divide/n+1` realizează împărțirea domeniului de date de intrare `Source` în n subdomenii (la majoritatea algoritmilor n fiind 2), iar `combine/n+1` realizează combinarea rezultatelor obținute în urma apelurilor recursive pentru obținerea datelor de ieșire `Destination`. Reamintim că mulți algoritmi care se încadrează acestei tehnici au proprietatea că fie `divide/n+1` fie `combine/n+1` este operația vidă (se realizează implicit, fără nici un efort computațional). La sortarea prin partiționare, `divide/n+1` este partiționarea (în două sau mai multe secvențe de intrare), iar `combine/n+1` este operația vidă (în varianta imperativă sortarea realizându-se în aceeași zonă de memorie, combinarea se realizează implicit). Să ne focalizăm pe implementarea logică a algoritmului.

7.2.1 Sortarea prin partiționare

```
%quicksort/2
%quicksort(lista intrare, lista ordonata de iesire)

quicksort([H|T], Result) :-
    partition(H, T, Left, Right),
    quicksort(Left, SortedLeft),
    quicksort(Right, SortedRight),
    append(SortedLeft, [H|SortedRight], Result) .
```

```

quicksort([], []).

%partition/4
%partition(pivot, sursa, destinatie elemente mai mici
decat pivot, destinatie elemente mai mari decat pivot)

partition(X, [H|T1], [H|Left], Right) :-
    H < X, !,
    partition(X, T1, Left, Res).
partition(X, [H|T1], Left, [H| Right]) :-
    partition(X, T1, Left, Right).
partition(_, [], [], []).
| ?- quicksort([6,9,3,7,2], Result).
Result = [2,3,6,7,9] ?
yes

```

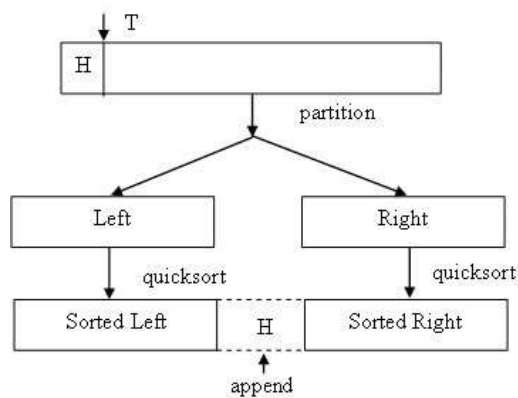


Fig. 7.6

Particularitatea limbajelor logice (reprezentarea structurilor de intrare și ieșire în entități distincte) impune reorganizarea datelor de ieșire după apelurile recursive, și deci, necesitatea predicatului `combine`, în cazul de față fiind reprezentat de `append/3`.

Predicatul `partition/4` realizează operația de divizare a listei de intrare, în lista conținând toate elementele din intrare care au proprietatea de a fi mai mici decât pivotul (elementul în funcție de care se realizează partiționarea) – elemente care se plasează în lista argument 3, prin intermediul clauzei 1, și respectiv lista conținând toate elementele din intrare care au proprietatea de a fi mai mari sau egale cu pivotul (comparația din prima clauză urmată de tăiere de backtracking ne scutește de a mai realiza comparația în clauza 2) – elemente care se plasează în lista argument 4, prin intermediul clauzei 2.

Ultima clauză asigură terminarea recursivității în cazul în care intrarea a devenit vidă (în această situație "închizând" și listele de ieșire, prin legarea cozii acelor liste la lista vidă []).

Sortarea propriu-zisă `quicksort/2`, setează pivotul la primul element al listei de intrare `H`, și formează cele două partiții `Left` și `Right` în funcție de valorile elementelor din intrare, prin comparație cu pivotul. Urmează două apeluri recursive, rezultatele acestor apeluri `SortedLeft` și `SortedRight` fiind liste ordonate. Cum aceste liste conțin elemente toate mai mici, respectiv mai mari sau egale decât pivotul, înseamnă că avem relația de ordine totală:

`SortedLeft <H <=SortedRight`

Acum putem să combinăm aceste rezultate parțiale pentru obținerea soluției, ceea ce realizează `append/3`. Deși acest apel final nu produce nici un neajuns eficienței algoritmului (rămânând în aceeași clasă, $O(n \lg n)$ în cazul mediu statistic, și respectiv $O(n^2)$ în cazul cel mai defavorabil), dorim să vedem dacă prin programare logică nu putem să evităm acest apel final. Am văzut în capitolele precedente cum listele terminate în variabilă elimină o seamă de neajunsuri. În acest context, listele diferență vor rezolva problema.

```
%qsort_LD1/3
%qsort_LD1(lista intrare, prim element lista iesire,
ultim element lista iesire)
```

```
qsort_LD1 ([H|T], LFirst, LLast):-
    partition(H, T, Left, Right),
    qsort_LD1 (Left, FirstLeft, LastLeft),
    qsort_LD1 (Right, FirstRight, LastRight),
    LFirst=FirstLeft,
    LLast=LastRight,
    LastLeft=[H|FirstRight].
qsort_LD1 ([], L, L).
```

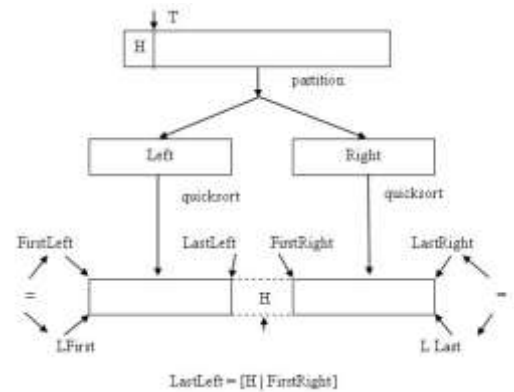


Fig. 7.7

```
%quicksort1/2
%quicksort1(lista intrare, lista ordonata de iesire)
```

```
quicksort1(List,Result):-
    qsort_LD1(List,Result,[]).

| ?- quicksort1([6,9,3,7,2],Result).
Result = [2,3,6,7,9] ?
yes
```

Predicatul `qsort_LD1` implementează problema în tehnica listelor diferență cu instanțiere explicită. Apelurile recursive din varianta clasică au aceeași listă (regulată) la intrare, obținută din partiționare (am omis rescrierea acestui predicat, care este identic ca în varianta inițială), în timp ce argumentele de ieșire sunt reprezentate ca pereche de argumente, liste diferență, cu prim și ultim element. După apeluri, unificările explicite `LFirst=FirstLeft`, `LLast=LastRight`, `LastLeft=[H|FirstRight]` produc instanțierea ieșirii. Apelul inițial specific (cere listă vidă ca al treilea argument) se realizează prin intermediul `quicksort2/2`.

```
%qsort_LD2/3
%qsort_LD2(lista intrare, prim element lista iesire,
ultim element lista iesire)
```

```
qsort_LD2([H|T],LFirst,LLast):-
    partition(H,T,Left,Right),
    qsort_LD2(Left,LFirst,[H|R]),
    qsort_LD2(Right,R,LLast).
qsort_LD2([],List,List).
```

```
%quicksort2/2
%quicksort2(lista intrare, lista ordonata de iesire)
```

```
quicksort2(List,Result):-
    qsort_LD2(List,Result,[]).

| ?- quicksort2([6,9,3,7,2],Result).
Result = [2,3,6,7,9] ?
yes
```

Predicatul `qsort_LD2/3` implementează aceeași problemă în tehnica listelor diferență cu instanțiere implicită, singura diferență față de `qsort_LD1/3` fiind momentul unificării argumentelor, în timp ce `quicksort2/2` realizează apelul inițial (care cere listă vidă ca al treilea argument).

Deși până acum am exemplificat listele diferență pentru cazul în care structura de ieșire se realizează prin instanțierea listelor diferență din două apeluri recursive (ale aceluiași predicat, cel din capul regulii), nu aceasta este unica situație posibilă. Mai mult, în practică sunt dese cazurile când sunt necesare mai multe apeluri (sau chiar mai puține, deci doar unul!) de predicate diferite. Structura generică pentru două apeluri ale aceluiași predicat este:

```
%differenceList_1/3
%differenceList_1(input, first output, last output)

differenceList_1(In,FirstOut,LastOut):-
    split(In,In1,In2),
    differenceList_1(In1,FirstOut,Interm),
    differenceList_1(In2,Interm, LastOut).
differenceList_1(empty,L,L).
```

Acest șablon poate suferi mici modificări atunci când în diferite poziții se mai adaugă anumite elemente (de exemplu la sortarea prin partiționare și la parcurgerea în inordine a arborilor `Interm` este afectat, în timp ce la parcurgerea în preordine și postordine elementele `FirstOut` și respectiv `LastOut` sunt afectate). Oricum șablonul de pasare a argumentelor este același: un apel transmite terminarea listei diferență apelului următor, în timp ce primul și respectiv ultimul apel își primesc/returnează valori de la/la capul regulii. Am precizat condiția de oprire în `empty` pentru că argumentul de intrare nu e obligatoriu o listă.

Apelul inițial se realizează:

```
?-differenceList_1(In, Out, []).
```

Dacă șablonul de mai sus îl generalizăm pentru n apeluri recursive ale aceluiași predicat, avem:

```
differenceList_2(In,FirstOut,LastOut):-
    split(In,In1,In2,In3,...,Inn),
```

```

differenceList_2(In1,FirstOut,Interm1),
differenceList_2(In2,Interm1,Interm2).
differenceList_2(In3,Interm2,Interm3),
...,
differenceList_2(Inn,Intermn,LastOut),
differenceList_2(empty,L,L).

```

cu apelul inițial:

```
?-differenceList_2(In, Out, []).
```

Am menționat că predicatele din apeluri pot fi distincte, și diferite de cele din capul regulii:

```

differenceList_3(In,FirstOut,LastOut):-
    split(In,In1,In2,In3,...,Inn),
    call_p1(In1,FirstOut,Interm1),
    call_p2(In2,Interm1,Interm2).
    call_p3(In3,Interm2,Interm3),
    ...,
    call_pn(Inn,Intermn,LastOut),
differenceList_3(empty,L,L).

```

cu apelul inițial:

```
?-differenceList_3(In, Out, []).
```

7.2.2 Exerciții și probleme propuse

1. Rescrieți `generate_ordered_list/2` pentru cazul arborilor terminați în variabile.
2. Rescrieți `generate_ordered_list/3` pentru cazul arborilor terminați în variabile.
3. Rescrieți `generate_preorder_list/2` pentru cazul arborilor terminați în variabile.
4. Rescrieți `generate_preorder_list/3` pentru cazul arborilor terminați în variabile.

5. Rescrieți `generate_postorder_list/2` pentru cazul arborilor terminați în variabile.

6. Rescrieți `generate_postorder_list/3` pentru cazul arborilor terminați în variabile.

7. Care este rezultatul unui apel

```
?-generate_ordered_list(Tree,List,[13,17]).
```

în cazul în care

```
Tree=t(t(t(nil,2,nil),3,t(nil,4,nil)),6,t(nil,11,nil)).
```

Justificați.

8. Rulați

```
| ?- quicksort([6,9,3,7,2],Result).
```

cu trasarea execuției.

9. Rulați

```
| ?- quicksort1([6,9,3,7,2],Result).
```

cu trasarea execuției. Urmăriți modul cum se instanțiază variabilele.

10. Modificați predicatul `quicksort1/2` sub forma:

```
quicksort1(List,Result):-
```

```
    qsort_LD1(List,Result,[1,2]).
```

și apoi rulați

```
?- quicksort1([6,9,3,7,2],Result).
```

Explicați rezultatul obținut.

11. Modificați predicatul `qsort_LD1/3` sub forma:

```
qsort_LD1([H|T],LFirst,LLast):-
```

```
    partition(H,T,Left,Right),
```

```
    qsort_LD1(Right,FirstRight,LastRight),
```

```
    qsort_LD1(Left,FirstLeft,LastLeft),
```

```
    LFirst=FirstLeft,
```

```
    LLast=LastRight,
```

```
    LastLeft=[H|FirstRight].
```

```
qsort_LD1([],L,L).
```

și apoi rulați

```
?- quicksort1([6,9,3,7,2],Result).
```

urmărind la trasare instanțierea variabilelor.

12. Rulați

?- quicksort2([6,9,3,7,2],Result).
cu trasarea execuției. Urmăriți modul cum se instanțiază variabilele.

13. Modificați predicatul quicksort2/2 sub forma:

```
quicksort2(List,Result):-
    qsort_LD2(List,Result,[1,2]).
```

și apoi rulați

?- quicksort2([6,9,3,7,2],Result).

Explicați rezultatul obținut.

14. Modificați predicatul qsort_LD2/3 sub forma:

```
qsort_LD2([H|T],LFirst,LLast):-
    partition(H,T,Left,Right),
    qsort_LD2(Right,R,LLast),
    qsort_LD2(Left,LFirst,[H|R]).
qsort_LD2([],List,List).
```

și apoi rulați

?- quicksort1([6,9,3,7,2],Result).

Urmărind la trasare instanțierea variabilelor.

15. Implementați un predicat partition3/6 care să realizeze partiționarea unei liste de intrare în trei partiții, în funcție de doi pivoți, P1, P2, cu proprietatea că $P1 < P2$, care la întrebarea:

?- partition3(4,7,[6,9,3,7,2,5,8],Small,Middle,Large).

Să returneze:

Small=[2,3], Middle=[5,6], Large=[7,8,9] ?

16. Implementați un predicat quicksort/2 care are factorul de divizare $n=3$ (utilizează predicatul partition3/6 din specificațiile de la exercițiul precedent).

17. Implementați și rulați cu trasare qsort_LD1/3 (cu trei apeluri recursive). Urmăriți instanțierea argumentelor liste diferență.

18. Implementați și rulați cu trasare qsort_LD2/3 (cu trei apeluri recursive). Urmăriți instanțierea argumentelor liste diferență.

7.3 Transformarea între diferite tipuri de liste

Remarcăm că până în acest moment am întâlnit deja trei tipuri de liste: complete, incomplete (terminate în variabilă) și diferență. În funcție de problema rezolvată, de datele de intrare furnizate, respectiv de datele de ieșire solicitate, am putea avea o situație în care manipularea simultană a diferitelor tipuri de listă să fie necesară (în fapt la varianta îmbunătățită a sortării prin partiționare, lucrăm atât cu liste complete cât și cu liste diferență, în acel caz fără a fi necesar a realiza transformări dintr-o formă în cealaltă). Să vedem cum putem realiza transformări între diferite tipuri de liste.

7.3.1 Listă completă la listă incompletă

```
%lcomplete2LTV/2
%lcomplete2LTV(lista completa, lista incompleta)

lcomplete2LTV([H|T],[H|R]):-
    lcomplete2LTV(T,R).
lcomplete2LTV([],_).

run1(L):- lcomplete2LTV([1,2,3,4,5,6,7],L).
| ?- run1(L).
L = [1,2,3,4,5,6,7|_A] ? ;
no
```

7.3.2 Listă incompletă la listă completă

```
% lTV2lcomplete/2
% lTV2lcomplete (lista completa, lista incompleta)

lTV2lcomplete([],L):-
    var(L),!.
lTV2lcomplete([H|T],[H|R]):-
    lTV2lcomplete(T,R).

run2(L):-lTV2lcomplete(L,[1,2,3,4,5,6,7|_]).
```

```
| ?- run2(L).
L = [1,2,3,4,5,6,7] ? ;
no
```

7.3.3 Listă completă la listă diferență

```
%lcomplete2ldiff/3
%lcomplete2ldiff(lista completa, inceputul listei
diferenta, sfarsitul listei diferenta)

lcomplete2ldiff([H|T],[H|R],L):-
    lcomplete2ldiff(T,R,L).
lcomplete2ldiff([],L,L).

run3(L):-lcomplete2ldiff([1,2,3,4,5,6,7],L,[]).

| ?- run3(L).
L = [1,2,3,4,5,6,7] ? ;
no
```

7.3.4 Listă diferență la listă completă

O primă soluție ar părea a fi:

```
%ldiff2lcomplete/3
%ldiff2lcomplete(lista completa, inceputul listei
diferenta, sfarsitul listei diferenta)

ldiff2lcomplete([],L,L).
ldiff2lcomplete([H|T],[H|R],L):-
    ldiff2lcomplete(T,R,L).
```

Dar surprinzător, soluția nu este cea așteptată. Dacă rulăm pe o întrebare de forma:

```
run4(Result):-
ldiff2lcomplete(Result,[1,2,3,4,5,6,7|L],L).
```

Rezultatul este surprinzător, și anume:

```
L = [] ? ;
L = [1] ? ;
L = [1,2] ? ;
```

```

L = [1,2,3] ? ;
L = [1,2,3,4] ? ;
L = [1,2,3,4,5] ? ;
L = [1,2,3,4,5,6] ? ;
L = [1,2,3,4,5,6,7] ? ;
L = [1,2,3,4,5,6,7,_A] ? ;
L = [1,2,3,4,5,6,7,_A,_B] ?
yes

```

Notă: yes-ul final sugerează existența și altor (infinit de multe în fapt) soluții.

Care este explicația? La apelul inițial variabila L (argumentul 3) se unifică (cu succes) cu variabila (argumentul 2). De ce? Prologul NU face occur check (nu verifică apariția variabilei care se unifică în argumentul cu care se unifică, adică în), și permite unificarea!

Soluția? Test de inedititate (==) a variabilelor.

```

ldiff2lcomplete_1([],L1,L2):-L1==L2,! .
ldiff2lcomplete_1([H|T],[H|R],L):-
    ldiff2lcomplete_1(T,R,L) .

run4_1(Result):-
    ldiff2lcomplete_1(Result,[1,2,3,4,5,6,7|L],L) .
| ?- run4_1(L) .
L = [1,2,3,4,5,6,7] ? ;
no

```

Se observă că transformarea din listă completă în listă diferență diferă de transformarea inversă doar prin numele predicatului și modalitatea de apel. Ceea ce ne permite să afirmăm că un același predicat (apelat corespunzător) poate realiza ambele transformări:

```

%interchange_lcomplete_ldiff/3
%interchange_lcomplete_ldiff(lista completa, inceputul
listei diferenta, sfarsitul listei diferenta)

interchange_lcomplete_ldiff([],L,L):-! .
interchange_lcomplete_ldiff([H|T],[H|R],L):-
    interchange_lcomplete_ldiff(T,R,L) .

```

Care apelat:

```

run5(L):-
    interchange_lcomplete_ldiff([1,2,3,4,5,6,7],L,[]) .

```

```
| ?- run5(L).
L = [1,2,3,4,5,6,7] ? ;
no
```

Realizează primul tip de transformare, în timp ce al doilea tip este realizat printr-un apel de forma:

```
run6(Result):-
interchange_lcomplete_ldiff(Result,[1,2,3,4,5,6,7|L],L).
| ?- run6(L).
L = [] ? ;
L = [1] ? ;
L = [1,2] ? ;
L = [1,2,3] ? ;
L = [1,2,3,4] ? ;
L = [1,2,3,4,5] ? ;
L = [1,2,3,4,5,6] ? ;
L = [1,2,3,4,5,6,7] ? ;
L = [1,2,3,4,5,6,7,_A] ? ;
L = [1,2,3,4,5,6,7,_A,_B] ?
Yes
```

Iar pentru schimbarea bidirecțională corectă:

```
interchange_lcomplete_ldiff_1([],L1,L2):-!.
interchange_lcomplete_ldiff_1([H|T],[H|R],L):-
    interchange_lcomplete_ldiff_1(T,R,L).

run5_1(L):-
interchange_lcomplete_ldiff_1([1,2,3,4,5,6,7],L,[]).

run6_1(Result):-
interchange_lcomplete_ldiff_1(Result,[1,2,3,4,5,6,7|L],L).

| ?- run5_1(L).
no
| ?- run6_1(L).
L = [1,2,3,4,5,6,7] ? ;
no
```

7.3.5 Listă incompletă la listă diferență

```
%lTV2ldiff/3
%lTV2ldiff(lista      incompleta,      inceputul      listei
diferenta, sfarsitul listei diferenta)
```

```

lTV2ldiff(V,L,L):-
    var(V),!.
lTV2ldiff([H|T],[H|R],L):-
    lTV2ldiff(T,R,L).

run7(L):-lTV2ldiff([1,2,3,4,5,6,7|_],L,[]).

| ?- run7(L).
L = [1,2,3,4,5,6,7] ? ;
no

```

7.3.6 Listă diferență la listă incompletă

Aceeași problemă ca și în cazul transformării în listă completă apare și de data aceasta. Prezentăm codul și rulările, fără explicații.

```

%ldiff2LTV/3
%ldiff2lcomplete(lista completa,
                  inceputul listei diferenta,
                  sfarsitul listei diferenta)

ldiff2LTV(V,L,L).
ldiff2LTV([H|T],[H|R],L):-
    ldiff2LTV(T,R,L).

run8(Result):-ldiff2LTV(Result,[1,2,3,4,5,6,7|L],L).
| ?- run8(L).
true ? ;
L = [1|_A] ? ;
L = [1,2|_A] ? ;
L = [1,2,3|_A] ? ;
L = [1,2,3,4|_A] ? ;
L = [1,2,3,4,5|_A] ? ;
L = [1,2,3,4,5,6|_A] ? ;
L = [1,2,3,4,5,6,7|_A] ? ;
L = [1,2,3,4,5,6,7,_A|_B] ? ;
L = [1,2,3,4,5,6,7,_A,_B|_C] ? ;
L = [1,2,3,4,5,6,7,_A,_B,_C|...] ?
yes

```

Problema se soluționează la fel, conducând la:

```
ldiff2LTV_1(V,L1,L2):-L1==L2,! .
ldiff2LTV_1([H|T],[H|R],L):-
    ldiff2LTV_1(T,R,L) .

run8_1(Result):-
ldiff2LTV_1(Result,[1,2,3,4,5,6,7|L],L) .

| ?- run8_1(L) .
L = [1,2,3,4,5,6,7|_A] ? ;
no
```

Ca și în cazul anterior se observă că transformarea din listă incompletă în listă diferență diferă de transformarea inversă doar prin numele predicatului și modalitatea de apel. Doar aparent: în cazul trecerii de la listă incompletă în listă diferență se testează `var(V), !.`, clauza trebuind să apară în mod obligatoriu prima, în timp ce pentru trecerea inversă se adaugă o variabilă liberă la sfârșitul listei care tocmai se creează (testul nemaiputându-se realiza). Această observație conduce la afirmația că un același predicat (apelat corespunzător) nu poate realiza în acest caz ambele transformări. Dacă am încerca:

```
%interchange_LTV_ldiff/3
%interchange_LTV_ldiff(lista_completa, inceputul_listei
diferenta, sfarsitul_listei_diferenta)

interchange_LTV_ldiff(V,L1,L2):-
    var(V),L1==L2,! .
interchange_LTV_ldiff([H|T],[H|R],L):-
    interchange_LTV_ldiff(T,R,L) .
```

Care apelat:

```
| ?- interchange_LTV_ldiff(L,[1,2,3|X],X) .
L = [1,2,3|_A] ? ;
no
```

Realizează un tip de transformare, în timp cealaltă transformare intră în buclă infinită

Evitarea buclei infinite se poate realiza printr-un artificiu de forma:

```
interchange_LTV_ldiff(V,L1,L2):-
    var(V), (L1==L2;L1=L2), ! .
interchange_LTV_ldiff([H|T],[H|R],L):-
    interchange_LTV_ldiff(T,R,L) .
```

Caz în care se inversează rolurile de insucces, obținându-se răspunsurile:

```
| ?- interchange_LTV_ldiff([1,2,3|_],L,[ ]).
L = [1,2,3] ? ;
no
| ?- interchange_LTV_ldiff(L,[1,2,3|X],X).
X = [1,2,3,1,2,3,1,2,3,1|...] ? ;
no
```

7.3.7 Concatenări între diferite tipuri de liste

Concatenarea unei liste complete (primul argument) cu una incompletă (al doilea argument) nu impune nici un fel de probleme, pentru că se parcurge doar primul argument, care fiind o listă completă, varianta originală a predicatului `append/3` este suficientă. Singura remarcă este că lista de ieșire va fi o listă incompletă (terminată în variabilă), iar dacă problema în care această concatenare intervine necesită ca argumentul de ieșire să fie o listă completă, atunci predicatul `lTV2lcomplete` trebuie apelat după concatenare, pentru a obține rezultatul ca o listă completă. Cu această observație, concatenarea unei liste complete cu o listă incompletă, rezultatul fiind o listă incompletă se realizează printr-un simplu apel de forma:

```
append_C_LTV(LC,LTV,RezLTV):-
    append(LC,LTV,RezLTV) .
```

Iar dacă rezultatul este dorit ca o listă completă:

```
append_C_LTV(LC,LTV,RezC):-
    append(LC,LTV,RezLTV) ,
    lTV2lcomplete(RezLTV, RezC) .
```

Este evidentă ineficiența acestei modalități de abordare a problemei, deoarece o dată se parcurge prima listă (LC) pentru concatenare, după care lista rezultat (RezLTV) deci în fapt LC și LTV) pentru transformarea listei incomplete în listă completă. În total două parcurgeri ale primului argument, și unul al celui de-al doilea. Dubla parcurgere a primului argument (la fel ca la `append3`) poate fi evitată prin inversarea ordinii apelurilor, sub forma:

```
append_C_LTV(LC,LTV,RezC):-
    lTV2lcomplete(LTV,LC2) ,
```



```
append(LC, LC2, RezC) .
```

Concatenarea unei liste incomplete cu una completă necesită un predicat special:

```
%append_ LTV_C/3
%append_ LTV_C(lista incompleta, lista completa, lista
completa - concatenarea primelor doua argumente)

append_ LTV_C(V, L, L) :-
    var(V), !.
append_ LTV_C([H|T], L, [H|R]) :-
    append_ LTV_C(T, L, R) .
```

Cu observația că testul de atingere a variabile finale este obligatoriu să se realizeze la fiecare pas al execuției; deci faptul (de terminare a recursivității) devine acum o clauză (datorită testului `var(V), !.`), iar această clauză trebuie obligatoriu să preceadă clauza recursivă.

Dacă rezultatul se dorește să se obțină sub forma unei liste incomplete, atunci înainte de apelul acestui predicat, argumentul al doilea se transformă în listă incompletă (prin intermediul `lcomplete2LTV`), și atunci practic sunt manipulate două liste incomplete, denumirea corectă a predicatului ar fi `append_ LTV_LTV`, având însă clauzele identice cu `append_ LTV_C`.

7.4 Transformări în arbori

Așa cum am mai menționat în capitolele precedente, programarea logică se pretează foarte bine pentru utilizarea ca limbaj de specificații: naturaletă trecerii de la gândirea și raționamentul uman la specificații procedurale, simplitatea implementării, cu foarte puține restricții, ușurința demonstrării corectitudinii în modelul procedural, face ca acest stil de programare să fie des utilizat pentru tranziția de la idee la implementarea propriu-zisă.

Vom argumenta afirmațiile anterioare printr-un exemplu, pentru care de la specificarea problemei, prin implementarea procedurală, demonstrare de corectitudine și execuție vom justifica de ce, de la necesitate la soluție drumul poate trece prin specificația procedurală, și un astfel de drum conferă certitudinea corectitudinii.

Pornim de la problema propusă în [Cormen, Leiserson, Rivest, 14.2-4, pag 229]: Arătați că orice arbore binar de căutare având n noduri poate fi transformat în orice alt arbore binar de căutare având n noduri, folosind $O(n)$ rotații.

7.4.1 Transformări prin rotații

Vom rezolva mai întâi o problemă mai simplă, urmând ca ulterior să rezolvăm problema specificată, cu mențiunea că problema rezolvată la primul pas se folosește ulterior. Și anume dorim să arătăm că orice arbore binar de căutare având n noduri se poate transforma într-o listă ordonată conținând cheile celor n noduri, folosind $O(n)$ rotații. O metodologie care ne asigură păstrarea execuției în limitele $O(n)$ este soluționarea problemei prin recursivitate liniară, adică existența unui singur apel recursiv pe fiecare ramură a arborelui de execuție. Cum problema prin însăși formularea ei sugerează oarecum soluția, rotațiile care se efectuează vor trebui să aibă următoarele proprietăți:

- să păstreze structura ordonată (prin rotații arborele să rămână de căutare),
- să se aplice cel mult o dată fiecărui nod, deoarece avem menționată cerința de păstrarea execuției în limitele $O(n)$.

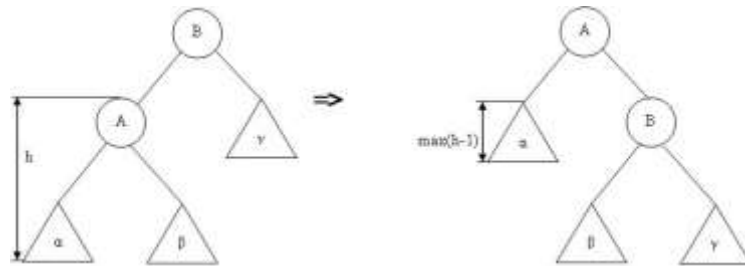


Fig. 7.8

Desenul de mai sus sugerează o rotație elementară către dreapta (în fapt este rotația simplă dreapta din cazul arborilor AVL).

Se observă că în urma unei operații elementare de rotație, subarborele Left pierde din înălțime cel puțin o unitate: dacă în forma inițială avea înălțimea h , după rotație înălțimea este fie $h-1$ (în cazul în care înălțimea era măsurată înainte de rotație din subarborele α) fie mai mică (dacă inițial măsurarea provenea din β). Dacă printr-o astfel de rotație înălțimea subarborelui stânga

scade cu 1, înseamnă că într-un număr h maxim de pași subarborele stânga va avea înălțimea 0, și deci arborele va avea forma:

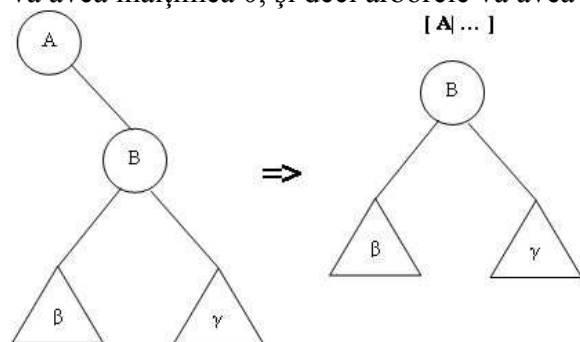


Fig. 7.9

În acest moment cheia din nodul rădăcină a arborelui (A) poate fi preluată în lista de ieșire, iar arborele de intrare poate fi considerat cel având rădăcina în B (renunțând la rădăcina A), cu subarborii săi. Acestui nou arbore (cu rădăcina B) i se aplică acum șablonul rotației, subarborele β urmând să se descompună după șablonul sugerat de modelul din partea stângă a desenului: $t(\alpha, A, \beta)$. După un număr finit de pași, din nou vom avea un nod izolat în rădăcină, pe care îl transferăm în lista de ieșire, iar subarborelui dreapta i se aplică din nou șablonul de descompunere a întregii structuri, etc. Până în acest moment am dovedit că într-un număr finit de rotații dreapta transformăm BST în listă ordonată crescător. Ce n-am dovedit însă este păstrarea execuției în limitele $O(n)$.

Vom dovedi păstrarea execuției în limitele $O(n)$ folosindu-ne în argumentație de primul desen. La trecerea din forma inițială, înainte de rotație, la forma de după rotație, există un nod care are o proprietate extrem de valoroasă: se poate afla într-o astfel de poziție în decursul tuturor etapelor de transformare cel mult o dată. Acest nod este cel cu cheia A. Un nod poate să fie nod rădăcină a subarborelui stânga cel mult o dată deoarece, dacă se află la un pas în acea poziție, prin rotația schițată el trece rădăcină a arborelui, și nici o rotație (toate sunt spre dreapta) nu va putea readuce nodul ca rădăcină a subarborelui stânga! Cum în arbore sunt n noduri, înseamnă că dacă, în cazul cel mai defavorabil (inițial arborele este degenerat stânga) fiecare nod este sau devine în urma rotațiilor rădăcină a subarborelui stânga, sunt necesare n rotații. În orice altă situație, sunt necesare mai puține.

Să trecem acum să realizăm specificațiile procedurale ale celor menționate anterior. Vom lua rând pe rând situațiile, și le vom transcrie procedural.

Menționăm încă o dată avantajul de a nu fi nevoiți să ne definim o structură de date, sau să realizăm declarații ale variabilelor, structurilor, procedurilor.

Definim mai întâi clauza (surprinzător, e una singură, un proces relativ complex este descris de o singură clauză Prolog!) care descrie rotația dreapta.

```
tree2chain(t(t(Alfa,A,Beta),B,Gama),Chain):-
    tree2chain(t(Alfa,A,t(Beta,B,Gama)),Chain).
```

Acum, dacă am ajuns la situația în care șablonul din stânga nu mai este aplicabil deoarece subarborele stânga e vid, înseamnă că am ajuns la șablonul specificat de desenul al doilea, care se rescrie procedural tot printr-o singură clauză:

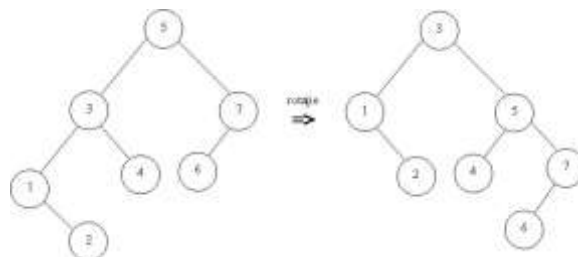
```
tree2chain(t(nil,A,t(Beta,B,Gama)),[A|RestChain]):-
    tree2chain(t(Beta,B,Gama),RestChain).
```

După cum se observă (în desen putem intui, clauza ne conferă certitudinea), descompunerea subarborelui dreapta $t(Beta,B,Gama)$ este inutilă, deoarece odată cu depunerea rădăcinii A în soluție se avansează (dreapta) în structură, așa cum este ea, ceea ce înseamnă că șablonul poate fi înlocuit printr-o structură mai generală Right, codul devenind astfel:

```
tree2chain(t(nil,A,Right),[A|RestChain]):-
    tree2chain(Right,RestChain).
```

Un singur pas ne desparte de terminarea problemei: terminarea recursivității. Condiția de terminare este cea banală, structura vidă de intrare transpunându-se în structură vidă de ieșire, adică:

```
tree2chain(nil,[]).
```



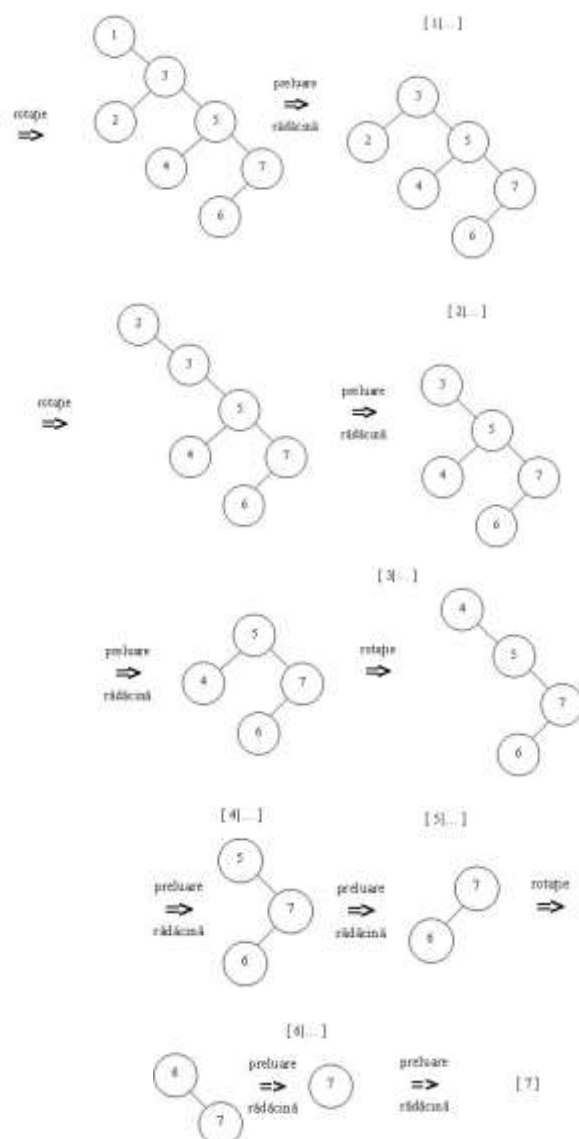


Fig. 7.10

Astfel, predicatul nostru este:

```
%tree2chain/2
%tree2chain(arbore, lista).
```

```

tree2chain(t(t(Alfa,A,Beta),B,Gama),Chain):-
    tree2chain(t(Alfa,A,t(Beta,B,Gama)),Chain).
tree2chain(t(nil,A,Right),[A|RestChain]):-
    tree2chain(Right,RestChain).
tree2chain(nil,[]).

```

O execuție simplă:

```

run1(Chain):-
    Tree=
    t(t(t(nil,2,nil),3,t(t(nil,4,nil),5,t(nil,6,nil))),7,t(nil,11,
nil)),
    tree2chain(Tree,Chain).

| ?- run1(Chain).
Chain = [2,3,4,5,6,7,11] ? ;
no

```

Ne vom focaliza acum pe o demonstrație mai riguroasă a corectitudinii (parțiale și totale) a secvenței descrise anterior. Fie $P(n,h)$ funcția booleană "Transformarea arborelui de n noduri, având înălțimea ramurii stânga h se realizează corect." Vom demonstra prin inducție că transformarea aplicată de clauzele predicatului `tree2chain/2` este corectă.

$P(0,0)$ adevărat.

Această afirmație este adevărată datorită clauzei a 3a. Faptul confirmă că un arbore fără nici un nod ($n=0$, deci implicit $h=0$) se transformă corect într-o listă ordonată crescător.

$P(n,h)$ adevărat $\Rightarrow P(n+1,0)$ adevărat

Această implicație este asigurată de clauza a două: dacă presupunem că un arbore cu n noduri și înălțime h se transformă corect într-o listă crescătoare, atunci și arborele care mai are în plus un nod, iar arborele din ipoteză ca subarbore dreapta, deci un arbore cu $n+1$ noduri și înălțimea ramurii stânga 0, se transformă corect în aceeași listă care are în plus cheia din noul nod rădăcină ca prim nod al listei.

$P(n,h)$ adevărat $\Rightarrow P(n,h+1)$

Această implicație este asigurată de prima clauză: dacă presupunem că un arbore cu n noduri și înălțime h se transformă corect într-o listă crescătoare, atunci și arborele care are exact aceleași noduri, dar înălțimea dreapta cu 1 mai mult se transformă corect în listă. Prima clauză face doar o rotație, păstrând exact aceleași noduri într-o altă ordine; oricum, arborele din capul clauzei are înălțimea ramurii stângi mai mare cu 1, deci terminarea execuției pentru un arbore de înălțime h implică terminarea pentru un arbore cu înălțimea $h+1$.

Deci:

(pasul de verificare)

$P(0,0)$ adevărat

(pașii de inducție)

$$\left. \begin{array}{l} P(n,h) \text{ adevărat} \Rightarrow P(n+1,0) \text{ adevărat} \\ P(n,h) \text{ adevărat} \Rightarrow P(n,h+1) \end{array} \right\} \Rightarrow P(n,h) \text{ adevărat, } \forall n, h \in \mathbb{N}.$$

Deci corectitudinea totală a secvenței este demonstrată.

Deocamdată am demonstrat că în timp liniar orice arbore se poate transforma într-o listă crescătoare. Pentru a rezolva problema propusă mai avem un pas, și anume să arătăm că un BST poate fi transformat în orice alt BST conținând aceleași noduri în timp liniar.

Dacă $T1$ este un arbore cu n noduri și $T2$ este un alt arbore cu n noduri, atunci:

$$\begin{array}{ccc} & \text{tree2chain}(T1, L1) & \\ T1 & \xrightarrow{\quad} & L1 \\ & O(n) & \end{array}$$

$$\begin{array}{ccc} & \text{tree2chain}(T1, L1) & \\ T2 & \xrightarrow{\quad} & L1 \\ & O(n) & \end{array}$$

Cum ambii arbori au fost transformați în timp liniar în același lanț spre dreapta (aceeași listă ordonată crescător), înseamnă că transformarea între $T1$ și $T2$ este tot liniară.

7.4.2 Transformare prin creare copie

O altă variantă a problemei transformă efectiv orice arbore într-un altul cu aceleași chei, prin generarea acestuia (ar fi echivalentul problemei anterioare cu pașii de transformări $T1 \rightarrow L1 \rightarrow T2$, deci a doua transformare ar fi "funcția" inversă a primeia). Soluția a doua a problemei, care respectă cerința de a fi o transformare liniară, se constituie în trei pași: primul creează o copie goală a unuia din arbori, al doilea creează lista crescătoare a cheilor celui alt arbore, iar pasul al treilea realizează umplerea arborelui gol (copie a primuia) cu valori din listă, deci cu chei din al doilea arbore. În felul acesta se creează o copie (ca formă) a primului arbore, cu chei (valori) din al doilea arbore.

Crearea "copiei goale" a arborelui se face simplu: se parcurge primul arbore, generând unul nou, având exact aceeași formă, dar nodurile conținând variabile libere, care se instanțiază la un pas următor. Este evident că această operație este liniară.

```
%tree2tree/2
%tree2tree(arbore vechi, arbore nou fara valori in
noduri).

tree2tree(nil,nil).
tree2tree(t(L,K,R),t(NL,NK,NR)):-
    tree2tree(L,NL),
    tree2tree(R,NR).
```

Pentru a "umple"arborele creat astfel, colectăm nodurile într-o listă. Acest lucru se poate realiza prin tehnica precizată la prima metodă de rezolvare a problemei (prin rotații), sau mai simplu, printr-o parcurgere în inordine a celui de-al doilea arbore. Dacă parcurgerea utilizează o listă diferență ca argument de ieșire, atunci se asigură obținerea listei nodurilor în timp liniar:

```
%tree2list/3
%tree2list(arbore de intrare, listă de iesire prim
element,listă de iesire ultim element).

tree2list(nil,L,L).
tree2list(t(L,K,R),First,Last):-
    tree2list(L,First,[K|Int]),
    tree2list(R,Int,Last).
```


Acum, cu cheile colectate în lista de ieșire a `tree2list/2`, umplem arborele de ieșire rezultat din `tree2tree/3`.

```
%list2tree/3
%list2tree(listă de intrare prim element,listă de
intrare ultim element, arbore de iesire).
```

```
list2tree(L,L,nil).
list2tree([H|T],T,t(nil,H,nil)).
list2tree(L1,[H|L2],t(L,H,nil)):-
    list2tree(L1,L2,L).
list2tree([H|L1],L2,t(nil,H,R)):-
    list2tree(L1,L2,R).
list2tree(L1,L2,t(L,H,R)):-
    list2tree(L1,[H|Li],L),
    list2tree(Li,L2,R).
```

O transformare completă:

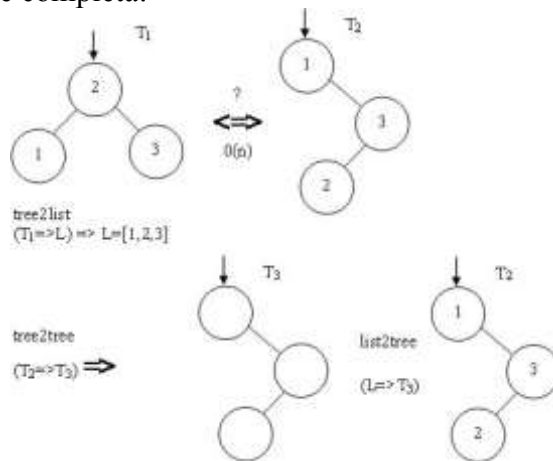


Fig. 7.11

```
?- T=t(t(nil,1,nil),2,t(nil,3,nil)),
    T2=t(nil,1,t(t(nil,2,nil),3,nil)),
    tree2list(T,L,[]),
    tree2tree(T2,T3),
    write(T3),nl,write(L),nl,
    list2tree(L,[],T3).
t(nil,_1676,t(t(nil,_1737,nil),_1710,nil))
[1,2,3]
L = [1,2,3],
```

```

T = t(t(nil,1,nil),2,t(nil,3,nil)),
T2 = t(nil,1,t(t(nil,2,nil),3,nil)),
T3 = t(nil,1,t(t(nil,2,nil),3,nil)) ?
Yes

```

7.4.3 Exerciții și probleme propuse

1. La justificarea timpului de execuție $O(n)$ pentru predicatul `tree2chain/2` în demonstrație s-a luat în considerare nodul B (rădăcina subarborelui stânga). Este corectă demonstrația dacă se reface raționamentul luând în considerare pentru justificare nodul A (rădăcina arborelui)? Argumentați.

2. Rescrieți predicatul `generate_list/2` în cazul arborilor incompleți (terminați în variabile). Ce se modifică? Argumentați.

3. Rescrieți predicatul `generate_list_dl/3` în cazul arborilor incompleți (terminați în variabile). Ce se modifică? Argumentați.

4. Rescrieți predicatul `find_key/2` în cazul arborilor incompleți (terminați în variabile). Ce se modifică? Argumentați.

5. Rescrieți predicatul `generate_list/2` în cazul arborilor incompleți (terminați în variabile), iar listele din noduri sunt de asemenea liste incomplete (terminate în variabile). Ce se modifică? Argumentați.

6. Rescrieți predicatul `generate_list_dl/3` în cazul arborilor incompleți (terminați în variabile), iar listele din noduri sunt de asemenea liste incomplete (terminate în variabile). Ce se modifică? Argumentați.

7. Rescrieți predicatul `find_key/2` în cazul arborilor incompleți (terminați în variabile), iar listele din noduri sunt de asemenea liste incomplete (terminate în variabile). Ce se modifică? Argumentați.

8. Se dă o listă completă ale cărei elemente sunt arbori binari de căutare. Generați lista ordonată a cheilor din întreaga structură, folosind predicatul `append/3`.

9. Refaceți exercițiul anterior, folosind ca tehnică de rezolvare listele diferență.

10. Pentru exercițiile anterioare, căutați după prima (și numai prima) apariție a unui anumit element în structura inițială.

11. Repetați precedentele trei exerciții pentru cazul în care lista este incompletă (terminată în variabilă).

12. Repetați exercițiul anterior pentru cazul în care arborii sunt incompleți (terminați în variabile).

8. Structuri complexe

8.1 Arbori de liste - structuri complete

Acum vom aborda o problemă aparent mai complexă, și vom vedea că specificațiile procedurale transformă problema într-una simplu de rezolvat. Dându-se un arbore binar de căutare ale cărui elemente sunt liste ordonate, construcția "structurii aplatizate", adică a listei plate, ordonate, care să conțină elementele tuturor listelor aflate la nivelul nodurilor arborelui, ar putea constitui o cerință. În primul rând să analizăm structura de date propusă, și să observăm că nu reprezintă altceva decât o particularizare a arborilor B (particularizare pentru că arborele rămâne binar, deci fiecare nod are maxim doi fii, în același timp reprezintă și o generalizare, deoarece numărul de elemente dintr-un nod – deci lungimea listei - nu este restricționat, cum nu se impune nici o restricție asupra înălțimii arborelui, sau mai precis asupra nivelului la care se află frunzele. Oricum, analogia cu arborii B rămâne valabilă). Mai jos se poate observa o astfel de structură.

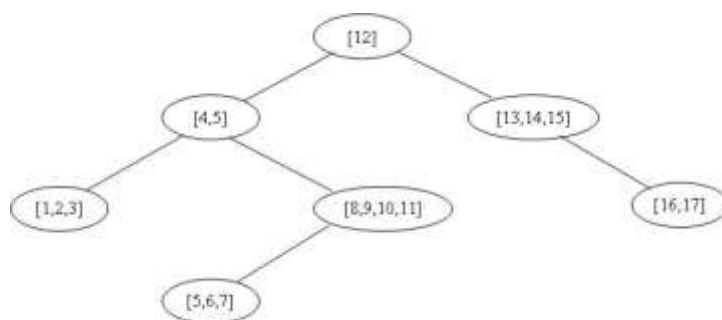


Fig. 8.1

8.1.1 Generare listă

Problema în discuție poate fi analizată și ca o variantă a uneia din problemele anterioare: în locul listei adânci (listă de liste) care se dorește a fi aplatizată, se cere să se aplatizeze o structură combinată, un arbore de liste.

```
%generate_list/2  
%generate_list(arbore de intrare,lista de iesire).
```

```

generate_list(t(Left,KeyList,Right),List):-
    generate_list(Left,LeftList),
    generate_list(Right, Right List),
    append3(LeftList,KeyList,RightList,List).
generate_list(nil,[]).

```

O analiză a de eficiență ne sugerează imediat utilizarea listelor diferență, cu evitarea concatenării din final, predicatul transformându-se în:

```

%generate_list_dl/3
%generate_list_dl(arbore de intrare,inceput lista de
iesire, sfarsit lista de iesire).

generate_list_dl(t(Left,KeyList,Right),
    FirstList,LastList):-
    generate_list_dl(Left,FirstLeft,LastLeft),
    lcomplete2ldiff(KeyList,FirstKey,LastKey),
    generate_list_dl(Right,FirstRight,LastRight),
    FirstLeft=FirstList,
    LastLeft= FirstKey,
    LastKey= FirstRight,
    LastList= LastRight.
generate_list_dl(nil,L,L).

```

O rulare a acestui predicat pe intrarea

```

run(List):-
    Tree=
    t(t(t(nil,[1,2],nil),[3],t(nil,[4,5],nil)),
    [6,7,8],
    t(nil,[10,11],nil)),
    generate_list_dl(Tree,List,[]).

```

Furnizează rezultatul:

```

| ?- run(List).
List = [1,2,3,4,5,6,7,8,10,11] ? ;
no

```

Unde predicatul `lcomplete2ldiff/3` este cel care realizează trecerea de la lista completă (formatul listei din noduri) la lista diferență, format necesar transformării.

```

%lcomplete2ldiff/3

```

```
%lcomplete2ldiff(lista completa, inceputul listei
diferenta, sfarsitul listei diferenta)
```

```
lcomplete2ldiff([H|T],[H|R],L):-
    lcomplete2ldiff(T,R,L).
lcomplete2ldiff([],L,L).
```

Varianta cu unificare explicită evidențiază toate unificările care se realizează, fiind mai ușor de urmărit. Unificările pot fi observate și pe desenul alăturat:

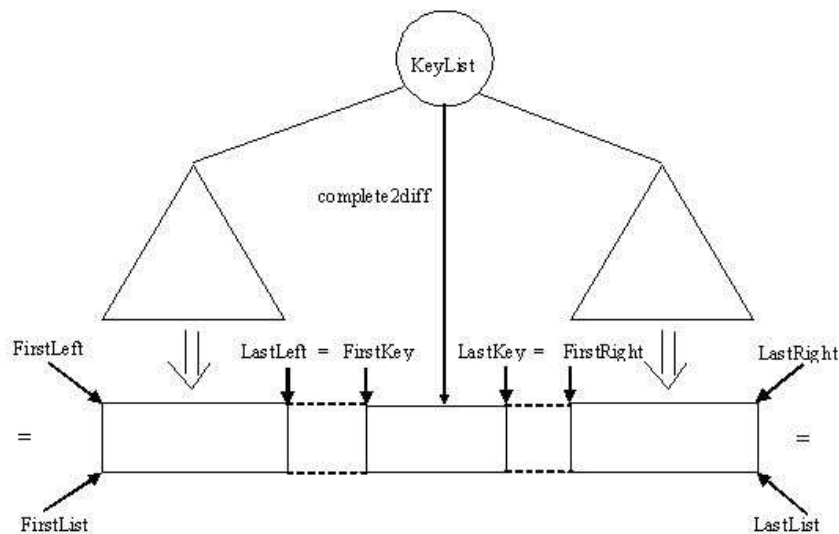


Fig. 8.2

Unificarea explicită este imediată:

```
generate_list_dl(t(Left,KeyList,Right),
    FirstList,LastList):-
    generate_list_dl(Left,FirstList,Int1),
    lcomplete2ldiff(KeyList,Int1, Int2),
    generate_list_dl(Right,Int2, LastList).
generate_list_dl(nil,L,L).
```

Se observă aceeași "pasare" a argumentelor listei diferență de la un apel la altul: cel mai din stânga apel recursiv (din punct de vedere a poziționării în structura de date, nu în ordinea apelurilor, care evident poate fi inversată, fără să afecteze rezultatul) primește ca parametru începutul listei rezultat (FirstList), al doilea său parametru (Int1) fiind un rezultat intermediar,

pasat următorului apel (`lcomplete2ldiff`). Acesta își completează rezultatul într-un nou parametru intermediar (`Int2`), transmis următorului apel (ultimul în exemplul nostru), și așa mai departe, până când se epuizează toate apelurile necesare.

8.1.2 Căutare

Identificarea primei apariții a unui anumit element în structura originală (de exemplu prima apariție a cheii 7), ar necesita o parcurgere condiționată în inordine a arborelui, urmărind schema de mai jos. Se observă că într-un subarbore stânga se continuă căutarea dacă și numai dacă cheia căutată este mai mică decât prima cheie din lista nodului curent, în timp ce în subarboarele dreapta se avansează doar dacă cheia căutată este mai mare decât ultima cheie din lista nodului curent.

```
%find_key/2
%find_key(cheia cautata, structura arborescenta in care
se realizeaza cautarea).

find_key(Key, t(Left, [H|RestKeyList], Right)) :-
    Key < H, !,
    find_key(Key, Left).
find_key(Key, t(Left, KeyList, Right)) :-
    member(Key, KeyList), !.
find_key(Key, t(Left, KeyList, Right)) :-
    last(KeyList, Last),
    Key > Last, !,
    find_key(Key, Right).

%last/2
%last(lista de intrare, ultimul element al listei)

last([_|T], Last) :-
    last(T, Last).
last([Last], Last) :- !.
```

Unde predicatul `last/2` returnează ultimul element al unei liste complete.

Este de menționat implementarea elegantă, specific declarativă a predicatului `last/2`, implementare care ne permite să subliniem încă o dată valențele predicatului `append/3`.


```
last(List, Last) :-
    append(_, [Last], List).
```

Paradoxal, predicatul are două clauze recursive, și o clauză care apelează un alt predicat, fără să aibă (aparent) nici o clauză de terminare a recursivității. Atunci cum se încheie căutarea (unde găsește elementul căutat)? Un apel de căutare se realizează sub forma:

```
run(Key, Tree) :-
    Tree =
    t(t(t(nil, [1, 2], nil), [3], t(nil, [4, 5], nil)),
    [6, 7, 8],
    t(nil, [10, 11], nil)),
    find_key(Key, Tree).
```

O căutare a unei chei existente în arbore se încheie cu succes, prin simpla raportare a acestui fapt, de forma:

```
| ?- run(5, Tree).
Tree = t(t(t(nil, [1, 2], nil), [3], t(nil, [4, 5], nil)),
[6, 7, 8],
t(nil, [10, 11], nil)) ? ;
no
```

Terminarea cu succes a execuției este asigurată de terminarea cu succes a predicatului `member/2`, care identifică (sau nu) existența elementului căutat în lista din nodul curent. Odată cu identificarea elementului, următoarea apariție nu mai este remarcată: așa cum se specifică, predicatul trebuie să identifice *prima și doar prima* apariție a elementului în structură, ceea ce face ca a doua apariție a elementului să nu mai fie identificată (la repetarea întrebării anterioare răspunsul este nu, chiar dacă elementul 5 mai este 0 dată prezent în structură).

Acest aspect este asigurat de tăierea de backtracking poziționată după apelul `member/2`.

Ce se întâmplă la execuția unei întrebări care solicită căutarea unei chei inexistente în arbore? Acest lucru ar trebui raportat ca eșec (fail la execuție), dar nu avem o clauză de forma:

```
find_key(Key, nil) :- fail.
```

Se returnează totuși răspunsul de "negăsire" a elementului căutat în structură? Cum se realizează?

Să urmărim o trasare, și să căutăm explicația:

```
| ?- run(9,Tree).
      1      1 Call: run(9,_481) ?
      2      2 Call: _481=
t(t(t(nil,[1,2],nil),[3],t(nil,[4,5],nil)),[6,7,8],t(nil,[10,11],nil))?
      2      2 Exit:
t(t(t(nil,[1,2],nil),[3],t(nil,[4,5],nil)),[6,7,8],t(nil,[10,11],nil))
=t(t(t(nil,[1,2],nil),[3],t(nil,[4,5],nil)),[6,7,8],t(nil,[10,11],nil)) ?
      3      2 Call: find_key(9,
t(t(t(nil,[1,2],nil),[3],t(nil,[4,5],nil)),[6,7,8],t(nil,[10,11],nil))) ?
      4      3 Call: 9<6 ?
      4      3 Fail: 9<6 ?
      5      3 Call: member(9,[6,7,8]) ?
      6      4 Call: member(9,[7,8]) ?
      7      5 Call: member(9,[8]) ?
      8      6 Call: member(9,[]) ?
      8      6 Fail: member(9,[]) ?
      7      5 Fail: member(9,[8]) ?
      6      4 Fail: member(9,[7,8]) ?
      5      3 Fail: member(9,[6,7,8]) ?
      9      3 Call: last([6,7,8],_3252) ?
     10      4 Call: last([7,8],_3252) ?
     11      5 Call: last([8],_3252) ?
     12      6 Call: last([],_3252) ?
     12      6 Fail: last([],_3252) ?
     11      5 Exit: last([8],8) ?
?      10      4 Exit: last([7,8],8) ?
?      9      3 Exit: last([6,7,8],8) ?
     13      3 Call: 9>8 ?
     13      3 Exit: 9>8 ?
     14      3 Call: find_key(9,t(nil,[10,11],nil)) ?
     15      4 Call: 9<10 ?
     15      4 Exit: 9<10 ?
     16      4 Call: find_key(9,nil) ?
     16      4 Fail: find_key(9,nil) ?
     14      3 Fail: find_key(9,t(nil,[10,11],nil)) ?
      3      2 Fail: find_key(9,
t(t(t(nil,[1,2],nil),[3],t(nil,[4,5],nil)),[6,7,8],t(nil,[10,11],nil))) ?
      1      1 Fail: run(9,_481) ?

no
```

Este foarte simplu: odată ce cheia căutată nu a fost găsită pe ramura pe care ar fi putut să se afle, și căutarea a ajuns la un arbore vid, simpla lipsă a clauzei corespunzătoare arborelui vid produce eșecul.

8.2 Arbori de liste - structuri incomplete

Lucrurile se complică puțin dacă listele din noduri sunt liste incomplete, adică structura are forma:

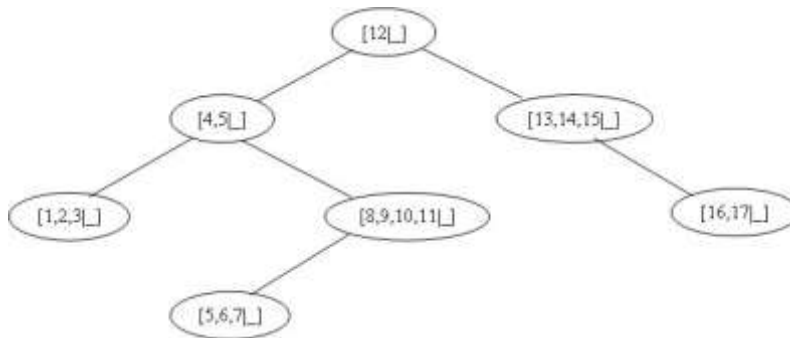


Fig. 8.3

Căutarea în primul rând trebuie să aibă în vedere că listele sunt incomplete, și deci predicatul `member` trebuie adaptat, altfel căutarea în prima listă ar produce inserarea. Deci predicatul de căutare `member/2` din versiunea anterioară, trebuie înlocuit cu:

```

%member_LTV/2
%member_LTV(element, listă)

member_LTV(H,L):-var(L),!,fail.
member_LTV(H,[H|_]).
member_LTV(H,[_|T]):-
    member_LTV(H,T).
  
```

Mai mult, căutarea pe ramura dreaptă fiind condiționată de inegalitatea `Key>Last` predicatul trebuie adaptat pentru a manevra o listă incompletă. Predicatul devine:

```

%last_LTV /2
%last_LTV (lista de intrare, ultimul element al listei)

last_LTV([Last|V],Last):-var(V),!.
last_LTV(_|T,Last):-
    last_LTV(T,Last).
  
```

Notă: ca la toate predicatele care manevrează structuri incomplete, condiția de terminare a recursivității (ajungerea la variabila finală) este prima care se testează (se plasează prima în lista clauzelor), altfel, se intră în buclă infinită pe variabila finală. Ca atare, dacă la structura completă, poziționarea clauzelor sub forma

```
last([_|T], Last) :-
    last(T, Last) .
last([Last], Last) :- !.
```

sau sub forma

```
last([Last], Last) :- !.
last([_|T], Last) :-
    last(T, Last) .
```

este indiferentă, prima fiind mai eficientă (evitând testarea la fiecare pas dacă s-a ajuns la lista cu un singur element), pentru listele incomplete, ordinea clauzelor

```
last_LTV([Last|V], Last) :- var(V), !.
last_LTV([_|T], Last) :-
    last_LTV(T, Last) .
```

este obligatorie, o ordonare de forma

```
last_LTV([_|T], Last) :-
    last_LTV(T, Last) .
last_LTV([Last|V], Last) :- var(V), !.
```

producând intrarea în ciclu infinit.

8.2.1 Căutare

Cu aceste observații, predicatul de căutare a unui element într-o structură arbore, cu elementele din noduri liste incomplete devine:

```
%find_key_LTV/2
%find_key_LTV(cheia cautata, structura arborescenta cu
liste incomplete in care se realizeaza cautarea).
```

```

find_key_LTV(Key,t(Left,[H|RestKeyList],Right)):-
    Key<H,! ,
    find_key_LTV(Key,Left).
find_key_LTV (Key,t(Left,KeyList,Right)):-
    member_LTV(Key,KeyList),!.
find_key_LTV(Key,t(Left,KeyList,Right)):-
    last_LTV(KeyList,Last),
    Key>Last,! ,
    find_key_LTV(Key,Right).

%member_LTV/2
%member_LTV(element, lista incompleta)

member_LTV(H,L):-var(L),!,fail.
member_LTV(H,[H|_]).
member_LTV(H,[_|T]):-
    member_LTV(H,T).
%last_LTV /2
%last_LTV (lista incompleta de intrare, ultimul element
-exceptand variabila finala- al listei incomplete)

last_LTV([Last|V],Last):-var(V),!.
last_LTV([_|T],Last):-
    last_LTV(T,Last).

```

O întrebare pentru căutarea unei chei în structura curentă:

```

run_LTV(Key,Tree):-
    Tree=
t(t(t(nil,[1,2|_],nil),[3|_],t(nil,[4,5|_],nil)),
[6,7,8|_],
t(nil,[10,11|_],nil)),
    find_key_LTV(Key,Tree).

```

Iar execuția (cu trasare la căutarea cu success, și simplă la cea cu eșec):

```

| ?- run_LTV(5,Tree).
    1          1 Call: run_LTV(5,_481) ?
    2          2 Call:
    _481=t(t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil)),
[6,7,8|_1088],
t(nil,[10,11|_1080],nil)) ?
    2          2 Exit:
t(t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil)),
[6,7,8|_1088],
t(nil,[10,11|_1080],nil))
=t(t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil)),

```

```

[6,7,8|_1088],
t(nil,[10,11|_1080],nil)) ?
      3      2 Call: find_key_LTV(5,
t(t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil)),
[6,7,8|_1088],
t(nil,[10,11|_1080],nil))) ?
      4      3 Call: 5<6 ?
      4      3 Exit: 5<6 ?
      5      3 Call: find_key_LTV(5,
t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil))) ?
      6      4 Call: 5<3 ?
      6      4 Fail: 5<3 ?
      7      4 Call: member_LTV(5,[3|_1102]) ?
      8      5 Call: var([3|_1102]) ?
      8      5 Fail: var([3|_1102]) ?
      9      5 Call: member_LTV(5,_1102) ?
     10      6 Call: var(_1102) ?
     10      6 Exit: var(_1102) ?
      9      5 Fail: member_LTV(5,_1102) ?
      7      4 Fail: member_LTV(5,[3|_1102]) ?
     11      4 Call: last_LTV([3|_1102],_5233) ?
     12      5 Call: var(_1102) ?
     12      5 Exit: var(_1102) ?
     11      4 Exit: last_LTV([3|_1102],3) ?
     13      4 Call: 5>3 ?
     13      4 Exit: 5>3 ?
     14      4 Call: find_key_LTV(5,t(nil,[4,5|_1094],nil)) ?
     15      5 Call: 5<4 ?
     15      5 Fail: 5<4 ?
     16      5 Call: member_LTV(5,[4,5|_1094]) ?
     17      6 Call: var([4,5|_1094]) ?
     17      6 Fail: var([4,5|_1094]) ?
     18      6 Call: member_LTV(5,[5|_1094]) ?
     19      7 Call: var([5|_1094]) ?
     19      7 Fail: var([5|_1094]) ?
?      18      6 Exit: member_LTV(5,[5|_1094]) ?
?      16      5 Exit: member_LTV(5,[4,5|_1094]) ?
     14      4 Exit: find_key_LTV(5,t(nil,[4,5|_1094],nil)) ?
      5      3 Exit: find_key_LTV(5,
t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil))) ?
      3      2 Exit: find_key_LTV(5,
t(t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil)),
[6,7,8|_1088],
t(nil,[10,11|_1080],nil))) ?
      1      1 Exit: run_LTV(5,
t(t(t(nil,[1,2|_1104],nil),[3|_1102],t(nil,[4,5|_1094],nil)),
[6,7,8|_1088],
t(nil,[10,11|_1080],nil))) ?
Tree = t(t(t(nil,[1,2|_A],nil),[3|_B],t(nil,[4,5|_C],nil)),
[6,7,8|_D],
t(nil,[10,11|_E],nil)) ?
yes
| ?- run_LTV(9,Tree).
no

```

8.2.2 Generare listă

Aplatizarea structurii, în cele două versiuni, cu append și cu liste diferență nu are nimic specific până în punctul în care trebuie să se producă concatenarea listelor: listele produse de apelurile recursive sunt liste complete, în timp ce lista din nodul rădăcină este una incompletă. Deci avem de acomodat versiunea anterioară concatenării unui triplet de liste: completă, incompletă, completă.

```

append3_LTV(L1,L2,L3,Rezult):-
    append_LTV_C(L2,L3,Int),
    append(L1,Int,Rezult).

%append_LTV_C/3
%append_LTV_C(lista incompleta, lista completa, lista
completa - concatenarea primelor doua argumente)

append_LTV_C(V,L,L):-
    var(V),!.
append_LTV_C([H|T],L,[H|R]):-
    append_LTV_C(T,L,R).

%generate_list_LTV/2
%generate_list_LTV(arbore de intrare,lista de iesire).

generate_list_LTV(t(Left,KeyList,Right),List):-
    generate_list_LTV(Left,LeftList),
    generate_list_LTV(Right,RightList),
    append3_LTV(LeftList,KeyList,RightList,List).
generate_list_LTV(nil,[]).

Și execuția:
run_LTV(List):-
    Tree=
t(t(t(nil,[1,2|_],nil),[3|_],t(nil,[4,5|_],nil)),
[6,7,8|_],
t(nil,[10,11|_],nil)),
    generate_list_LTV(Tree,List).

| ?- run_LTV(List).
List = [1,2,3,4,5,6,7,8,10,11] ? ;
no

```

Varianta cu liste diferență trebuie să aibă în vedere transformarea listei de la nivelul nodurilor, din liste incomplete în liste diferență în rest, predicatul de aplatizare a structurii rămâne același.

```
generate_list_dl(t(Left,KeyList,Right),
                FirstList,LastList):-
    generate_list_dl(Left,FirstList,Int1),
    lTV2ldiff(KeyList,Int1, Int2),
    generate_list_dl(Right,Int2, LastList).
generate_list_dl(nil,L,L).

%lTV2ldiff/3
%lTV2ldiff(lista incompleta, inceputul listei
diferenta, sfarsitul listei diferenta)

lTV2ldiff(V,L,L):-
    var(V),!.
lTV2ldiff([H|T],[H|R],L):-
    lTV2ldiff(T,R,L).

run_LTV_dl(List):-
    Tree=
t(t(t(nil,[1,2|_],nil),[3|_],t(nil,[4,5|_],nil)),
  [6,7,8|_],
t(nil,[10,11|_],nil)),
    generate_list_dl_LTV(Tree,List,[]).

| ?- run_LTV_dl(List).
List = [1,2,3,4,5,6,7,8,10,11] ? ;
no
```

8.3 Liste adânci

Lipsa tipizării Prologului permite manipularea unor structuri de date complexe, eterogene, a căror formă nu trebuie declarată sau cunoscută apriori. Avantajul este acela că se pot forma structuri complexe, a căror formă evoluează după necesitate, fără a fi nevoie de precauții legate de rezervare de spațiu, declararea formei sau dimensiunii. Aceste structuri sunt manipulate apoi ca atare, indiferent de conținutul lor. Spre exemplu, presupunem existența unei structuri de date de tip listă adâncă, adică listă a cărei elemente pot fi elemente atomice sau, recursiv, liste adânci. Un exemplu de astfel de structură este:


```
L=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]]
```

În exemplul anterior am reprezentat toate elementele unei liste având valoarea egală cu nivelul imbricării (adâncimii) la care se află lista. O transformare elementară a acestei structuri ar fi cea prin care s-ar trece de la lista adâncă la lista plată, adică:

```
L=[1,1,2,2,2,3,3,4,3,2,2,3,4,5,5,5,2]
```

Această transformare s-ar putea realiza cu un predicat de forma:

```
%deep2flat/2
%deep2flat(lista adanca, lista aplatizata).

deep2flat([],[]) :-!.
deep2flat(H,[H]) :-
    atomic(H),!.
deep2flat([H|T],[H|FlattenedTail]) :-
    atomic(H),!,
    deep2flat(T,FlattenedTail).
deep2flat([H|T],FlattenedList) :-
    deep2flat(H,FlattenedHead),
    deep2flat(T,FlattenedTail),
    append(FlattenedHead,FlattenedTail,FlattenedList).

run1(DL,FL) :-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]],
    deep2flat(DL,FL).

| ?- run1(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4|...]],2]],
FL = [1,1,2,2,2,3,3,4,3,2|...] ? ;
no
```

Deoarece reprezentarea este trunchată după un anumit număr de elemente, exemplificăm cu o rulare pe o listă care conține câte un element la intrarea (și eventual și la ieșirea) în nivelul respectiv de imbricare, adică:

```
run2(DL,FL) :-
    DL=[1,[2,[3,[4],3],2,[3,[4,[5]]],2]],
    deep2flat(DL,FL).
```

```
| ?- run2(DL,FL).
DL = [1,[2,[3,[4],3],2,[3,[4,[5]]],2]],
FL = [1,2,3,4,3,2,3,4,5,2] ? ;
no
```

Dacă îndepărtăm neajunsul concatenării din final, utilizând liste diferență (varianta cu unificare implicită), atunci avem:

```
%deep2flat_dl/2
%deep2flat_dl(lista adanca, lista aplatizata).

deep2flat_dl([],L,L):-!.
deep2flat_dl(H,[H|Last],Last):-
    atomic(H),!.
deep2flat_dl([H|T],[H|FlattenedTail],Last):-
    atomic(H),!,
    deep2flat_dl(T,FlattenedTail,Last).
deep2flat_dl([H|T],FlattenedList,Last):-
    deep2flat_dl(H,FlattenedList,Int),
    deep2flat_dl(T,Int,Last).
```

Dacă se dorește ca din lista originală să se creeze o listă plată conținând doar primele elemente ale oricărei liste, adică pentru:

```
L=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]]
```

Să se genereze:

```
L=[1,2,3,4,3,4,5]
```

Atunci predicatul Prolog ar trebui să evidențieze preluarea pentru prima dată a unui element dintr-o listă nouă. Acest aspect poate fi evidențiat fie prin utilizarea unui nou predicat (care să părăsească execuția normală după ce capul unei liste a fost transmis soluției), fie să se folosească o variabilă suplimentară (un flag) care să marcheze întotdeauna dacă din lista curentă (nivelul curent de imbricare) capul a fost preluat. Prezintă pe rând ambele variante. Predicatul `heads_deep2flat/2` prelucrează lista adâncă originală; dacă primul element al listei este atomic (clauza 3), acesta va face parte din componența listei rezultat, dar nici un alt nivel de pe nivelul curent de imbricare (adâncime a listei) nu mai este luat în considerare. Acest lucru este asigurat prin faptul că se adaugă `H` în rezultat, apelându-se apoi `heads_deep2flat_from_tail /2` e coada listei de intrare, predicat care ignoră toți atomii listei.

```
%heads_deep2flat/2
%heads_deep2flat(lista adanca, lista aplatizata).
```

```

heads_deep2flat([], []):-!.
heads_deep2flat(H, []):-
    atomic(H),!.
heads_deep2flat([H|T],[H|FlattenedTail]):-
    atomic(H),!,
    heads_deep2flat_from_tail(T,FlattenedTail).
heads_deep2flat([H|T],FlattenedList):-
    heads_deep2flat(H,FlattenedHead),
    heads_deep2flat_from_tail(T,FlattenedTail),
    append(FlattenedHead,FlattenedTail,FlattenedList).

%heads_deep2flat_from_tail /2
%heads_deep2flat_from_tail(lista adanca,
                           lista aplatizata).

heads_deep2flat_from_tail([], []):-!.
heads_deep2flat_from_tail([H|T],FlattenedTail):-
    atomic(H),!,
    heads_deep2flat_from_tail(T,FlattenedTail).
heads_deep2flat_from_tail([H|T],FlattenedList):-
    heads_deep2flat(H,FlattenedHead),
    heads_deep2flat_from_tail(T,FlattenedTail),
    append(FlattenedHead,FlattenedTail,FlattenedList).

```

Urmărend rulările pentru crearea listei capurilor de listă:

```

run5(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat(DL,HFL).

| ?- run5(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4|...]],2]],
FL = [1,2,3,4,3,4,5] ? ;
no

```

adică exact elementele marcate la care ne așteptăm: $L=[\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{3}, \underline{4}, \underline{5}]$

Alte rulări:

```

run6(DL,HFL):-
DL=[1,1,[2,2,2,[3,3,[4],3],2,[[4],3],2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat(DL,HFL).

| ?- run6(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,[[4],3],2,[3|...]|...]],
FL = [1,2,3,4,4,3,4,5] ? ;
no

```

```

run7(DL,HFL):-
DL=[1,1,[2,2,2,[3,3,[4],3],2,[[4],7,8],2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat(DL,HFL).

| ?- run7(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,[[4],7,8],2,[3|...]|...]],
FL = [1,2,3,4,4,3,4,5] ? ;
no

```

Acțiunile specifice pentru al doilea predicat (`heads_deep2flat_from_tail /2`) pot fi implementate în predicatul original cu ajutorul unei variabile suplimentare (`flag`), care să primească o valoare când se intră într-o listă nouă, și o altă valoare când se avansează într-o listă al cărei cap a fost deja preluat. Concret, se observă că predicatele `heads_deep2flat/2` și `heads_deep2flat_from_tail/2` din varianta precedentă se deosebesc doar prin adăugarea respectiv neadăugarea lui `H` în lista rezultat atunci când `H` e un element atomic, discriminare realizându-se în funcție de cazul `H` este sau nu este primul element atomic într-o listă adâncă. Adăugând un al treilea parametru, `flagul`, cu semantica: `flag = 0` dacă elementul atomic aflat în capul listei curente trebuie adăugat, și `1` în caz contrar, putem observa că în fapt trebuie dublată doar clauza 3 a predicatului, odată cu argumentul 3 având valoarea `0`, caz în care `H` se adaugă la rezultat, respectiv cu argumentul 3 cu valoarea `1`, caz în care `H` nu se adaugă la rezultat. Rezultă:

```

%heads_deep2flat_2/3
%heads_deep2flat_2(lista adanca, lista aplatizata,
flag).

heads_deep2flat_2([],[],_):-!.
heads_deep2flat_2(H,[],_):-
    atomic(H),!.
heads_deep2flat_2([H|T],[H|FlattenedTail],0):-
    atomic(H),!,
    heads_deep2flat_2(T,FlattenedTail,1).
heads_deep2flat_2([H|T],FlattenedTail,1):-
    atomic(H),!,
    heads_deep2flat_2(T,FlattenedTail,1).
heads_deep2flat_2([H|T],FlattenedList,_):-
    heads_deep2flat_2(H,FlattenedHead,0),
    heads_deep2flat_2(T,FlattenedTail,1),
    append(FlattenedHead,FlattenedTail,FlattenedList).

```

```

run8(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat_2(DL,HFL,0).

run9(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,[[4],3],2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat_2(DL,HFL,0).

run10(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,[[4],7,8],2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat_2(DL,HFL,0).

| ?- run8(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4|...]],2]],
FL = [1,2,3,4,3,4,5] ? ;
no

| ?- run9(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,[[4],3],2,[3|...]|...]],
FL = [1,2,3,4,4,3,4,5] ? ;
no

| ?- run10(DL,FL).
DL                                     =
[1,1,[2,2,2,[3,3,[4],3],2,[[4],7,8],2,[3|...]|...]],
FL = [1,2,3,4,4,3,4,5] ? ;
no

```

Eliminând din variantele precedente concatenarea din final, prin utilizarea listelor diferență obținem:

```

%heads_deep2flat_dl/2
%heads_deep2flat_dl(lista adanca, lista aplatizata).

heads_deep2flat_dl([],L,L):-!.
heads_deep2flat_dl(H, L,L):-
    atomic(H),!.
heads_deep2flat_dl([H|T],[H|FlattenedTail],Last):-
    atomic(H),!,
    heads_deep2flat_dl_from_tail(T,FlattenedTail,Last).
heads_deep2flat_dl([H|T],FlattenedList,Last):-
    heads_deep2flat_dl(H,FlattenedList,Int),
    heads_deep2flat_dl from_tail(T,Int,Last).

```

```

%heads_deep2flat_dl_from_tail /2
%heads_deep2flat_dl_from_tail(lista      adanca,      lista
aplatizata).

heads_deep2flat_dl_from_tail([],L,L):-!.
heads_deep2flat_dl_from_tail([H|T],FlattenedTail,Last):-
    atomic(H),!,
    heads_deep2flat_dl_from_tail(T,FlattenedTail,Last).
heads_deep2flat_dl_from_tail([H|T],FlattenedList,Last):-
    heads_deep2flat_dl(H, FlattenedList,Int),
    heads_deep2flat_dl_from_tail(T,Int,Last).

run11(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat_dl(DL,HFL,[],0).

run12(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,[[4],3],2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat_dl(DL,HFL,[],0).

| ?- run11(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4|...]],2]],
FL = [1,2,3,4,3,4,5] ? ;
no

| ?- run12(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,[[4],3],2,[3|...]|...]],
FL = [1,2,3,4,4,3,4,5] ? ;
no

```

Și respectiv:

```

heads_deep2flat_dl_2([],L,L):-!.
heads_deep2flat_dl_2(H,L,L):-
    atomic(H),!.
heads_deep2flat_dl_2([H|T],[H|FlattenedTail],Last,0):-
    atomic(H),!,
    heads_deep2flat_dl_2(T,FlattenedTail,Last,1).
heads_deep2flat_dl_2([H|T],FlattenedTail,Last,1):-
    atomic(H),!,
    heads_deep2flat_dl_2(T,FlattenedTail,Last,1).
heads_deep2flat_dl_2([H|T],FlattenedList,Last,_):-
    heads_deep2flat_dl_2(H, FlattenedList,Int,0),
    heads_deep2flat_dl_2(T,Int,Last,1).

```

```

run13(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat_dl_2(DL,HFL,[],0).

run14(DL,HFL):-
    DL=[1,1,[2,2,2,[3,3,[4],3],2,[4],3],2,[3,[4,[5,5,5]]],2]],
    heads_deep2flat_dl_2(DL,HFL,[],0).

| ?- run13(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4|...]],2]],
FL = [1,2,3,4,3,4,5] ? ;
no

| ?- run14(DL,FL).
DL = [1,1,[2,2,2,[3,3,[4],3],2,[4],3],2,[3|...]|...]],
FL = [1,2,3,4,4,3,4,5] ? ;
no

```

8.4 Exerciții și probleme propuse

1. Transformați predicatul `deep2flat/2` astfel încât la aplatizare să treacă în rezultat doar elementele de pe nivel par/împar. Adică întrebări de forma:

```
?-deep2flat_even([1,1,[2,2,[3,3,[4],3],2,[3,[4,[5,5,5]]],2]],R).
```

să returneze

```
L=[2,2,4,2,4,2],
```

iar

```
?-deep2flat_odd([1,1,[2,2,[3,3,[4],3],2,[3,[4,[5,5,5]]],2]],R).
```

să returneze:

```
L=[1,1,3,3,3,3,5,5,5].
```

2. Transformați predicatul `deep2flat` varianta cu `append/3` astfel încât să funcționeze pentru liste de intrare adânci, terminate în variabilă, de forma: `[1,1,[2,2,[3,3,[4|_],3|_]2|_]_]`.

3. Rescrieți predicatul de la punctul anterior cu liste diferență.

4. Transformați predicatul `heads_deep2flat` astfel încât să funcționeze pentru liste de intrare adânci, terminate în variabilă. Scrieți mai întâi varianta Prolog cu `append/3`, apoi pe cea cu liste diferență.

5. Simetric cu `heads_deep2flat` scrieți predicatul care returnează lista aplatizată a elementelor atomice aflate pe ultima poziție într-o listă, adică pentru intrarea:

```
[1, 1, [2, 2, [3, 3, [4], 3], 2, [3, [4, [5, 5, 5]]], 2]],
```

returnează la ieșire:

```
[4, 3, 5, 2].
```

Scrieți mai întâi varianta Prolog cu `append/3`, apoi pe cea cu liste diferență.

6. Rescrieți predicatul de la punctul anterior pentru liste terminate în variabile.

7. Scrieți un predicat care contorizează numărul de liste dintr-o listă adâncă.

8. Scrieți un predicat care returnează lista aflată la ceam mai mare adâncime (nivel de imbricare) într-o listă adâncă, adică pentru lista de intrare:

```
[1, 1, [2, 2, [3, 3, [4], 3], 2, [3, [4, [5, 5, 5]]], 2]]
```

returnează

```
[5, 5, 5].
```

9. Scrieți un predicat care contorizează numărul de liste aflate pe un anumit nivel într-o listă adâncă.

10. Scrieți un predicat care aplatizează (prin adăugarea doar a elementelor atomice de pe acel nivel) listele aflate pe un anumit nivel într-o listă adâncă. Adică dacă lista de intrare este:

```
[1, 1, [2, 2, [3, 3, [4], 3], 2, [3, [4, [5, 5, 5]]], 2]]
```

iar nivelul care se dorește a fi aplatizat este 3, atunci să returneze:

```
[3, 3, 3, 3].
```


9. Accesarea și manipularea programului

Programele Prolog analizate până acum s-a presupus că sunt rezidente în memoria calculatorului; n-am pus în discuție modalitatea de reprezentare internă, și nici modalitatea în care clauzele program au ajuns acolo. În prelucrarea simbolică, multe aplicații sunt dependente de accesarea și manipularea clauzelor program; aceasta înseamnă completul acces la clauze. Dacă se dorește modificarea programelor la execuție, atunci trebuie pusă la dispoziție o metodă de adăugare și înlăturare de clauze program chiar în timpul execuției (în fapt permisiunea de modificare a codului în timpul execuției sale, deci metaprogramare).

Prin intermediul predicatelor de adăugare și respectiv de eliminare a clauzelor se iese din nou din sfera programării logice pure, aceste predicate fiind considerate extra-logice, ele introducând noțiunea de efecte laterale.

9.1 Efecte laterale: *assert* și *retract*

Pentru adăugarea de noi clauze există predicatele sistem *asserta* (sau simplu *assert*) și *assertz*: prima formă adaugă clauza transmisă ca și parametru ca și prima clauză a predicatului corespunzător (adică a predicatului cu același nume și aritate. Dacă un astfel de predicat nu există, îl creează prin specificarea clauzei asertate ca și unică regulă – pentru moment – a predicatului corespunzător), în timp ce *assertz* adaugă clauza transmisă ca și parametru ca și ultimă clauză a predicatului corespunzător (cu aceeași mențiune ca și înainte).

De exemplu `assert(parent(ion,maria))` va adăuga programului faptul dat ca și argument. Dacă programul conține deja predicatul `parent/2`, atunci adaugă acest fapt înaintea oricărei clauze a predicatului; dacă pe de altă parte predicatul nu are nici o definiție (nu există nici o regulă a sa în program) atunci faptul va fi pentru moment singura regulă de definiție a predicatului. Dacă se adaugă clauze complete, atunci un rând suplimentar de paranteze sunt necesare pentru a dezambigua eventualele probleme legate de precedența termenilor. De exemplu forma,

```
assert((father(X,Y):-male(X),parent(X,Y)))
```

este o sintaxă corectă, în timp ce forma

```
assert(father(X,Y):-male(X),parent(X,Y))
```

nu este o sintaxă corectă (datorită lipsei parantezelor din jurul clauzei-argument).

Predicatele sistem `asserta` și `assert` sunt identice (deși au nume diferite, au semantici identice). Predicatul `assertz` are comportament simetric cu al celor menționate anterior, doar că adăugarea se realizează ca ultimă clauză a predicatului cu același nume și aceeași aritate (respectiv ca singura clauză, în cazul în care nu are nici o definiție până în acel moment). Dacă la apelul unui astfel de predicat argumentul este neinstanțiat (adică în `assert(Clause)`, `Clause` este neinstanțiat, sau are forma `Head:-Body` cu `Head` și `Body` neinstanțiat) atunci se produce o eroare la execuție.

Predicatul sistem `retract` înlătură din program prima clauză a programului care se unifică cu argumentul lui `retract`. Predicatul are un singur argument, reprezentând un termen cu care clauza (care se elimină din program) trebuie să se unifice. Termenul trebuie să fie suficient instanțiat ca predicatul clauzei să fie determinat. Odată determinată (identificată) clauza (cu care termenul se unifică), această clauză se elimină din program (dacă nu există nici o clauză care să satisfacă condițiile de unificare, atunci `retract` eșuează). Dacă se face o tentativă de resatisfacere a predicatului `retract(Term)` (fie datorită eșecului unui scop ulterior lui, fie datorită backtrackingului în tentativa de căutare de soluții alternative), Prolog caută din acel punct (din care s-a produs eliminarea clauzei) căutând o altă clauză cu care termenul să se unifice. Dacă este găsită o astfel de clauză lucrurile se repetă ca și în cazul precedent (clauza este eliminată din program, păstrându-se poziția, pentru ca în caz de revenire să se reia căutarea din acel punct). Și așa mai departe. De subliniat faptul că o clauză odată eliminată nu mai este niciodată readăugată; nici la backtracking (dimpotrivă, se mai elimină o clauză a predicatului respectiv dacă e posibil!) nici la eșec explicit (fail). Deci, spre deosebire de `assert` (care nu eșuează niciodată, și nu face backtracking), `retract` dimpotrivă, face backtracking, și poate și să eșueze.

Programele Prolog care fac apel la predicatele sistem `assert` și `retract` sunt programe care utilizează efectele laterale. Deși rezultatele obținute prin intermediul acestora sunt deseori spectaculoase, codul depinzând de efecte laterale este greu de citit, dificil de depanat și raționamentele formale implicate de acestea sunt complexe. Din acest punct de vedere, utilizarea acestor predicate este controversată; există voci care interpretează utilizarea lor ca incompetență sau "lene intelectuală" (ultima afirmație fiind susținută prin faptul că multe programe Prolog pot fi rescrise utilizând `assert` și

`retract`, efortul scrierii acestor forme de cod fiind substanțial mai scăzut, dar, cu dezavantajul unui cod mai puțin limpede, și categoric mai puțin eficient).

Totuși, se poate furniza o anumită justificare logică utilizărilor limitate, în cazuri specifice, a efectelor laterale. Adăugarea unei noi clauze este absolut justificată atunci când acea clauză urmează, ca o implicație logică a programului existent, care rulează. Adăugarea sa nu va afecta în nici un fel înțelesul programului, deoarece nici o consecință nouă nu derivă din noua clauză adăugată. Mai mult, adăugarea sa poate duce la creșterea performanței (codul devine mai eficient) deoarece anumite consecințe se pot deriva mai rapid, prin intermediul ei. Simetric, eliminarea unei clauze este justificată în cazul în care acea clauză este redundantă din punct de vedere logic, într-o astfel de situație, eliminarea clauzei poate fi văzută ca o colectare de reziduuri, în scopul reducerii dimensiunii programului, și creșterii eficienței.

Efectele laterale au fost prezentate ca mecanisme care permit adăugarea și eliminarea unor fapte și clauze care definesc predicate; acestea au însă și valoarea de memorare a unor structuri ordinare, construite în timpul execuției programului. Orice structură necesară într-un punct al secvenței Prolog se transmitea, prin intermediul argumentelor, de la un predicat la altul. Dacă într-o anumită zonă a programului este necesară o anumită informație care este furnizată de o altă zonă a programului (predicatele care creează, respectiv doresc să utilizeze informația respectivă nu se apelează unul pe altul, deci nu există posibilitatea pasării informației ca argumente, prin intermediul variabilei partajate), aceasta implică "încărcarea" tuturor predicatelor dintre cele două puncte cu argumente suplimentare, care să realizeze "transportul" informației între cele două puncte. O altă alternativă este însă stocarea acelei informații (prin intermediul efectelor laterale, și a unui predicat auxiliar), prelungind astfel durata de viață a structurilor memorate în acest fel. Astfel se obține un efect similar celui al variabilelor globale din limbajele imperative (variabilele locale corespunzând variabilelor ascunse în Prolog, cele care apar în corp, fără să apară în capul clauzei). Un exemplu de utilizare a efectelor laterale în scopul simulării variabilelor globale este prezentat în lucrare la arborii AVL (informația referitoare la modificarea înălțimii unui arbore fiind stocată în acest fel).

9.2 Arbori AVL

Este prezentată în continuare soluția unei probleme cunoscute pentru îmbunătățirea performanțelor în operațiile de bază într-un arbore binar de căutare. Aceasta se bazează pe menținerea sa într-o anumită stare de "echilibru" (arbori AVL), care asigură menținerea înălțimii acestor arbori între anumite limite (care vor fi specificate și justificate în continuare). Reamintim că arborii de căutare reprezintă structuri specializate la categoria arborilor, a căror introducere este justificată de reducerea numărului de operații efectuate, reducere de la $O(n)$ la liste la $O(h)$ la arbori de căutare. Arborii AVL reprezintă încă un pas în procesul de specializare, asigurând pentru înălțimea arborelui relația: $\lg n \leq h \leq 1.45 \lg n$, care este o reală creștere de performanță față de arborii binari de căutare (BST) pentru care $\lg n \leq h \leq n$.

O soluție pentru îmbunătățirea performanțelor operațiilor de bază asupra arborilor de căutare constă în menținerea lor în stare de echilibru, evaluând în același timp costul pe care menținerea echilibrului îl induce (creșterea performanței în efectuarea operațiilor de baza să nu conducă în cele din urmă la degradarea performanței induse de menținerea structurii). Din acest punct de vedere există două nivele de ierarhizare a echilibrului, în funcție de înălțime și respectiv în funcție de numărul de noduri, prima fiind o variantă mai puțin strictă a echilibrului în arbore care asigură apartenența înălțimii într-un domeniu mai larg (dar de ordin $\sim \lg n$), dar cu un cost mai redus al menținerii. Definiția echilibrului în funcție de înălțime a fost formulată de Adelson-Velski și Landis, și anume:

Un arbore de căutare este echilibrat după înălțime dacă și numai dacă pentru fiecare nod înălțimile celor doi subarbori ai săi diferă cu cel mult 1.

Deci arborii AVL (îi vom denumi cu acest nume în continuare) au proprietățile următoare:

- sunt BST (arborii binari de căutare) (deci mulțimea AVL este submulțime BST)
- înălțimea subarborilor unui AVL diferă prin cel mult 1.

Observație: orice subarbore a unui AVL este un AVL, deci proprietatea legată de înălțime se referă la **fiecare** nod al arborelui, nu doar la rădăcină.

Operațiile de căutare, inserare și ștergere efectuate asupra unui arbore AVL se pot efectua cu un efort de ordinul $\log_2 N$, chiar și în situația cea mai

defavorabilă. Această afirmație se bazează pe o teoremă demonstrată de Adelson-Velski și Landis, teoremă care garantează că un arbore AVL nu va depăși înălțimea arborelui corespunzător perfect echilibrat cu mai mult de 45%, indiferent de numărul de noduri.

Arborii care satisfac condiția de mai sus se numesc arbori AVL (după inițialele celor care i-au propus). Definiția este simplă, cere un efort mic de păstrare a echilibrului, și adâncimea căii de căutare în arbore este practic identică cu cea dintr-un arbore perfect echilibrat.

Cei mai dezechilibrați arbori AVL pot fi definiți astfel: notăm T_h arborele AVL cu număr minim de noduri, și cu înălțime h . T_0 este arborele vid și T_1 este arborele cu un singur nod. Pentru a construi arborele T_h , cu $h > 1$, vom adăuga la rădăcină doi subarbori care au la rândul lor un număr minim de noduri.

$$\begin{aligned} T_0 &= \text{nil} \\ T_1 &= t(\text{nil}, K_1, \text{nil}) \\ T_2 &= t(T_1, K_2, \text{nil}) \\ T_3 &= t(T_2, K_3, T_1) \\ T_4 &= t(T_3, K_4, T_2) \\ &\dots \\ T_n &= t(T_{n-1}, K_n, T_{n-2}) \end{aligned}$$

Se observă imediat că soluția simetrică este posibilă (cea în care subarborii stânga și dreapta se interschimbă). Criteriul de construcție a sugerat denumirea acestei secvențe de arbori definită inductiv, arbori Fibonacci. Astfel, numărul de noduri N_k în arborele T_k este:

$$N_0=0, \quad N_1=1, \quad N_k=N_{k-1}+1+N_{k-2}$$

Inserarea în arbori AVL

În continuare vom discuta modul de menținere a echilibrului într-un AVL în cazul inserării unui nod, cu demonstrarea (informală) a corectitudinii, urmată de specificarea predicativă a operației.

Inserarea unui nod în AVL se poate face în subarborele stânga (dacă cheia de inserat e mai mică decât cea din nodul rădăcină) respectiv în dreapta (în caz

contrar). Notăm cu E echilibrul nodului curent, reprezentând diferența dintre înălțimile celor doi subarbori, adică:

$E = h_{\text{Right}} - h_{\text{Left}}$, deci:

$$E = \begin{cases} -1 & \text{dacă } h_{\text{Right}} < h_{\text{Left}} \text{ (mai exact: } h_{\text{Left}} = h_{\text{Right}} + 1) \\ 0 & \text{dacă } h_{\text{Right}} = h_{\text{Left}} \\ +1 & \text{dacă } h_{\text{Right}} > h_{\text{Left}} \text{ (mai exact: } h_{\text{Right}} = h_{\text{Left}} + 1) \end{cases}$$

Vom efectua analiza în detaliu pentru cazul inserării în stânga, cazul al doilea fiind absolut simetric. La inserarea pe stânga poate apare unul din următoarele trei cazuri:

1. $h_{\text{Left}} = h_{\text{Right}}$: subarborii Left și Right devin de înălțimi inegale dar criteriul de echilibru nu este violat.
2. $h_{\text{Left}} < h_{\text{Right}}$: echilibru este îmbunătățit.
3. $h_{\text{Left}} > h_{\text{Right}}$: criteriul de echilibru este violat și este necesară restructurarea arborelui.

Deci două din trei cazuri NU afectează criteriul legat de echilibru, și deci NU necesită nici o operație suplimentară față de inserarea propriu-zisă (al cărei timp $O(h)$ este dat de timpul necesar găsirii poziției în care se realizează inserarea).

În cazul inserării care produce un dezechilibru pe ramura stângă distingem următoarele două cazuri distincte (vom justifica după specificarea cazurilor de ce acestea sunt singurele posibile): descompunem subarboarele stânga în subarborii constitutivi, și aici din nou putem insera în stânga sau în dreapta. Fiecare din aceste cazuri este analizat separat, și este schițat.

Cazul 1 (îl vom denumi generic LeftLeft, deoarece inserarea face primii doi pași din rădăcină spre stânga Fig. 9.1). Arborele (înainte de inserare) are proprietatea deordonare (în paranteză sunt reprezentați subarborii, cu semnificația că aceeași relație este valabilă pentru subarboare însuși) :

$$\begin{matrix} (\alpha) < \mathbf{A} < (\beta) < \mathbf{B} < (\gamma) \\ \mathbf{h} & & \mathbf{h} & & \mathbf{h} \end{matrix}$$

Inserarea LeftLeft conduce la creșterea înălțimii subarborelui α la $\mathbf{h}+1$, care NU afectează echilibrul subarborelui cu rădăcina \mathbf{A} , care ar trece în -1 (după cum se observă pe desenul din partea stângă). Dar pentru arborele cu rădăcina \mathbf{B} subarboarele stâng are înălțimea $\mathbf{h}+2$ iar cel drept \mathbf{h} , ceea ce ar conduce la un dezechilibru -2 la nivelul nodului \mathbf{B} , dezechilibru în afara limitelor.

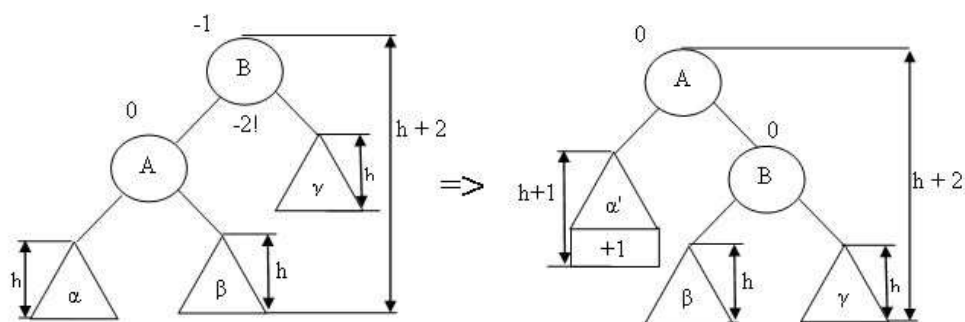


Fig. 9.1

$$\begin{array}{ccccc}
 (\alpha') & < & \mathbf{A} & < & (\beta) & < & \mathbf{C} & < & (\gamma) \\
 h+1 & & & & h & & & & h \\
 \leftarrow h+1 \rightarrow & & & & \leftarrow \text{---} h+1 \text{---} \rightarrow & & & & \\
 \leftarrow \text{-----} h+2 \text{-----} \rightarrow & & & & & & & &
 \end{array}$$

Dar păstrând relația de ordonare (pentru BST) și analizând înălțimile subarborilor observăm că subarborile cu rădăcina **B** are aceeași înălțime cu subarborile α' , și deci o soluție convenabilă ar fi cea în care nodul **A** devine rădăcina a arborelui, caz în care subarborii săi ar avea aceeași înălțime ($h+1$), ceea ce înseamnă că echilibrul nodului ar deveni 0, situația ideală (reamintim că dacă un nod are echilibrul, indiferent de ramura pe care se realizează inserarea la pasul următor echilibrul nodului rămâne în limite acceptabile). Transformarea poate fi urmărită pe desenul din dreapta. Arborele AVL rezultat are două proprietăți remarcabile:

- înălțimea sa este $h+2$, adică aceeași ca și a arborelui de intrare ÎNAINTE de inserare. Aceasta înseamnă că dacă arborele cu rădăcina **B** este subarbore într-un arbore mai mare, acesta din urmă nu este afectat (nu sesizează) inserarea în discuție. Proprietatea este extrem de importantă, deoarece ne asigură că aceasta transformare (o vom numi rotație simplă dreapta; simplă pentru că implică o singură pereche de noduri, **AB**, respectiv dreapta pentru că perechea de noduri are o rotație spre dreapta) este singulară: dacă ea se efectuează, nici o altă transformare nu mai este necesară la inserarea curentă.
- echilibrul nodului rădăcină al arborelui rezultat este 0, ceea ce asigură că nici la o inserare ulterioară nu vor fi necesare reechilibrări.

Aceste observații ne pot permite să formulăm următoarea

Lema 1: Inserarea cu rotație simplă dreapta nu mai necesită nici o altă transformare la inserarea curentă, și nici la cea următoare.

Cazul 2 (îl vom denumi generic LeftRight, deoarece inserarea face primul pas din rădăcină spre stânga, apoi spre dreapta Fig. 9.2). Arborele (înainte de inserare) are proprietatea deordonare (în paranteză sunt reprezentați subarborii, cu semnificația că aceeași relație este valabilă pentru subarborii însuși) :

$$\begin{array}{ccccccc} (\alpha) < A < (\beta) < B < (\gamma) < C < (\delta) \\ h+1 & & h & & h & & h+1 \end{array}$$

Inserarea LeftRight conduce la creșterea înălțimii subarborului β (δ) la $h+1$, care NU afectează echilibrul subarborului cu rădăcina B , care ar trece în -1 (după cum se observă pe desenul din partea stânga), și nici pe cea a arborelui cu rădăcina A , a cărui echilibru devine $+1$. Dar pentru arborele cu rădăcina C subarborii stâng are înălțimea $h+3$ iar cel drept $h+1$, ceea ce ar conduce la un dezechilibru -2 la nivelul nodului C , dezechilibru în afara limitelor. Dacă inserarea se produce în subarborii β , acesta transformându-se în β' , cu înălțimea $h+1$, atunci se poate observa că:

$$\begin{array}{ccccccc} (\alpha) < A < (\beta') < B < (\gamma) < C < (\delta) \\ \wedge & & \wedge & & \wedge & & \wedge \\ h+1 & & h+1 & & h & & h+1 \\ \leftarrow \text{---} h+2 \text{---} \rightarrow & & \leftarrow \text{---} h+2 \text{---} \rightarrow & & & & \\ \leftarrow \text{---} h+3 \text{---} \rightarrow & & & & & & \end{array}$$

Iar dacă inserarea se realizează în subarborii γ , acesta devenind în urma inserării γ' , cu înălțimea $h+1$, atunci avem:

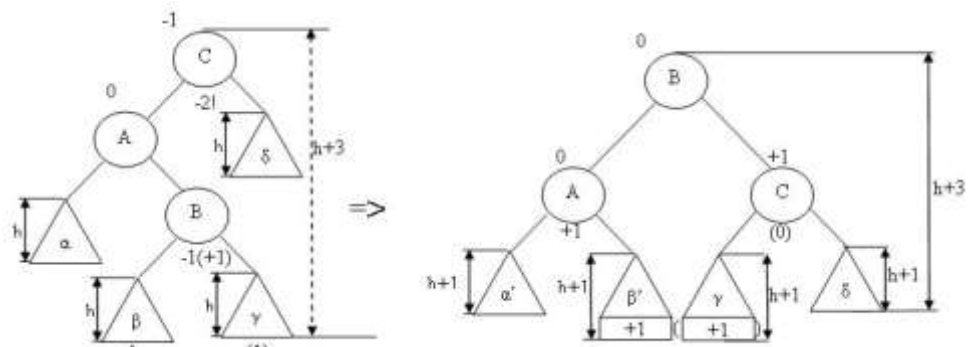
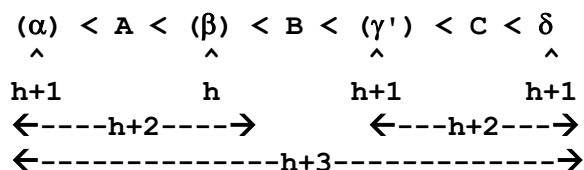


Fig. 9.2



Vom face analiza cu explicații detaliate pentru cazul inserării în β (deosebirile la inserarea în δ vor fi specificate în paranteze, la fel cum sunt specificate și pe desen). Păstrând relația de ordonare (pentru BST) și analizând înălțimile subarborilor observăm că subarboarele cu rădăcina \mathbf{A} are aceeași înălțime cu subarboarele \mathbf{C} , și deci o soluție convenabilă ar fi cea în care nodul \mathbf{A} devine rădăcină a arborelui, caz în care subarborii săi ar avea aceeași înălțime ($h+2$), ceea ce înseamnă că echilibrul nodului ar deveni 0, situația ideală. Transformarea poate fi urmărită pe desenul din dreapta. Arborele AVL rezultat are două proprietăți remarcabile:

- înălțimea sa este $h+3$, adică aceeași ca și a arborelui de intrare ÎNAINTE de inserare. Aceasta înseamnă că dacă arborele cu rădăcina \mathbf{C} este subarbor într-un arbore mai mare, acesta din urmă nu este afectat (nu sesizează) inserarea în discuție. Proprietatea este extrem de importantă, deoarece ne asigură că aceasta transformare (o vom numi rotație dublă dreapta; dublă pentru că implică două perechi de noduri, \mathbf{AB} , pe de o parte, și respectiv \mathbf{AC} pe de altă parte, dreapta pentru că perechile de noduri sunt angrenate într-o rotație spre dreapta) este singulară: dacă ea se efectuează, nici o altă transformare nu mai este necesară la inserarea curentă.
- echilibrul nodului rădăcină al arborelui rezultat este 0, ceea ce asigură că nici la o inserare ulterioară nu vor fi necesare reechilibrări.

Aceste observații ne pot permite să formulăm următoarea

Lemă 2: Inserarea cu rotație dublă dreapta nu mai necesită nici o altă transformare la inserarea curentă, și nici la cea următoare.

O altă observație importantă este că ambele rotații (simplă și dublă) se realizează în timp constant ($O(1)$). Importanța observației rezidă în aceea că inserarea (spre stânga) într-un AVL, cu menținerea echilibrului, se realizează deci în același timp ca într-un BST ($O(h)$); cum acum ne-am asigurat că

$h \leq 1gn$, înseamnă că inserarea stânga cu menținerea echilibrului se realizează în $O(1gn)$.

Cazurile de inserare spre dreapta sunt absolut simetrice (se interschimbă doar dreapta cu stânga și +1 cu -1).

Facem în continuare următoarea afirmație:

Teoremă: La inserarea într-un AVL, singurele cazuri de inserare care conduc la realizarea unui dezechilibru sunt cele denumite generic Left Left (care se rezolvă prin rotație simplă dreapta), LeftRight (care se rezolvă prin rotație dublă dreapta), RightRight (care se rezolvă prin rotație simplă stânga) și respectiv RightLeft (care se rezolvă prin rotație dublă stânga). Fiecare din aceste transformări se realizează în $O(1)$ și deci nu afectează performanța la inserare în BST.

Justificare:

Justificarea va fi intuitivă și mai degrabă informală, sugerând tehnica pe care o demonstrație riguroasă ar trebui s-o urmeze.

Demonstrația se poate realiza prin reducere la absurd: presupunem că există o altă structură a arborelui de intrare, în a cărei inserare s-ar produce dezechilibre, demonstrăm fie că un astfel de caz nu poate să apară, fie că acesta se reduce la unul din cazurile prezentate.

Presupunem următoarea situație: în cazul inserării LeftLeft echilibrul în **A** înainte de inserare nu este 0 (care pare cazul favorabil), ci este -1. Vom demonstra că acest caz NU poate apare:

Dacă la inserarea LeftLeft în α echilibrul în **A** înainte de inserare este -1, înseamnă că șablonul de descompunere pentru acest caz ($(\alpha) < \mathbf{A} < (\beta) < \mathbf{C} < (\gamma)$) poate fi aplicat cu un nivel mai jos (adică nodul **A** să fie rădăcină, iar α se descompune la rândul său după $(\alpha) < \mathbf{A} < (\beta)$). Această afirmație stă categoric în picioare, deoarece inserarea se produce pe o ramură a arborelui, iar reechilibrarea se produce pe exact aceeași ramură la revenirea din recursivitate (și deci discuția reechilibrării se pune mai întâi pentru subarbore – stânga în acest caz - și eventual apoi pentru arborele întreg). Dar dacă descompunem după șablon subarboarele cu rădăcina **A**, acest nod având echilibrul +1 (înainte de inserare), înseamnă că acest subarbore necesită el însuși o reechilibrare (prin rotație simplă sau dublă dreapta, după cum

inserarea s-a produs LeftLeft sau respectiv LeftRight). Dar, conform **Lemă 1** și **Lemă 2** inserarea cu rotație simplă respectiv dublă dreapta nu mai necesită nici o altă transformare la inserarea curentă. Cu alte cuvinte, datorită faptului că după inserare și rotație dreapta nodul **A** (care înainte de inserare avea echilibrul -1) are aceeași înălțime ca și înainte de inserare, nodul părinte (pentru care nodul **A** este subarbore stânga) nu "simte" inserarea. Pentru acest nod arborele pare neschimbat, deci nici el și nici un alt nod în sus pe ramură, până la rădăcină nici nu mai trebuie să facă verificări legate de echilibru! Am demonstrat deci că acest caz, dacă apare, el este rezolvat un nivel mai jos în arbore, printr-o rotație simplă sau dublă dreapta.

Analog, orice altă situație poate fi demonstrat că se reduce (cu un nivel sau două mai jos în arbore) la o altă situație din cele luate în discuție.

De exemplu, dacă în cazul LeftRight presupunem că echilibrul inițial al lui **B** este $+1$ și se produce o inserare în gama, atunci cazul se reduce la rotația simplă stânga două nivele mai jos; dacă echilibrul inițial în **A** este $+1$, cazul se reduce la rotația simplă stânga un nivel mai jos, pe când cazul cu echilibrul în **A** -1 nu conduce la dezechilibrare. La fel se pot analiza toate situațiile posibile, dovedind fie că nu conduc la dezechilibre, fie că ele se rezolvă la pași anteriori.

Putem acum să urmărim implementarea procedurală a inserării în arbori AVL, ținând cont de câteva particularități:

- conform demonstrației anterioare, în urma inserării unui nod într-un AVL, maxim o reechilibrare este necesară (rotație simplă/dublă dreapta/stânga) în cazul producerii unui dezechilibru. Dacă s-a realizat o reechilibrare înălțimea subarborelui (având nodul rădăcină nodul de referință al rotației) înainte de inserare este aceeași cu înălțimea arborelui corespunzător după inserare și echilibrare. Înseamnă că nodul părinte (și toate nodurile strămoș de-a lungul ramurii pe care s-a realizat inserarea, până la rădăcina întregului arbore) "nu sesizează" inserarea; înălțimea tuturor acestor arbori pare neschimbată în urma inserării, deci nu trebuie să facă nici o verificare (Notă: orice reechilibrare în arbore este necesară doar dacă se modifică înălțimea arborelui.). Această informație trebuie menținută ca o informație globală; faptul că s-a modificat înălțimea unui arbore privește toți subarborii săi, și toți arborii în a căror componență arborele implicat este un subarbore. O "variabilă globală" se implementează prin intermediul efectelor laterale. În cazul implementării curente,

predicatul `h(bool)` "memorează" această informație, și are următoarea semantică:

$$h(B) = \begin{cases} B = \text{true} & \text{dacă arborele a crescut în înălțime} \\ B = \text{false} & \text{dacă arborele și-a păstrat înălțimea,} \end{cases}$$

- inserarea în AVL se realizează ca în BST, prin intermediul predicatului `insertInAVL/3`, urmată dacă este cazul de o singură reechilibrare,
- reprezentarea arborelui este de forma: `t(nod, echilibru, stânga, dreapta)`,
- reechilibrările posibile sunt:
 - rotație simplă dreapta implementată de `restr1Right/3`
 - rotație dublă dreapta implementată de `restr2Right/3`
 - rotație simplă stânga implementată de `restr1Left/3`
 - rotație dublă stânga implementată de `restr2Left/3`

Prezentăm pe scurt fiecare din predicatele care inervin în implementare:

```
%insertInAVL/3
%insertInAVL(nod de inserat,
             arbore AVL de intrare,
             arbore AVL rezultat in urma inserarii)
```

Inserarea se poate face într-un arbore vid (clauza 1), caz în care înălțimea sa crește deci se asertează `h(true)`, sau nevid (clauzele 2-4). Dacă arborele este nevid nodul de inserat poate coincide cu rădăcina (deci poate fi deja în arbore - clauza 2), caz în care arborele nu se modifică deci se asertează `h(false)`, sau poate să difere, caz în care se va insera într-unul din subarborii stânga/dreapta. Această inserare determină un proces de restructurare (reechilibrare) conform celor prezentate mai sus. Oricum, inserarea se face printr-un apel recursiv al predicatului `insertInAVL` pe subarboarele stânga/dreapta, deci subarboarele respectiv va rămâne un arbore AVL.

```
insertInAVL(Key, nil, t(Key, 0, nil, nil)) :- !,
    retract(h(_)),
    asserta(h(true)).
insertInAVL(Key, t(Key, E, Left, Right), t(Key, E, Left, Right)) :- !,
    retract(h(_)),
```

```

    asserta(h(false)).
insertInAVL(InsKey,t(Key,E,Left,Right),t(NKey,NE,NLeft,NRight)):-
    InsKey<Key,!,
    insertInAVL(InsKey,Left,TLeft),
    retract(h(H)),!,
restrLeft(H,t(Key,E,Left,Right),t(NKey,NE,NLeft,NRight),TLeft).
insertInAVL(InsKey,t(Key,E,Left,Right),
            t(NKey,NE,NLeft,NRight)):-
    insertInAVL(InsKey,Right,TRight),
    retract(h(H)),!,
restrRight(H,t(Key,E,Left,Right),t(NKey,NE,NLeft,NRight),TRight).

```

Procesul de restructurare este prezentat pentru cazurile de inserare în subarborele stânga/dreapta. Restructurarea este efectuată de către predicatele:

```

%restrLeft/4
%restrLeft(daca_arborele_a_crescut_?,
    arbore_AVL=A,
    integ_arborele_AVL_dupa_inserare_si_restructurare,
    rezultatul_inserarii_in_subarborele_stanga_al_A)

```

și simetricul său `restrLeft /4` în care există un singur parametru de ieșire (parametrul al treilea din lista parametrilor).

Prezentăm acum clasificarea problemei pe cazuri specificate prin clauze Horn pe baza parametrilor de intrare. Dacă subarborele stânga nu a crescut (clauza 1) se asertează "h(false)". Altfel, dacă arborele a crescut fără însă să se altereze echilibrul cerut în definiția arborelui AVL (clauzele 2 și 3) se marchează păstrarea ("h(false)") sau creșterea ("h(true)") înălțimii arborelui. Altfel, creșterea subarborelui stâng violează condiția din definiția arborelui AVL și se va efectua restructurarea LL sau LR de către predicatele `rls`, respectiv `r2s`.

```

restrLeft(false,t(Key,E,Left,Right),t(Key,E,NLeft,Right),NLeft):-
    !,asserta(h(false)).
restrLeft(true,t(Key,1,Left,Right),t(Key,0,NLeft,Right),NLeft):-
    !,asserta(h(false)).
restrLeft(true,t(Key,0,Left,Right),
    t(Key,-1,NLeft,Right),NLeft):-!,
    asserta(h(true)).
restrLeft(true,t(Key,-1,Left,Right),
    t(NKey,NE,NLeft,NRight),TLeft):-
    TLeft=t(_, -1, _, _),!,
    restrlLeft(t(Key,-1,Left,Right),
    t(NKey,NE,NLeft,NRight),TLeft),
    asserta(h(false)).

```

```

restrLeft(true,t(Key,-1,Left,Right),
           t(NKey,NE,NLeft,NRight),TLeft):-
    restr2Left(t(Key,-1,Left,Right),
              t(NKey,NE,NLeft,NRight),TLeft),
    asserta(h(false)).

```

Predicatele `restr1Left/3` și `restr2Left/3` stabilesc următoarele relații între parametri (parametrul al doilea este de ieșire în fiecare caz):

```

%restr1Left/3
%restr1Left(arbore_AVL=A,
            intreg_arborele_AVL_dupa_inserare_si_restructurare_LL,
            rezultatul_inserarii_in_subarborele_stanga_al_A)

%restr2Left/3
%restr2Left(arbore_AVL=A,
            intreg_arborele_AVL_dupa_inserare_si_restructurare_LR,
            rezultatul_inserarii_in_subarborele_stanga_al_A)

```

Predicatul `restr1Left/3` este definit printr-o singură clauză.

```

restr1Left(t(B,-1,_,R),t(A,0,LL,t(B,0,LR,R)),t(A,-
1,LL,LR)).

```

și corespunde rotației simple dreapta, în timp ce pentru `restr2Left/3` (corespunzând rotației duble dreapta) se disting trei cazuri (numele de restructurare stânga de la faptul că inserarea este pe o ramură stânga). Două din ele se datorează inserării (cu creșterea înălțimii) pe stânga sau pe dreapta în ramura din dreapta a subarborelui stânga. Aceste cazuri sunt specificate în clauzele 1 și 2. Mai există însă cazul când arborele `LeftRight` era vid (adică nodul **B** este **nil**). În acest caz prin inserare el va crește în înălțime însă factorul său de echilibru va fi 0.

```

restr2Left(t(C,-1,_,R),t(B,0,t(A,0,LL,LRL),
                             t(C,1,LRR,R)),t(A,1,LL,t(B,-1,LRL,LRR))).
restr2Left(t(C,-1,_,R),t(B,0,t(A,-1,LL,LRL),
                             t(C,0,LRR,R)),t(A,1,LL,t(B,1,LRL,LRR))).
restr2Left(t(C,-1,_,R),t(B,0,t(A,0,LL,LRL),
                             t(C,0,LRR,R)),t(A,1,LL,t(B,0,LRL,LRR))).

```

Clauzele predicatelor `restrRight/4`, `restrlRight/3` și `restr2Right/3` exprimă același raționament pentru restructurările `RightRight` și respective `RightLeft`.

```
%restrRight/4
%restrLeft(daca_arborele_a_crescut_?,
           arbore_AVL=A,
           intreg_arborele_AVL_dupa_inserare_si_restructurare,
           rezultatul_inserarii_in_subarborele_dreapta_al_A)

restrRight(false,t(Key,E,Left,Right),
            t(Key,E,Left,NRight),NRight):-!,
            asserta(h(false)).
restrRight(true,t(Key,-1,Left,Right),
            t(Key,0,Left,NRight),NRight):-!,
            asserta(h(false)).
restrRight(true,t(Key,0,Left,Right),
            t(Key,1,Left,NRight),NRight):-!,
            asserta(h(true)).
restrRight(true,t(Key,1,Left,Right),
            t(NKey,NE,NLeft,NRight),TRight):-
            TRight=t(_,1,_,_),!,
            restrlRight(t(Key,1,Left,Right),
                        t(NKey,NE,NLeft,NRight),TRight),
            asserta(h(false)).
restrRight(true,t(Key,1,Left,Right),
            t(NKey,NE,NLeft,NRight),TRight):-
            restr2Right(t(Key,1,Left,Right),
                        t(NKey,NE,NLeft,NRight),TRight),
            asserta(h(false)).

%restrlRight/3
%restrlRight(arbore_AVL=A,
             intreg_arborele_AVL_dupa_inserare_si_restructurare_LL,
             rezultatul_inserarii_in_subarborele_dreapta_al_A)

restrlRight(t(A,1,L,_),
            t(B,0,t(A,0,L,RL),RR),t(B,1,RL,RR)).

%restr2Right/3
%restr2Right(arbore_AVL=A,
             intreg_arborele_AVL_dupa_inserare_si_restructurare_LR,
             rezultatul_inserarii_in_subarborele_dreapta_al_A)

restr2Right(t(A,1,L,_),t(B,0,t(A,-1,L,RL),
```

```

t(C,0,RLR,RR)), t(C,-1,t(B,1,RLL,RLR),RR)).
restr2Right(t(A,1,L,_),t(B,0,t(A,0,L,RLL),
t(C,1,RLR,RR)), t(C,-1,t(B,-1,RLL,RLR),RR)).
restr2Right(t(A,1,L,_),t(B,0,t(A,0,L,RLL),
t(C,0,RLR,RR)),t(C,-1,t(B,0,RLL,RLR),RR)).

```

Pentru vizualizarea rezultatelor se folosește un predicat de tipărire:

```

writeAVL(Tree):-
    writeAVL(0,Tree).

writeAVL(_,nil):-!.
writeAVL(Level,t(Key,E,Left,Right)):-
    LevelPlus1 is Level + 1,
    writeAVL(LevelPlus1,Right),
    nl,writeSpaces(Level),
    write('['),write(Key),
    write(', '),write(E),write(']'),
    writeAVL(LevelPlus1,Left).

writeSpaces(0):-!.
writeSpaces(N):-
    write(' '),
    NMinus1 is N - 1,
    writeSpaces(NMinus1).

```

Și un exemplu de rulare:

```

asserta(h(false)),
nl,write('Insert                                     4
:'),insertInAVL(4,nil,T1),writeAVL(T1),
nl,write('Insert                                     5
:'),insertInAVL(5,T1,T2),writeAVL(T2),
nl,write('Insert                                     7
:'),insertInAVL(7,T2,T3),writeAVL(T3),
nl,write('Insert                                     2
:'),insertInAVL(2,T3,T4),writeAVL(T4),
nl,write('Insert                                     1
:'),insertInAVL(1,T4,T5),writeAVL(T5),
nl,write('Insert                                     3
:'),insertInAVL(3,T5,T6),writeAVL(T6),
nl,write('Insert                                     6
:'),insertInAVL(6,T6,T7),writeAVL(T7).

```

```

Insert 4 :
[4,0]
Insert 5 :

```



```

    [5,0]
  [4,1]
Insert 7 :
    [7,0]
  [5,0]
    [4,0]
Insert 2 :
    [7,0]
  [5,-1]
    [4,-1]
      [2,0]
Insert 1 :
    [7,0]
  [5,-1]
    [4,0]
      [2,0]
        [1,0]
Insert 3 :
    [7,0]
  [5,1]
  [4,0]
    [3,0]
      [2,0]
        [1,0]
Insert 6 :
    [7,0]
  [6,0]
    [5,0]
  [4,0]
    [3,0]
      [2,0]
        [1,0]
T1 = t(4,0,nil,nil),
T2 = t(4,1,nil,t(5,0,nil,nil)),
T3 = t(5,0,t(4,0,nil,nil),t(7,0,nil,nil)),
T4 = t(5,-1,t(4,-1,t(2,0,nil,nil),nil),t(7,0,nil,nil)),
T5 = t(5,-
1,t(2,0,t(1,0,nil,nil),t(4,0,nil,nil)),t(7,0,nil,nil)),
T6 =
t(4,0,t(2,0,t(1,0,nil,nil),t(3,0,nil,nil)),t(5,1,nil,t(
7,0,nil,nil))),
T7 =
t(4,0,t(2,0,t(1,0,nil,nil),t(3,0,nil,nil)),t(6,0,t(5,0,
nil,nil),t(7,0,nil,nil))) ?
yes

```

Pentru arborii Fibonacci (cei mai dezechilibrați arbori AVL)

```
buildFibonacciAVL(N,Tree):-
    asserta(fibKey(1)),
    fibonacciAVL(N,Tree),
    retractall(fibKey(_)).

fibonacciAVL(0,nil).
fibonacciAVL(1,t(Key,-1,nil,nil)):-
    retract(fibKey(Key)),
    KeyPlus1 is Key + 1,
    asserta(fibKey(KeyPlus1)).
fibonacciAVL(N,t(Key,-1,Left,Right)):-
    NMinus1 is N - 1,
    NMinus2 is N - 2,
    fibonacciAVL(NMinus1,Left),
    retract(fibKey(Key)),
    KeyPlus1 is Key + 1,
    asserta(fibKey(KeyPlus1)),
    fibonacciAVL(NMinus2,Right).
```

Exemplu:

```
| ?- buildFibonacciAVL(5,T),writeAVL(T).

      [12,-1]
    [11,-1]
      [10,-1]
        [9,-1]
[8,-1]
      [7,-1]
        [6,-1]
    [5,-1]
      [4,-1]
        [3,-1]
          [2,-1]
            [1,-1]
T = t(8,-1,t(5,-1,t(3,-1,t(2,-1,t(1,-
1,nil,nil),nil),t(4,-1,nil,nil)),t(7,-1,t(6,-
1,nil,nil),nil)),t(11,-1,t(10,-1,t(9,-
1,nil,nil),nil),t(12,-1,nil,nil))) ?
Yes
```


Bibliografie

1. K. R. Apt, David S. Warren et al editors, *The Logic Programming Paradigm, a 25-Year Perspective*, Springer, 1998.
2. W.F. Clocksin, C.S. Mellish, *Programming in Prolog*, fourth edition, Springer-Verlag Telos, September 1994.
3. T. Conlon, *Programming in Parlog*, Addison-Wesley, March 1989.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Second Edition, MIT Press and McGraw-Hill, September, 2001.
5. S. Gregory, *Parallel Logic Programming in PARLOG The Language and its Implementation*, Addison-Wesley, 1987.
6. R. Kowalski, *Logic for Problem Solving*, North Holland, 1979.
7. J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.
8. K. Marriot, P. J. Stuckey, *Programming with Constraints An Introduction*, The MIT Press, 1998.
9. T.I. Mureșan, R. Potolea, E. Todoran, A. Suciu, *Programare Logică, Îndrumător de Laborator*, , Universitatea Tehnică din Cluj-Napoca 1997.
10. R. Potolea, *Contribuții la sistemele de programare logică multiparadigmă*, Universitatea Tehnică din Cluj-Napoca, Teză de doctorat, 1998.
11. R. Potolea, *Programare Logică: Fundamente Paradigme Extensii*, Editura Printeck, 2000.
12. V. Saraswat, P. Van Henrenryck, *Principles and Practice of Constraint Programming*, The MIT Press, 1995.
13. E. Shapiro, editor. *Concurrent Prolog*, vol 1 and 2. MIT Press, Dec. 1987.
14. SICStus Prolog, <http://www.sics.se/isl/sicstuswww/site/index.html> Manualul Sictus Prolog.
15. L. Sterling, E. Shapiro , *The Art of Prolog*, The MIT Press; 2 Sub edition March, 1994.