

Cuprins

PREFAȚĂ.....	9
CUVÂNT ÎNAINTE.....	11

PARTEA I. PROGRAMAREA BAZATĂ PE REGULI ȘI SISTEMELE EXPERT

CAPITOLUL 1. PARADIGME DE PROGRAMARE..... 17

1.1. PROGRAMAREA IMPERATIVĂ – REZOLVĂ DICTÂND CUM SĂ FACI	18
1.2. PROGRAMAREA LOGICĂ – REZOLVĂ SPUNÂND CE VREI SĂ FACI	19
1.3. PROGRAMAREA FUNCȚIONALĂ – REZOLVĂ APELÂND O FUNCȚIE	22
1.4. PROGRAMAREA ORIENTATĂ-OBIECT – REZOLVĂ CONSTRUIND OBIECTE CE INTERACȚIONEAZĂ	24
1.5. PROGRAMAREA BAZATĂ PE REGULI – REZOLVĂ CA ÎNTR-UN JOC DE PUZZLE SAU LEGO	26

CAPITOLUL 2. INTRODUCERE ÎN SISTEMELE EXPERT 31

2.1. CE SUNT SISTEMELE EXPERT ?	32
2.2. PARTICULARITĂȚI ALE DOMENIULUI INTELIGENȚEI ARTIFICIALE	34
2.3. PRIN CE DIFERĂ UN SISTEM EXPERT DE UN PROGRAM CLASIC?	35
2.4. EXEMPLE DE SISTEME EXPERT	37
2.5. EVALUAREA OPORTUNITĂȚII SISTEMELOR EXPERT	40

PARTEA A II-A. ORGANIZAREA ȘI FUNCȚIONAREA SISTEMELOR EXPERT

CAPITOLUL 3. ORGANIZAREA UNUI SISTEM EXPERT 45

3.1. BAZA DE CUNOȘTINȚE: FAPTELE.....	46
3.2. REGULILE.....	48
3.3. VARIABLE ȘI ȘABLOANE ÎN REGULI	50
3.4. LEGĂRI DE VARIABLE ȘI INSTANȚE DE REGULI	52
3.5. AGENDA	53
3.6. MOTORUL DE INFERENȚE.....	54

CAPITOLUL 4. ÎNLĂNȚUIREA REGULILOR ÎN MOTOARELE DE INFERENȚĂ 57

4.1. CĂUTAREA SOLUȚIEI ÎN PROBLEMELE DE INTELIGENȚĂ ARTIFICIALĂ	57
4.2. ÎNLĂNȚUIREA ÎNAINTE	59
4.3. DESPRE PREZUMȚIA DE LUME DESCHISĂ/ÎNCHISĂ	62
4.4. DESPRE MONOTONIE.....	64
4.5. DESPRE FAPTE NEGATE ȘI REGULI CU ȘABLOANE NEGATE	66

CAPITOLUL 5. REGIMUL DE LUCRU TENTATIV 67

5.1. SIMULAREA UNUI MOTOR TENTATIV PRINTR-UN <i>SHELL</i> DE MOTOR IREVOCABIL ..	69
CAPITOLUL 6. CONFRUNTAREA RAPIDĂ DE ȘABLOANE: ALGORITMUL REȚE	81
6.1. IMPORTANȚA ORDINII ȘABLOANELOR.....	92
 PARTEA A III-A. ELEMENTE DE PROGRAMARE BAZATĂ PE REGULI	
CAPITOLUL 7. PRIMII PAȘI ÎNTR-UN LIMBAJ BAZAT PE REGULI: CLIPS... ..	95
7.1. SĂ FACEM O ADUNARE.....	96
7.2. CUM REALIZĂM O ITERAȚIE?	102
CAPITOLUL 8. CONSTRÂNGERI ÎN CONFRUNTAREA ȘABLOANELOR.....	111
8.1. INTEROGĂRI ASUPRA UNEI BAZE DE DATE	111
8.2. UN EXEMPLU DE SORTARE	117
CAPITOLUL 9. DESPRE CONTROLUL EXECUȚIEI	123
9.1. CRITERII UTILIZATE ÎN ORDONAREA AGENDEI.....	123
9.2. URMĂRIREA EXECUȚIEI	125
9.3. STRATEGII DE REZOLUȚIE A CONFLICTELOR.....	129
9.4. IMPORTANȚA ORDINII ASERTĂRIILOR	130
9.5. EFICIENTIZAREA EXECUȚIEI PRIN SCHIMBAREA ORDINII COMENZILOR <i>RETRACT</i> ȘI <i>ASSERT</i>	132
CAPITOLUL 10. RECURSIVITATEA ÎN LIMBAJELE BAZATE PE REGULI... ..	135
10.1. TURNURILE DIN HANOI.....	135
10.2. CALCULUL FACTORIALULUI	141
 PARTEA A IV-A DEZVOLTAREA DE APLICAȚII	
CAPITOLUL 11. OPERAȚII PE LISTE, STIVE ȘI COZI	149
11.1. INVERSAREA UNEI LISTE	149
11.2. MAȘINI-STIVĂ ȘI MAȘINI-COADĂ	150
11.3. UN PROCES CARE LUCREAZĂ CU STIVA	153
11.4. UN PROCES CARE LUCREAZĂ SIMULTAN CU O STIVĂ ȘI O COADĂ	155
11.5. EVALUAREA EXPRESIILOR	158
CAPITOLUL 12. SISTEME EXPERT ÎN CONDIȚII DE TIMP REAL	165
12.1. SERVIREA CLIEȚILOR LA O COADĂ	166
12.2. EVENIMENTE EXTERNE PSEUDO-ALEATORII	170
CAPITOLUL 13. CONFRUNTĂRI DE ȘABLOANE ÎN PLAN	175
13.1. JOCUL 8-PUZZLE	178
13.2. JOCUL CU VAPORAȘE	180

CAPITOLUL 14. O PROBLEMĂ DE CĂUTARE ÎN SPAȚIUL STĂRILOR	187
14.1. MAIMUȚA ȘI BANANA – O PRIMĂ TENTATIVĂ DE REZOLVARE	187
14.2. <i>HILL-CLIMBING</i>	190
14.3. O MAIMUȚĂ EZITANTĂ: METODA TENTATIVĂ EXHAUSTIVĂ.....	197
14.4. O MAIMUȚĂ DECISĂ: METODA <i>BEST-FIRST</i>	199
CAPITOLUL 15. CALCULUL CIRCUITELOR DE CURENT ALTERNATIV	201
CAPITOLUL 16. REZOLVAREA PROBLEMELOR DE GEOMETRIE	207
16.1. REPREZENTAREA OBIECTELOR GEOMETRICE ȘI A RELAȚIILOR DINTRE ELE	208
16.2. LIMBAJUL DE DEFINIRE A PROBLEMEI	209
16.3. PROPAGAREA INFERENȚELOR	211
16.4. LUNGIMEA RULĂRII ȘI A DEMONSTRAȚIEI	218
CAPITOLUL 17. SFATURI DE PROGRAMARE BAZATĂ PE REGULI	221
17.1. RECOMANDĂRI DE STIL ÎN PROGRAMARE	221
17.2. ERORI ÎN EXECUȚIA PROGRAMELOR CLIPS	224
17.3. AȘTEPTĂRI NEÎNDEPLINITE	226
BIBLIOGRAFIE	229

Prefață

Inteligența artificială s-a impus ca una dintre cele mai dinamice ramuri ale tehnologiei informației prin realizările remarcabile atât pe plan teoretic cât și prin diversitatea aplicațiilor sale. Între acestea, sistemele expert ocupă un rol important în informatizarea unei mari diversități de domenii ale activității social-economice.

Realizarea unui sistem expert este, în primul rând, o activitate de echipă care reunește specialiști din domenii diverse alături de informaticieni. Pentru ca sistemul construit să poată dialoga cu viitorii utilizatori este necesar ca realizatorii să poată comunica între ei, să aibă un limbaj comun. Cred că unul din principalele merite ale lucrării de față este acela de a oferi un model prin care se poate realiza o astfel de comunicare. Construcțiile teoretice care fundamentează paradigmele programării bazate pe reguli și ale celei obiectuale sunt deduse în mod natural pornind de la exemple simple, sugestive.

Autorul realizează o remarcabilă prezentare a caracteristicilor sistemelor expert – fapte, reguli, motoare de inferență –, cu o mențiune specială asupra regimului tentativ de funcționare. În construcția unui sistem expert sunt necesare instrumente specializate. Există numeroase limbaje pe structura cărora se pot realiza sisteme expert. Din păcate, de multe ori prezentarea acestora le transformă în cadre rigide în care trebuie să fie cuprinse faptele din realitate – adesea cu eliminări ale unor trăsături esențiale. Abordarea insinuant obiectuală a limbajului CLIPS permite autorului ca, pornind de la fapte, să regăsească natural acele structuri ale limbajului care concură la surprinderea realității în esențialitatea ei în raport cu problema ce urmează a fi rezolvată.

Exemplele din ultima parte a cărții sunt de natură să pună în evidență diversitatea aplicațiilor în care instrumentele inteligenței artificiale oferă soluții elegante și eficiente, cu condiția alegerii judicioase a strategiei utilizate, cât și limitele, pentru moment, în abordarea unor probleme simple – în aparență – dar de mare complexitate – în esență –, cum ar fi cele ale demonstrării automate a teoremelor din geometrie.

Rod al unei prodigioase activități de cercetare dublate de dăruirea exemplară în munca la catedră, lucrarea profesorului Dan Cristea va reuși să devină o carte de referință pentru toți cei implicați în construcția și utilizarea sistemelor expert.

Călin Ignat

Cuvânt înainte

O carte dedicată programării bazate pe reguli ar putea stârni, în primul rând, curiozitatea informaticienilor profesioniști ori a studenților la informatică sau calculatoare. Pentru un informatician, întâlnirea cu un alt mod de a concepe actul programării decât cel pe care îl utilizează zilnic poate fi incitant. Dezvoltatorii profesioniști de programe sunt deprinși să-și treacă în cv-urile lor o listă de limbaje pe care le cunosc, de multe ori acestea acoperind mai multe paradigme. Ei învață mai multe limbaje, diferite ca modalități de abordare a actului programării, nu neapărat din necesitatea impusă de un angajator de a programa în acele limbaje, cât pentru a dobândi nivelul lui “aha, asta am mai întâlnit”, care înseamnă flexibilitate, înseamnă asocieri și soluții bazate pe experiență. Această experiență, dată de practică sau de lecturi, duce, în esență, la adoptarea celei mai nimerite atitudini în fața unei noi probleme.

Există, fără îndoială, și o anumită categorie de “meseriași”, care, prin natura activității lor, sunt puși în fața problemelor din sfera inteligenței artificiale sau a unui domeniu conexe acesteia. Lor, cred eu, cartea le poate fi de folos.

Cred apoi că această carte are ceva de spus profesorilor ce predau informatica în școală. Este foarte important ca informatica să fie predată, și nu numai în școlile cu programe speciale de informatică, într-o manieră atractivă pentru elevi. E atât de ușor să “molipsești” de informatică un elev isteț, dar e la fel de ușor să-l dezamăgești încât acesta să fugă toată viața lui de acest domeniu. Dacă programarea se predă “la tablă”, începând cu lecții de sintaxă rigidă a unui limbaj de programare, pentru că așa obligă programa, și continuă cu dezvoltarea de linii de cod, dacă expunerea e atât de strâns legată de un limbaj anume încât a ști să programezi se reduce la a scrie programe în acel unic Limbaj De Programare, dacă se pierde din vedere faptul esențial că activitatea de programare este în primul rând una de creație și calculatorul este un penson cu care poți realiza orice tablou iar nu un gherghet pe care trebuie să reproduci un desen impus, atunci uriașul potențial de creație care este imaginația atât de debordantă a copiilor noștri va fi închisată oficial în tipare iar notele mari și diplomele vor atesta umila docilitate și perversa abilitate de a reproduce, iar nu neastâmpărul, căutarea și arta. Cum cred că programarea prin reguli este o activitate pe care aș numi-o “confortabilă intelectual”, iar limbajul utilizat ca suport al argumentațiilor este atât de ușor de învățat, cartea ar putea constitui un ajutor pentru profesorii de informatică din școli în tentativa acestora de a stimula spre creație informatică elevii talentați.

Volumul pe care îl aveți acum în mână nu este un manual al unui limbaj de programare, deși limbajul CLIPS este utilizat aproape peste tot pentru a

exemplifica noțiunile tratate (cu excepția capitolului 5 în care notația este una generică, de pseudo-cod). CLIPS nu este predat în carte în sensul în care ne-au obișnuit cărțile de “Programare în Limbajul XYZ”. Dovadă: faptul că multe elemente ale limbajului, cum ar fi declarațiile de funcții, lista funcțiilor de bibliotecă, sau elementele de programare orientată-obiect pe care le încorporează, nu sunt tratate de loc ori doar episodic. Un manual de CLIPS poate fi ușor procurat din biblioteci sau de pe Internet. Mai greu de învățat decât un limbaj de programare este însă deprinderea unui stil, ajungerea la acea maturitate a actului de programare care să permită găsirea metodelor celor mai adecvate rezolvării unor probleme, dobândirea eleganței soluțiilor, a productivității muncii de programare și a eficienței codurilor. Am fost cu precădere interesat de aceste aspecte. Variantele complete ale programelor CLIPS prezentate în cuprinsul cărții pot fi accesate de cititor la adresa de Internet <http://www.infoiasi.ro/~dcristea/carti/PBR/>.

Pentru elaborarea cărții am folosit materiale utilizate de-a lungul anilor în cadrul a trei cursuri la Facultatea de Informatică a Universității “Alexandru Ioan Cuza” din Iași: cursul de bază de inteligență artificială, dedicat studenților anului III, cursul opțional de sisteme expert pentru studenții anului IV și cursul de inteligență artificială și CLIPS predat studenților formațiilor de studii post-universitare. În felul acesta, an de an s-au adăugat noi probleme sau au fost găsite noi soluții la probleme vechi. Nu m-am sfiit să folosesc în carte și idei sau fragmente de cod sugerate de foști studenți ai mei, pe care îi amintesc în lucrare și cărora le mulțumesc pe această cale.

În fazele inițiale ale introducerii unui neologism în limbă este întotdeauna dificil de apreciat dacă termenul care își face acum loc prin împrumut va fi până la urmă acceptat ori nu. În limbajul tehnic ori științific, dificultatea este și mai mare datorită abundenței de cuvinte străine și vitezei cu care acestea apar. În informatică însă, acest fenomen este exacerbat prin invazia de termeni de îngustă specialitate, limba din care se fac împrumuturi fiind, în exclusivitate, engleza. Apoi e deja notorie apetența informaticienilor spre un limbaj amestecat, uneori voit colorat cu englezisme, datorat comodității de a utiliza termeni străini în locul echivalenților autohtoni și a-i considera ca fiind ai noștri dintotdeauna. Deciziile dificile sunt aici, probabil ca și în alte domenii, nu atât în privința termenilor ce nu-și găsesc nicicum un echivalent în românește și care, fără discuție, trebuie preluați ca atare pentru a ne putea înțelege între noi (în astfel de cazuri am notat cuvintele englezești în italice și le-am adăugat terminațiile românești despărțite prin cratimă, de exemplu, *shell*, *shell*-uri), ci mai ales în privința acelor pentru care dicționarele indică cel puțin un corespondent, dar sensurile indicate sunt ușor diferite, deci imperfecte, pentru noua utilizare. Un termen din această categorie, pentru care am optat să utilizez o traducere românească, este cel de *pattern*, cu pleiada lui de compuși ori sintagme derivate (*pattern matching*, *pattern recognition*). Am preferat să utilizez o traducere a sa, poate în dezacord cu alți colegi ai mei ce trudesesc în același câmp, prin românescul *șablon* (și nu *tipar*, ce îmi sugerează prea mult uzanța lui din

croitorie). *Șablon* mi s-a părut că are, sau poate primi ușor, încărcarea semantică din programare, care e legată de două operații: una în care selectează obiecte asemenea lui și cealaltă în care produce obiecte, ori componente ale obiectelor, de un anumit tip. În privința lui *pattern matching*, am considerat întotdeauna că sintagma are două conotații ce trebuie traduse diferit în românește: una este dinamică, semnificând o operație de triere a unor obiecte ce corespund șablonului, deci o *confruntare de șabloane*, cealaltă este statică și corespunde rezultatului confruntării, în esență boolean, da ori nu, adică s-a verificat ori nu dacă *șablonul s-a potrivit peste obiect*.¹

Cartea este structurată în patru părți. **Partea I – Programarea bazată pe reguli și sistemele expert** realizează în primul rând o introducere în programarea bazată pe reguli, prezentând specificul acestui tip de programare. Cititorul este purtat prin marea familie a paradigmelor de programare prin rezolvarea unei probleme în maniera “de casă” a fiecăreia dintre ele. Se descriu apoi sistemele expert, “copiii minune” ai paradigmei, și se inventariază tipurile lor și realizările din acest domeniu.

Partea a II-a se intitulează **Organizarea și funcționarea sistemelor expert** și prezintă detalii constructive ale motoarelor de sisteme expert. Se descrie organizarea generală a oricărui sistem expert, cum arată un ciclu din funcționarea unui astfel de motor, cunoscut și sub numele de motor de inferențe, ce sunt faptele și regulile, cum se leagă variabilele la valori în confruntarea regulilor asupra faptelor și ce este agenda. Se prezintă apoi maniera de căutare a soluției cea mai uzuală în sistemele expert: dinspre fapte inițiale spre concluzii. Se descriu cele două regimuri de funcționare a motoarelor de inferență: irevocabil (decis: un pas făcut, chiar dacă pe o cale greșită, rămâne bun făcut) și tentativ (ezitant: ai ajuns într-un punct mort, nu-i nimic, ia-o pe o altă cale), cât și condiții suficiente de găsire a soluției pentru motoarele ce funcționează într-o manieră irevocabilă. Pentru că cele mai multe motoare de inferență ale sistemelor expert implementează un comportament irevocabil, iar acesta nu garantează soluția, se arată cum poate fi modelat un comportament tentativ, care duce întotdeauna la găsirea unei soluții, atunci când ea există, pe o arhitectură irevocabilă. Se prezintă apoi un algoritm celebru de confruntare a unei mulțimi de șabloane peste o mulțime de fapte - RETE. Acest algoritm stă la baza realizării celor mai multe *shell*-uri de sisteme expert.

Partea a III-a, sub titlul **Elemente de programare bazată pe reguli**, prezintă fundamentele programării prin reguli. Pe parcursul unui capitol se introduc sintaxa și elementele esențiale ale unui limbaj bazat pe reguli – CLIPS. Toate exemplele din carte vor fi apoi construite în acest limbaj. Mici aplicații, ca, de exemplu, una inspirată din activitatea profesorilor de liceu, vor fi tot atâtea pretexte

¹ Despre *pattern recognition*, un termen care nu e utilizat în carte, se pot spune, de asemenea, multe lucruri, traducerea prin *recunoașterea trăsăturilor* părându-mi-se mai puțin supusă confuziei decât *recunoașterea caracterelor*.

pentru construirea și comentarea unor soluții. Se insistă asupra agendei, structura care păstrează activările, și se arată ce strategii de rezoluție a conflictelor pot fi utilizate și maniera în care programatorul poate beneficia de schimbarea strategiei. Exemplele date intenționează să evidențieze situații care necesită impunerea de priorități regulilor. Se comentează importanța ordinii comenzilor care produc modificări în baza de fapte. Pentru că există motive care fac ca recursivitatea să nu fie la ea acasă într-un limbaj bazat pe reguli, se arată ce soluții se pot găsi care să simuleze un algoritm recursiv.

Ultima **parte, a IV-a** este dedicată **Dezvoltării de aplicații**. Fiecare capitol prezintă o altă problemă, se propun soluții și se comentează. Astfel se arată cum pot fi construite mașini specializate pentru structurile de stivă și coadă și cum pot fi acestea integrate în aplicații. Un alt capitol prezintă elemente de proiectare a aplicațiilor în care pot apărea evenimente externe și în care variabila timp este la mare preț. Sub pretextul unor aplicații ce necesită recunoașterea unor obiecte planare sau capacitatea de a naviga în plan, se descrie maniera în care șabloanele părților stângi ale regulilor sunt făcute să “semene” cu obiectele pe care dorim să le descoperim. Se prezintă apoi o problemă cunoscută din inteligență artificială care, deși aparent banală, se relevă a avea o soluție ce depășește în complexitate tot ceea ce s-a prezentat anterior. O problemă de fizică de liceu oferă pretextul prezentării propagării fluxului de calcul în maniera “ghidată de date” (*data-driven*) și în care nedeterminismul intrinsec paradigmei este exploatat în privința ordinii efectuării operațiilor. Se propune apoi un demonstrator de teoreme aplicat în rezolvarea automată a problemelor de geometrie. Soluția adoptată este una de explozie combinatorială a faptelor ce pot fi generate din ipoteze prin aplicarea adevărilor cunoscute (teoreme). În sfârșit, în ultimul capitol al cărții sunt puse în evidență, comentate și corectate câteva erori întâlnite în practica limbajului CLIPS.

Partea I

Programarea bazată pe reguli și sistemele expert

Paradigme de programare

Introducere în sistemele expert

Capitolul 1

Paradigme de programare

Programarea este o activitate mentală pe care oamenii o fac de foarte multă vreme. În general, această activitate este făcută cu scopul de a gândi o dată și a aplica rezultatul acestui efort ori de câte ori este nevoie apoi. În loc să cânte efectiv la pian, oamenii au inventat un tambur sau o bandă, cu găuri sau cu ace, care, învârtindu-se, permite unui mecanism să producă sunete în ritmul în care reperele de pe tambur ori bandă acționează asupra unor senzori. O flașnetă nu este decât un calculator primitiv, capabil să reproducă un program înregistrat pe un bandă. Dacă se schimbă banda, se obține o altă melodie. Meșterul ce a produs pentru prima dată o bandă cu găuri pentru flașnetă în scopul reproducerii unei melodii a fost un programator. Într-un anumit sens, o carte de bucate este o colecție de programe. O rețetă ne învață ce ingrediente trebuie să folosim și ce operații trebuie să facem asupra lor ori de câte ori ni se face dor de un fel de mâncare sau de o prăjitură.

Apariția calculatoarelor a transformat activitatea de programare, episodică până atunci, într-o știință – informatica. De când oamenii au început să se aplece asupra calculatoarelor din necesitatea de a rezolva probleme mai repede și mai bine, din pasiunea iscată de curiozitate, ori pentru satisfacerea necesității de a-și folosi imaginația, programarea a evoluat în două direcții. Pe de o parte s-a produs o perfecționare a limbajelor, în așa fel încât actul programării s-a depărtat tot mai mult de electronica rigidă a mașinii, apropiindu-se în schimb de standardele canalelor de comunicație umană (exprimarea prin imagini sau limbaj apropiat de cel natural) și, pe de altă parte, însăși maniera de a programa s-a diversificat, în așa fel încât rezolvarea unei probleme se poate acum gândi în multiple feluri. Prima direcție în dezvoltarea programării s-a manifestat pe linia măririi productivității actului de programare, expresivitatea instrucțiunilor crescând neconținut de la limbajul mașină, în care o instrucțiune exprima o comandă ce era subliminală problemei de rezolvat, până la limbajele moderne, în care un singur apel de funcție de bibliotecă, ce concentrează mii ori zeci de mii de instrucțiuni mașină, codifică pași semnificativi în rezolvarea problemei. Cea de a doua direcție a însemnat o diversificare a paradigmelor de programare, fiecare, prin trăsăturile ei, oferind o altă alternativă de a gândi o soluție dar și, uneori, o specializare a tipului de probleme la care se pretează.

În [26] colegul meu Dorel Lucanu preia o problemă propusă de Gries [18] pentru a analiza câteva soluții. **Problema platoului:** *Se consideră un șir finit de*

întregi crescători. Să se găsească lungimea celui mai lung platou (șir de întregi egali).

Soluțiile ce urmează intenționează să prezinte caracteristicile definitorii ale celor mai cunoscute paradigme de programare. Ele nu trebuie luate în nici un caz drept singurele soluții posibile în paradigmele respective. Pentru simularea rulărilor vom considera următorul vector de întregi crescători: (1 2 2 2 3).

1.1. Programarea imperativă – rezolvă dictând cum să faci

În imaginarea unei soluții în maniera imperativă, ceea ce contează este depistarea unor operații și găsirea ordinii în care acestea trebuie efectuate. Menționarea unui șir de numere aproape că invită la gândirea unei operații care trebuie repetată pentru fiecare element al șirului. Astfel, în soluția dată în cartea citată, variabila care ține lungimea platoului maxim este întâi inițializată la 1, pentru ca operația care se iterează pentru fiecare element al șirului să fie incrementarea acesteia în cazul în care elementul curent aparține unui platou cu 1 mai lung decât cel considerat maxim până la pasul anterior. Dacă variabila p ține lungimea maximă a platoului curent, iar elementul al i -lea al vectorului este notat $vec[i]$, primul fiind $vec[0]$, atunci apartenența lui $vec[i]$ la un platou de lungime $p+1$ se face verificând dacă elementul aflat la distanță p spre stânga elementului al i -lea este egal cu acesta, adică $vec[i]==vec[i-p]$ (în această condiție este, evident, esențială proprietatea de șir ordonat crescător). Cu aceasta, presupunând lungimea șirului n , algoritmul arată astfel, într-o notație care împrumută mult din limbajul C:

```
for(i=1, p=1; i<n; i++) if vec[i]==vec[i-p] then p++;
```

O formulare echivalentă a acestei soluții este: într-o parcurgere a vectorului stânga-dreapta, pentru toate pozițiile din vector ce corespund unor platouri cu 1 mai lungi decât cel mai lung platou găsit deja, incrementează cu 1 cel mai lung platou.

Execuția programului poate fi urmărită pe următorul tabel:

i	vec[i]	p înainte	i-p	vec[i-p]	vec[i]== vec[i-p]	p după
1	2	1	0	1	false	1
2	2	1	1	2	true	2
3	2	2	1	2	true	3
4	3	3	1	2	false	3

1.2. Programarea logică – rezolvă spunând ce vrei să faci

A imagina o soluție în programarea logică înseamnă a gândi în termeni găsirii de definiții pentru elementele centrale problemei. În cazul problemei noastre, preocuparea este să definim ce înseamnă platoul de lungime maximă al șirului din perspectiva unei anumite poziții a șirului.

Să considerăm un predicat `platou(i, p, n, q)` care afirmă că, dacă cel mai lung platou găsit în subșirul inițial până la indexul i al șirului dat are lungimea p , atunci lungimea platoului maxim al întregului șir, de index final n , este q . În Figura 1 elementele șirului sunt desenate ca mici dreptunghiuri, lungimile maxime ale platourilor, considerate asociate elementelor șirului, sunt marcate prin stegulețe, secvențele din șir cărora li se asociază aceste platouri sunt figurate prin benzi colorate, iar indecșii elementelor care mărginesc în partea dreaptă a acestor subșiruri sunt notați deasupra lor:

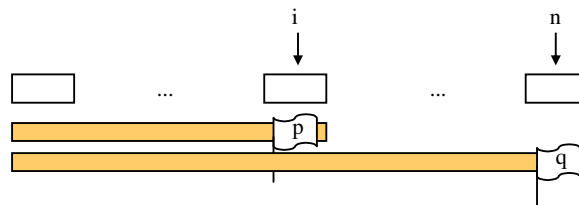


Figura 1: Semnificația predicatului `platou(i, p, n, q)`

Dacă `vector(i, v)` este un predicat care exprimă că elementul de index i din vector are valoarea v (în sensul că se evaluează la *true* în exact acest caz), primul element al vectorului fiind de index 0, atunci, într-o notație apropiată oricărei versiuni a limbajului Prolog, o soluție poate să arate astfel:

```
(1) platou(N, P, N, P) :- !.
(2) platou(I, P, N, Q) :- vector(I+1, V),
                           vector(I+1-P, V),
                           platou(I+1, P+1, N, Q).
(3) platou(I, P, N, Q) :- vector(I+1, V),
                           ~vector(I+1-P, V),
                           platou(I+1, P, N, Q).
```

Prima definiție exprimă situația asociată subșirului de lungime n al șirului dat, caz în care, așa cum se poate constata și din Figura 1, lungimea platoului maxim este q , deci p este egal cu q . Acest predicat este menit să termine demonstrația pentru cazul în care s-a reușit să se arate că lungimea platoului maxim

al unui subșir ce este identic cu șirul dat are lungimea p , ceea ce se răsfânge și asupra lungimii platoului maxim al șirului dat.

Definiția a doua exprimă relația dintre două predicate asociate la doi indecși adiacenți (i și $i+1$) ai vectorului și unde platourile maxime corespunzătoare subșirurilor inițiale ale șirului dat, pentru indecșii indicați, au lungimile p și respectiv $p+1$. Acest caz apare atunci când elementele șirului de indecși $i+1$ și $i+1-p$ sunt egale (vezi Figura 2, în care elementele egale aparținând platoului maxim aflat în dezvoltare sunt notate prin aceeași culoare).

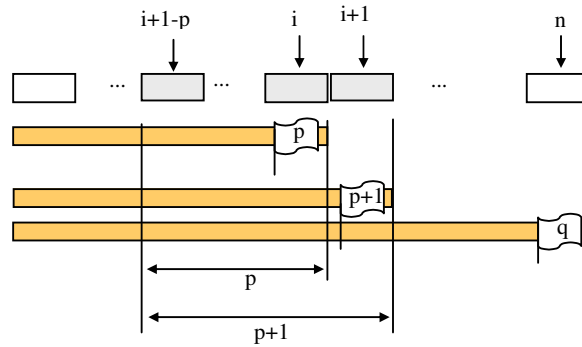


Figura 2: Relația dintre predicatele asociate la doi indecși alăturați cărora le corespund platouri maxime inegale

Textual, această definiție exprimă următoarele: ca să demonstrez că, dacă lungimea platoului inițial al subșirului dat până la indexul i al vectorului este p , lungimea platoului maxim al întregului șir este q în condițiile în care elementele vectorului de indecși $i+1$ și $i+1-p$ sunt egale, atunci trebuie să demonstrez că dacă lungimea platoului maxim al subșirului inițial până la indexul $i+1$ al vectorului este $p+1$, lungimea platoului maxim al vectorului de lungime n este q .

Definiția a treia exprimă relația dintre două predicate asociate la doi indecși adiacenți, cărora le corespund platouri de aceeași lungime p , atunci când elementele șirului de indecși i și $i-p$ sunt inegale (vezi Figura 3).

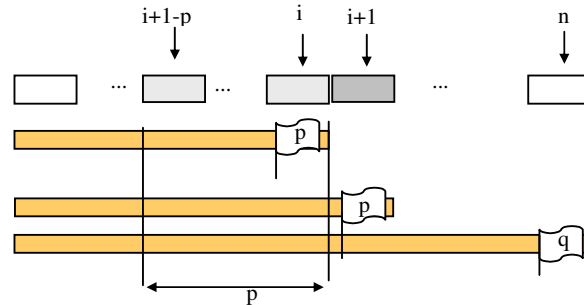


Figura 3: Relația dintre predicatele asociate la doi indecși alăturați cărora le corespund platouri maxime egale

Să remarcăm că deși figura exprimă situația în care platoul maxim al subșirului de index i este dispus în extremitatea dreaptă a subșirului, elementul de index $i+1$ fiind primul care nu aparține platoului, condițiile exprimate în predicat sunt valabile inclusiv pentru cazurile în care platoul maxim al subșirului inițial de index i al vectorului nu se află în capătul din dreapta al acestuia.

Vectorul folosit în secțiunea anterioară, în declarațiile specifice limbajului Prolog, arată acum astfel:

```
vector(0, 1).
vector(1, 2).
vector(2, 2).
vector(3, 2).
vector(4, 3).
```

Pentru acest șir, calculul platoului maxim este amorsat de apelul:

```
?- platou(0, 1, 4, X).
```

care concentrează întrebarea: dacă știm că în subșirul format din primul element al vectorului (până la indexul 0) lungimea platoului maxim are lungimea 1, care este lungimea platoului maxim al întregului șir (până la indexul 4)?

Să observăm că soluția este valabilă inclusiv pentru un șir banal de lungime totală 1. În acest caz răspunsul îl dă direct prima definiție, pentru că predicatul `platou(0, 1, 0, X)` se va potrivi peste faptul `platou(N, P, N, P)`, ceea ce va duce la legarea simultană a variabilelor P și X la valoarea 1.

Întrebarea va genera următorul lanț de confruntări de predicate și legări de variabile:

```

    platou(0, 1, 4, X) versus platou(N, P, N, P) în (1) ➔ eșec;
    platou(0, 1, 4, X) versus platou(I, P, N, Q) în (2) ➔ succes
cu: I=0, P=1, N=4, Q=X;
    vector(1, V) versus vector(1, 2) ➔ succes cu: V=2;
    vector(0, 2) versus vector(0, 1) ➔ eșec;
    platou(0, 1, 4, X) versus platou(I, P, N, Q) în (3) ➔ succes
cu: I=0, P=1, N=4, Q=X;
    vector(1, V) versus vector(1, 2) ➔ succes cu: V=2;
    vector(0, ~2) versus vector(0, 1) ➔ succes;
    platou(1, 2, 4, X) versus platou(N, P, N, P) în (1) ➔ eșec;
    platou(1, 2, 4, X) versus platou(I, P, N, Q) în (2) ➔ succes
cu: I=1, P=2, N=4, Q=X;
    vector(2, V) versus vector(2, 2) ➔ succes cu: V=2;
    vector(0, 2) versus vector(0, 1) ➔ eșec;
    platou(1, 2, 4, X) versus platou(I, P, N, Q) în (3) ➔ succes
cu: I=1, P=2, N=4, Q=X;
    vector(2, V) versus vector(2, 2) ➔ succes cu: V=2;
    vector(0, ~2) versus vector(0, 1) ➔ succes;
    platou(2, 2, 4, X) versus platou(N, P, N, P) în (1) ➔ eșec;
    platou(2, 2, 4, X) versus platou(I, P, N, Q) în (2) ➔ succes
cu: I=2, P=2, N=4, Q=X;
    vector(3, V) versus vector(3, 2) ➔ succes cu: V=2;
    vector(2, 2) versus vector(2, 2) ➔ succes;
    platou(3, 3, 4, X) versus platou(N, P, N, P) în (1) ➔
eșec;
    platou(3, 3, 4, X) versus platou(I, P, N, Q) în (2) ➔
succes cu: I=3, P=3, N=4, Q=X;
    vector(4, V) versus vector(4, 3) ➔ succes cu: V=3;
    vector(1, 3) versus vector(1, 2) ➔ eșec;
    platou(3, 3, 4, X) versus platou(I, P, N, Q) în (3) ➔
succes cu: I=3, P=3, N=4, Q=X;
    vector(4, V) versus vector(4, 3) ➔ succes cu: V=3;
    vector(1, ~3) versus vector(1, 2) ➔ succes;
    platou(4, 3, 4, X) versus platou(N, P, N, P) în (1) ➔
succes cu: N=4, P=3, X=3.

```

1.3. Programarea funcțională – rezolvă apelând o funcție

Noțiunea centrală în programarea funcțională este apelul de funcție. Iterația nu e firească în această paradigmă. Desigur, o iterație se poate realiza în toate limbajele funcționale existente, pentru că întotdeauna proiectanții de limbaje de programare au făcut concesii unor trăsături care impurificau conceptul de bază în avantajul “ergonomiei” actului de programare. Dimpotrivă recursivitatea, utilizând apelul de funcție din interiorul aceleiași funcții, este aici naturală.

O soluție funcțională a problemei platoului urmărește definirea unei funcții care trebuie să întoarcă lungimea platoului maxim al șirului dat, de lungime n , dacă

se cunoaște lungimea p a unui subșir de lungime i al șirului dat. Notând, ca în Lisp, `(platou p i n)` un apel al acestei funcții, atunci o soluție ar putea fi următoarea, dacă un apel `(nth i vector)` întoarce valoarea elementului de pe poziția i din vector, `(eq x y)` întoarce t (*true*) dacă x este egal cu y și nil (*false*) altfel, iar `(if <pred> <form1> <form2>)` întoarce rezultatul evaluării lui `<form1>` dacă `<pred>` se evaluează la t și rezultatul evaluării lui `<form2>` dacă `<pred>` se evaluează la nil :

```
(defun platou (i p n)
  (if (eq i n) p
      (if (eq (nth (- i p) vector) (nth i vector))
          (platou (+ i 1) (+ p 1) n)
          (platou (+ i 1) p n)
      )
  )
)
```

Definiția începe cu condiția de terminare a recursiei, aceeași ca și în soluția dată în Prolog, și anume: dacă se cunoaște lungimea platoului maxim al unui subșir inițial de lungime egală cu lungimea șirului, atunci aceasta este și lungimea platoului maxim al șirului dat. Altfel, dacă valoarea elementului din vector aflat pe poziția i este egală cu cea a elementului aflat pe poziția $i-p$, atunci valoarea platoului maxim al șirului este dată de valoarea întoarsă de aceeași funcție în condițiile în care valoarea platoului maxim pentru subșirul inițial de lungime $i+1$ este $p+1$. Altfel, ea este egală cu valoarea întoarsă de funcție în condițiile în care valoarea platoului maxim pentru subșirul inițial de lungime $i+1$ este tot p .

Rezultatul, pentru șirul nostru de lungime 5 este dat de apelul:

```
(platou 1 1 5)
```

Acest apel antrenează următorul șir de apeluri și rezultate intermediare pentru același vector considerat drept intrare, care aici este comunicat prin setarea `(setf vector '(1 2 2 2 3))`:

```
(platou 1 1 5)
0: (PLATOU 1 1 5)
   ; i=1: vector(1) ≠ vector(0), test false
   ; => p rămâne 1
1: (PLATOU 2 1 5)
   ; i=2: vector(2) = vector(1), test true
   ; => p devine 2
2: (PLATOU 3 2 5)
   ; i=3: vector(3) = vector(1), test true
```

```

        ; => p rămâne 3
3: (PLATOU 4 3 5)
        ; i=4: vector(4) ≠ vector(1), test false
        ; => p rămâne 3
4: (PLATOU 5 3 5)
        ; i=5: terminarea recursiei
4: returned 3
3: returned 3
2: returned 3
1: returned 3
0: returned 3
3
>

```

1.4. Programarea orientată-obiect – rezolvă construind obiecte ce interacționează

Un automobil este un obiect. Dar un automobil este format din caroserie, motor și roți. Fiecare dintre acestea sunt, la rândul lor, obiecte. Așa, spre exemplu, un motor are în componență șasiul, pistoanele, carburatorul, pompele de apă, de ulei, de benzină, generatorul de curent ș.a. Pentru ca o mașină să meargă, este nevoie ca toate obiectele componente, fiecare cu rolul lor, să-și îndeplinească funcțiile, în interacțiune cu celelalte, la momentele de timp când aceste funcții sunt solicitate. De exemplu, roțile motoare trebuie să se învârtască sincron, astfel încât să tragă la fel de tare, cele patru pistoane trebuie să împingă bielele în strictă corelație unele cu altele, exact atunci când aceste operații sunt cerute de un dispecer electronic etc.

Funcționalitatea unui obiect fizic, cum este o mașină, poate fi simulată printr-un program. În acest caz programul modelează obiecte ca cele din lumea reală. Și, tot ca în lumea reală, dacă suntem capabili să “realizăm” o mașină de un anumit tip, să zicem un Renault Meganne, atunci o putem multiplica în oricâte exemplare, producând instanțe ale ei, fiecare utilitate cu caroserie, motor și roți de Renault Meganne.

Ca să rezolvăm problema platoului în maniera orientată-obiect ar trebui să privim fiecare element al vectorului ca pe un obiect înzestrat cu capacitatea de a răspunde la întrebări prin introspecție sau schimbând mesaje cu alte obiecte asemenea lui. Spre exemplu, presupunând o ordonare stânga-dreapta a șirului de numere, să ne imaginăm că am construi un obiect ce ar corespunde unui element al șirului și care ar fi capabil să ne indice, atunci când ar fi interogată:

- pe de o parte, valoarea lui;

- pe de altă parte, presupunând că îi comunicăm lungimea celui mai lung platou de la începutul vectorului până la elementul aflat în stânga lui inclusiv, care este valoarea platoului maxim al întregului vector.

Dacă am avea construite obiecte cu această funcționalitate, atunci am putea afla lungimea platoului maxim al șirului interogând direct primul element, după ce i-am spus că lungimea celui mai lung platou de la începutul vectorului până la el este de 0 elemente.

Să încercăm să ne imaginăm cum ar putea “raționa” un obiect al vectorului pentru a răspunde la cea de a doua întrebare: “dacă știu că cel mai lung platou de la începutul șirului de numere până la elementul din stânga mea inclusiv este p , atunci,

- dacă valoarea mea este egală cu a unui element aflat cu p elemente la stânga, înseamnă că lungimea celui mai lung platou de la începutul șirului până la mine inclusiv este $p+1$ și atunci răspunsul meu trebuie să fie cel dat de vecinul meu din dreapta, atunci când îl voi ruga să-mi comunice lungimea celui mai lung platou al șirului, după ce îi voi fi spus că lungimea celui mai lung platou de la începutul șirului până la mine inclusiv este $p+1$;

- altfel, dacă valoarea mea nu e, deci, egală cu a elementului aflat cu p elemente la stânga, înseamnă că lungimea celui mai lung platou de la începutul șirului până la mine inclusiv este tot p și răspunsul meu trebuie să fie cel dat de vecinul meu din dreapta atunci când îl voi ruga să-mi comunice lungimea celui mai lung platou al șirului, după ce îi voi fi spus că lungimea celui mai lung platou de la începutul șirului până la mine inclusiv este p ”.

După cum se poate vedea deja, pentru a asigura o funcționalitate de acest fel, va trebui să facem ca un obiect – corespunzător unui element al șirului – să comunice, pe de o parte cu obiecte asemenea lui din șir, dar aflate în stânga lui, pentru a afla de la ele valoarea lor, iar pe de altă parte cu obiectul vecin lui în dreapta, pentru a afla de la el lungimea platoului maxim al întregului șir atunci când îi va comunica lungimea platoului maxim al elementelor de până la el. Valoarea aflată de la acesta va fi și răspunsul pe care îl va întoarce la întrebarea care i-a fost inițial adresată lui însuși.

Iată, într-o notație apropiată de limbajul C++, un astfel de program:

```
class Vector
{ int dim;
  int current;
  MyInt vec[5];
  Vector();
}

class MyInt
{ int val;
  MyInt (int i) {val = i;}
```

```

    int getVal (void) {return val;}
    int platou (int);
}

int MyInt::platou(int p)
{ if(vec.current >= vec.dim) return(p);
  if(val == vec[vec.current-p].getVal())
    return(vec[++vec.current].platou(++p));
  else return(vec[++vec.current].platou(p));
}

void main()
{ Vector myVector;
  int p = myVector[1].platou(1);
  printf("Lungimea platoului maxim = %d\n", p);
}

```

1.5. Programarea bazată pe reguli – rezolvă ca într-un joc de Puzzle sau Lego

Puzzle este un joc în care o mulțime de piese de forme și culori diferite pot fi asamblate pentru a forma un tablou. Pentru fiecare piesă, în principiu, există un singur loc în care aceasta poate fi integrată. În Lego dispunem de seturi de piese de același fel ce pot fi îmbinate între ele pentru a realiza diverse construcții. Aceleași piese pot fi utilizate în combinații diferite. În ambele cazuri, un ansamblu sau o construcție se realizează din elemente simple care au, fiecare în parte, funcționalități precizate în tabloul de ansamblu.

Analogia programării bazate pe reguli este mai puternică cu jocul de Lego decât cu cel de Puzzle pentru că, utilizând piesele din set, în Puzzle se poate crea un singur tablou, pe când în Lego putem realiza oricâte construcții. În programarea bazată pe reguli, utilizând aceleași piese de cunoaștere, care sunt regulile, putem, în principiu cel puțin, rezolva orice instanță a aceluiași probleme.

Să revenim la problema noastră încercând o rezolvare prin reguli. Să ne imaginăm că proiectăm testul de incrementare a lungimii unui platou maxim găsit deja. Vom avea în vedere proiectarea unei reguli care să mărească cu o unitate lungimea platoului maxim găsit până la un moment dat. Vom raționa astfel: “dacă știm că un platou de o lungime p a fost deja găsit, atunci putem afirma că am găsit un platou de lungime $p+1$ dacă găsim două elemente egale ale vectorului aflate la distanță p unul de altul în secvență”. Ordinea în care sunt interogate elementele vectorului în acest calcul nu mai este relevantă. În felul acesta, iterația în lungime vectorului, ce apărea într-un fel sau altul în toate soluțiile anterioare, este înlocuită cu o iterație în lungimea platoului, fără ca, de la un pas la următorul în iterație, să se păstreze neapărat o parcurgere în ordine a elementelor vectorului. Această

soluție, pentru cazul aceluiași exemplu de vector pe care l-am mai utilizat, poate fi redată de următorul program CLIPS (aici, pentru prima oară, ca și în restul cărții, secvențele de programe în CLIPS sunt redată pe un font gri):

```
(deffacts initial
  (vector 0 1)
  (vector 1 2)
  (vector 2 2)
  (vector 3 2)
  (vector 4 3)
  (platou 1)
)

(defrule cel-mai-lung-platou
  ?ip <- (platou ?p)
  (vector ?i ?val)
  (vector ?j&:(= ?j (+ ?i ?p)) ?val)
=>
  (retract ?ip)
  (assert (platou (+ ?p 1)))
)
```

Prima parte a programului definește vectorul de elemente și platoul maxim inițial (egal cu 1) prin niște declarații de fapte. Aici nu este necesară nici o declarație care să inițializeze un index al vectorului.

Partea a doua a programului definește o regulă. Numele ei – `cel-mai-lung-platou` – nu e esențial în derularea programului, dar ajută la descifrarea conținutului. Textual, ea spune că, dacă platoul maxim găsit până la momentul curent este `p`, și dacă în vector există două elemente egale aflate la distanța `p` unul de altul, atunci lungimea platoului maxim găsit trebuie actualizată la valoarea `p+1`. La fiecare aplicare a regulii, așadar, va avea loc o incrementare a platoului maxim găsit. Este de presupus că atunci când regula nu se mai poate aplica, faptul care reține valoarea platoului maxim găsit să indice platoul maxim al întregului șir.

Ceea ce urmează reprezintă o trasare comentată a rulării:

```
CLIPS> (reset)
```

CLIPS fiind un limbaj interpretat, comenzile se dau imediat după afișarea prompterului `CLIPS>`.

```
==> f-1      (vector 0 1)
==> f-2      (vector 1 2)
```

```
==> f-3      (vector 2 2)
==> f-4      (vector 3 2)
==> f-5      (vector 4 3)
==> f-6      (platou 1)
```

În liniile care apar după comanda `(reset)` sunt afișate faptele aflate inițial în bază, fiecare însoțit de un index (de la `f-1` la `f-6`). Faptul `(vector 0 1)` memorează valoarea 1 în poziția din vector de index 0 ș.a.m.d. Valoarea inițială a platoului maxim este 1.

```
CLIPS> (run)
```

Ceea ce urmează după comanda `(run)` semnalează aprinderea regulilor.

```
FIRE 1 cel-mai-lung-platou: f-6,f-3,f-4
```

Se aprinde pentru prima oară regula `cel-mai-lung-platou` datorită faptelor cu indicii `f-6`, `f-3` și `f-4`, respectiv: faptul care indică lungimea 1 a platoului maxim, elementul de vector `(vector 2 2)` și elementul de vector `(vector 3 2)`.

```
<== f-6      (platou 1)
```

Ca urmare, lungimea maximă a platoului este actualizată de la 1...

```
==> f-7      (platou 2)
```

... la 2, noul fapt primind indexul `f-7`.

```
FIRE 2 cel-mai-lung-platou: f-7,f-2,f-4
```

Regula se aprinde pentru a doua oară datorită faptelor de indici `f-7`, `f-2` și `f-4`, respectiv: faptul care indică noua lungime 2 a platoului și elementele de vector `(vector 1 2)` și `(vector 3 2)`, care sunt egale și se găsesc la o distanță de 2 elemente.

```
<== f-7      (platou 2)
```

Ca urmare, lungimea maximă a platoului este actualizată de la 2...

```
==> f-8      (platou 3)
```


... la 3. În continuare regula nu mai poate fi aprinsă pentru că în vector nu mai există două elemente egale aflate la o distanță de 3 indecși. Ca urmare rularea se oprește de la sine.

Se poate constata că, dacă în celelalte implementări exemplificate în acest capitol “lungimea” rulării a fost proporțională cu lungimea vectorului, în abordarea prin reguli ea a fost dictată de mărimea platoului maxim. Problema s-a inversat: în loc să iterez sau să recurez pe lungimea vectorului pentru ca la fiecare pas să incrementez sau nu o variabilă ce ține lungimea platoului, iterez pe lungimea platoului maxim și folosesc condiția de incrementare a acestuia drept condiție de terminare a rulării. Într-adevăr, când nu mai găsesc două elemente egale aflate la o distanță mai mare decât ceea ce știu că este lungimea unui platou din cuprinsul vectorului, înseamnă că am aflat răspunsul și pot opri calculele. În același timp, se poate remarca faptul că totul se petrece ca și cum cazurile ce duc la incrementarea variabilei ce “ține” lungimea platoului maxim găsit până la un moment dat ies la iveală singure sau “atrag” regula în care s-a specificat acea condiție, pentru ca ea să fie aplicată.

Continuând metafora de la începutul acestei secțiuni, vedem că exemplul a pus în evidență un joc de *puzzle* cu un singur tip de piesă, dar care a fost folosită de două ori în găsirea soluției. Piesa în chestiune nu face altceva decât să modeleze un microunivers de cunoaștere, încorporând o specificare a unei situații și acțiunile ce trebuie efectuate în eventualitatea că situația este recunoscută.

Aceasta este însăși esența programării bazată pe reguli.

Sintetizând diferența dintre programarea imperativă, cea mai utilizată paradigmă de programare clasică, și programarea bazată pe reguli, putem spune că în maniera imperativă, atunci când condiții diferite antrenează acțiuni diferite, programul trebuie să itereze toate aceste condiții pentru a le găsi pe cele ce pot fi aplicate. Simplificând, putem considera că, în paradigma bazată pe reguli, condițiile sunt organizate în pereche cu acțiunile respective, iar realizarea unei condiții aprinde automat acțiunea corespunzătoare, fără a avea nevoie de o iterare care să parcurgă ansamblul de condiții până la găsirea celei ori celor satisfăcute.

Capitolul 2

Introducere în sistemele expert

O varietate atât de mare de paradigme de programare, ca cea descrisă în capitolul precedent, oglindește necesitatea de a avea la dispoziție limbaje de programare orientate cu precădere spre anumite tipuri de probleme. Specializarea unei paradigme pentru probleme de un anumit tip nu reprezintă însă o restricție de a aplica această paradigmă la orice altă problemă, ci trebuie înțeleasă doar ca preferință. Problemele cu precădere rezolvabile în paradigma programării bazată pe reguli sunt cele din gama **sistemelor expert**.

Potrivit lui Francis Bacon, puterea stă în cunoaștere². Aplicarea acestui concept la sistemele artificiale înseamnă dotarea lor cu abilitatea de a se servi de cunoaștere specifică (cunoaștere expert). Simplificând foarte mult actul medical, putem spune că un medic este valoros atunci când, pus în fața unui bolnav, reușește să-i stabilească un diagnostic corect și, pe baza lui, să indice un tratament care să ducă la vindecarea bolnavului. În stabilirea diagnosticului, medicul se bazează pe un bagaj de cunoștințe generale dar și specifice despre boli și bolnavi. Parțial această cunoaștere a acumulat-o din cărți, parțial în cursul anilor de experiență clinică, prin atâtea cazuri în care s-a implicat și parțial prin puterea minții lui de a corela toate simptomele spre configurarea diagnosticului celui mai reprezentativ. În precizarea tratamentului, el face apel din nou la cunoștințe achiziționate, care arată că în anumite boli sunt indicate anumite medicații, regimuri, exerciții etc., la experiența în tratarea bolnavilor (atunci când aceasta nu s-a transformat într-o aplicare schematică datorită rutinei), dar și la puterea lui de corelație, sinteză și uneori chiar intuiție în a combina toate posibilitățile, în a evalua indicațiile și contra-indicațiile pentru a ajunge la cea mai fericită soluție de tratament.

Nu de puține ori în acest complex act de gândire, în care o problemă nu este aproape niciodată în totalitate rezolvată anterior, pentru că “există bolnavi iar nu boli” și nu s-au născut încă doi indivizi absolut la fel, intervin aprecieri extrem de subtile care țin de capacitatea minții omenești de a opera cu o cunoaștere imperfectă, aproximativă ori parțială, de a trece peste lacune, de a extrapola o serie de experiențe și de a apela la o manieră de raționament nenumeric, bazat pe analogii, aproximații și intuiții. O astfel de cunoaștere este numită, în general,

² Conceptul a fost preluat de E. Feigenbaum pentru a fi aplicat la sistemele inteligente.

cunoaștere expert iar sistemele artificiale care sunt capabile să dezvolte raționamente bazate pe cunoaștere expert se numesc **sisteme expert**.

2.1. Ce sunt sistemele expert ?

Edward Feigenbaum, profesor la Universitatea Stanford, un pionier al tehnologiei sistemelor expert, dă pentru un astfel de sistem următoarea definiție:

“... un program inteligent care folosește cunoaștere și proceduri de inferență pentru a rezolva probleme suficient de dificile încât să necesite o expertiză umană semnificativă pentru găsirea soluției.”

Să ne imaginăm următoarea situație: un domeniu oarecare al cunoașterii și două persoane: un expert în acel domeniu și un novice. Prin ce diferă modul în care novicele, respectiv expertul, abordează probleme din domeniul dat? Desigur, ne putem imagina situația în care novicele are cunoștințe generale asupra domeniului (așa cum se întâmplă de obicei cu domenii cum ar fi cel medical, cel meteorologic etc.). Novicele poate purta o discuție asupra vremii, făcând constatări care pot fi chiar caracterizate drept exacte asupra vremii locale, sau emițând pronosticuri asupra felului în care ea va evolua într-o zi, două. Totuși, cunoașterea sa asupra domeniului este cel puțin nebuloasă și fragmentară, adesea inexactă. Dacă cineva i-ar solicita să explice de ce azi a plouat, de exemplu, ar putea, eventual, argumenta printr-o tendință de răcire care s-a făcut simțită ieri, sau prin acumulările de nori pe care le-a observat venind dinspre nord, direcția obișnuită “de unde plouă”. Dar acestea sunt constatări de relevanță redusă și care dau o explicație foarte aproximativă a fenomenelor.

Sau, să presupunem că atât expertul cât și novicele cunosc toate datele care permit stabilirea evoluției vremii în intervalul imediat următor. Novicele va furniza un pronostic slab, expertul va face o prezicere bună, sau cel puțin apropiată de realitate. De ce? Pentru că expertul are o bună cunoaștere a regulilor care guvernează mișcările de aer, apariția ploii, schimbările de vreme. Putem spune că oricine are o anumită expertiză asupra fenomenelor meteorologice. Diferența constă în completitudinea sistemului și în maniera de sistematizare a acestor cunoștințe.

Așadar, putem spune că motivele pentru care noi, novicii, nu putem furniza o explicație satisfăcătoare asupra ploii de azi sunt cel puțin acestea:

- nu avem acces la o bază de informații suficientă, lipsește o cunoaștere a faptelor, și
- nu cunoaștem regulile care guvernează evoluția vremii, așadar ne lipsește o cunoaștere a fenomenelor.

Oricine știe că dacă ești răcit trebuie să iei aspirină. Dar un medic știe, în plus, că trebuie să interzică acest tratament unui bolnav care manifestă o

hipersensibilitate alergică, pentru că medicul știe că aspirina conține substanțe alergene care îi pot provoca o criză de astm. În acest caz, un plus de cunoaștere permite evitarea unor greșeli.

Multă lume dorește să slăbească. Pentru a da jos câteva kilograme în plus, mulți sunt dispuși să țină diete foarte severe, mâncând mai puțină pâine, renunțând la o masă, sau înfrânându-și pofta de a savura o prăjitură. Aceste restricții sunt gândite ca fiind naturale în a împiedica procesul de îngrășare. Dar puțini știu că există o metodă prin care poți să-ți menții o greutate riguros constantă fără a recurge la privațiuni, mâncând la fel de mult ca și înainte și din toate alimentele care-ți plac. Este vorba de regimul disociat rapid al lui William Howard Hay (1866-1940): secretul constă în a separa proteinele de lipide și de glucide la fiecare masă. Deci poți mânca pâine, dar nu împreună cu carne, poți mânca carne, dar fără legume și poți mânca legume la discreție împreună cu orice altceva. În acest caz, alimentația poate fi dirijată de o cunoaștere aprofundată a metabolismului corpului omenesc. Rezultatul practic: împiedicarea creșterii în greutate.

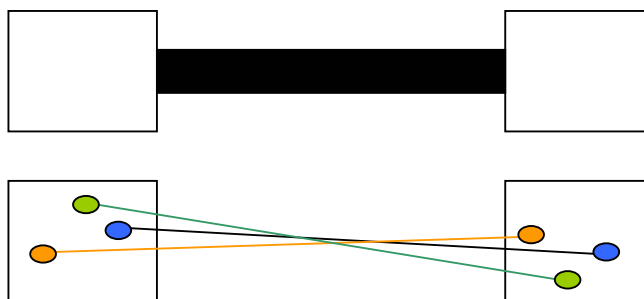


Figura 4: Aprofundarea cunoașterii înseamnă rafinarea conceptelor și a legăturilor dintre ele

Cunoașterea expertului este organizată, precisă, punctuală; a novicei este nestructurată, amorfă, globală. Insistând mai mult asupra acestui aspect, am putea spune că, pe măsură ce se aprofundează un domeniu, se rafinează conceptele domeniului și, ca urmare, și conexiunile ce se stabilesc între aceste concepte devin mai specifice (v. Figura 4).

Tehnologia sistemelor expert face parte din domeniul Inteligenței Artificiale, acea ramură a informaticii care se preocupă de dezvoltarea unor programe care să emuleze capacități cognitive (rezolvarea de probleme, percepția vizuală, înțelegerea limbajului natural etc.). Tehnologia sistemelor expert a oferit deja soluții interesante în diverse domenii: chimie organică, medicină internă și infecțioasă, diagnosticare tehnică, prospecțiuni miniere. Deși în fiecare din aceste domenii s-au putut realiza sarcini asemănătoare utilizând metode clasice de

programare, maniera de abordare a sistemelor expert este suficient de diferită pentru a merita o tratare aparte.

Achiziționarea, formalizarea și includerea cunoașterii expert în sistemele artificiale reprezintă scopul domeniului sistemelor expert.

2.2. Particularități ale domeniului inteligenței artificiale

Există o seamă de trăsături care diferențiază domeniul inteligenței artificiale de alte domenii ale informaticii. Pot fi considerate definitorii cel puțin următoarele trăsături:

- problemele de inteligență artificială necesită, în general, un raționament predominant simbolic;
- problemele se pretează greu la soluții algoritmice. De multe ori soluția poate fi rezultatul unei căutări într-un spațiu al soluțiilor posibile;
- problemele care invită la investigații tipice domeniului inteligenței artificiale sunt și cele care manipulează informație incompletă ori nesigură;
- nu se cere cu necesitate ca soluția să fie cea mai bună sau cea mai exactă. Uneori e suficient dacă se găsește o soluție, sau dacă se obține o formulare aproximativă a ei;
- în rezolvarea problemelor de inteligență artificială intervin adesea volume foarte mari de informații specifice. Găsirea unui diagnostic medical nu poate fi algoritmicizată, pentru că diferențele de date asupra pacientului duc la tipuri de soluții diferite;
- natura cunoașterii ce se manipulează în problemele de inteligență artificială poate fi ușor clasificată în procedurală și declarativă: este diferența dintre cunoașterea pe care o posedă un păianjen față de cea a unui inginer constructor, sau diferența dintre cunoașterea pe care o posedă un jucător de tenis și cea pe care o posedă un bun antrenor. Cunoașterea unuia este instinctivă, ori “în vârful degetelor”, a celuilalt constă într-un sistem de reguli.

Pot fi rezolvate probleme de inteligență artificială prin algoritmi clasici? Cu siguranță, dar soluțiile vor fi probabil greoaie, greu generalizabile, nefirești și neelegante.

Invers: pot fi rezolvate probleme clasice prin metode ale inteligenței artificiale? Iarăși lucrul este posibil, dar e ca și cum am folosi un strung ca să ascuțim un creion.

Un program de calcul al salariilor nu este un program de inteligență artificială, pe când un program care conduce un robot care mătură prin casă fără să distrugă mobila sau unul care recunoaște figuri umane sunt aplicații ale inteligenței artificiale.

2.3. Prin ce diferă un sistem expert de un program clasic?

Vom inventaria în cele ce urmează o seamă de trăsături care caracterizează diferența dintre sistemele expert și programele clasice (v. și [16]).

Modularitate. Cunoașterea care stă la baza puterii de raționament a unui sistem expert este divizată în reguli. În felul acesta piese elementare de cunoaștere pot fi ușor adăugate, modificate ori eliminate. Modularitatea reprezentării cunoașterii asigură totodată și posibilitatea de menținere la zi a bazei de cunoștințe de către mai mulți experți simultan. Ea poate reprezenta astfel opera unui colectiv de autori, adesea dezvoltându-se pe o perioadă lungă de timp, simultan cu intrarea ei în folosință.

Transparență. Un sistem expert poate explica soluția pe care o dă la o anumită problemă. Acesta este, de altfel, un factor de importanță majoră în asigurarea credibilității sistemelor expert puse să furnizeze diagnostice medicale, de exemplu. Pentru ca un medic să aibă încredere într-un diagnostic furnizat de mașină, el trebuie să îl înțeleagă.

Soluții în condiții de incertitudine. Sistemele expert pot oferi, în general, soluții problemelor care se bazează pe date nesigure ori incomplete. Dintr-un anumit punct de vedere un sistem expert funcționează ca o mașinărie care știe să niveleze asperitățile, ori care poate trece cu ușurință peste ele. Adesea un mecanism foarte fin este și foarte pretențios, el putând funcționa în exact condițiile pentru care a fost proiectat. Acesta este și cazul unui program clasic, pentru care neputința de a furniza valoarea exactă a unui parametru îl poate arunca pe o condiție de eroare. Un sistem expert este, în general, mult mai adaptabil pentru domenii difuze, adică este pregătit să facă față fie unor cunoștințe incomplete ori incerte asupra domeniului de expertiză, fie unor date de intrare incomplete ori incerte.

Categorii de sisteme expert. În funcție de domeniul lor de aplicabilitate, sistemele expert pot fi împărțite în trei categorii importante (după [16]):

1. Sisteme expert de diagnostic (sau clasificare). Problemele tratate de acestea pot fi recunoscute după următoarele proprietăți:

- domeniul constă din două mulțimi finite, disjuncte – una conținând observații, cealaltă soluții – și dintr-o cunoaștere complexă, adesea incertă și incompletă despre relațiile dintre aceste două mulțimi;
- problema este definită printr-o mulțime de observații, mulțime ce poate fi incompletă;
- rezultatul diagnosticului (clasificării) este selecția uneia sau mai multor soluții ale problemei;

- în cazul în care calitatea soluției poate fi îmbunătățită prin considerarea unor observații suplimentare, una din sarcinile clasificării o reprezintă găsirea acelei submulțimii de observații suplimentare care ar trebui cerute pentru a le completa pe cele existente.

Exemple din această categorie sunt:

- **diagnosticarea motoarelor de automobil** – unde sistemul expert este un program cuplat *on-line* cu dispozitive electronice care măsoară diverși parametri tehnici ai motorului (consum de benzină, unghiul de reglare al camelor, capacitatea de încărcare a bateriei etc) [36]. O valoare a unui anumit parametru, detectabilă prin senzori ca fiind ieșită din limitele normale, este apoi pusă în legătură cu o disfuncționalitate a unui organ al motorului și, de aici, cu piesa care trebuie înlocuită sau cu efectuarea unui anumit reglaj;

- **diagnosticarea hardware a calculatoarelor** – teste efectuate asupra calculatoarelor pot indica o funcționare eronată. Sistemul expert, de o complexitate mult mai mică decât în sistemele de diagnostic medical, de exemplu, este utilizat pentru indicarea componentei defecte ce se recomandă a fi înlocuită;

- **diagnosticarea rețelelor de calculatoare sau a rețelelor de distribuire a energiei** – mesaje de control, ori teste efectuate în anumite puncte importante, verifică satisfacerea protocoalelor pe liniile de comunicații ale rețelelor ori integritatea fizică a rețelelor cu configurații complicate. Dacă apare o defecțiune, ea este întâi semnalată. În continuare, iterativ, aria de investigații este micșorată până la izolarea completă a defecțiunii;

- **identificarea zăcămintelor minerale** – în geologie, recunoașterea într-o anumită zonă a anumitor roci poate fi pusă în legătură cu identificarea formațiunilor scoarței și, de aici, cu existența unor zăcăminte în arii adiacente. Cunoștințe de această natură pot ghida procesele de foraj pentru identificarea zăcămintelor petrolifere ori de gaze naturale și astfel pot contribui la micșorarea prețurilor pentru identificarea ori demarcarea zăcămintelor.

2. Sisteme expert de construcție: aici soluția nu mai poate fi găsită prin căutarea într-o mulțime existentă. Soluția este acum construită ca o secvență de pași ori o configurație de elemente intercondiționate (astfel văzută, o problemă de diagnostic poate fi considerată un caz special al unei probleme de construcție). Definirea problemei înseamnă precizarea condițiilor inițiale ale problemei, precizarea cerințelor asupra soluției și a spațiului soluțiilor (combinațiile teoretic posibile de obiecte elementare care respectă ori nu cerințele).

Exemple din această categorie:

- **asistent de vânzări în comerț** – un sistem expert poate recomanda produse unor clienți răspunzând întrebărilor acestora și furnizând recomandări cu aceeași dezinvoltură ca un foarte experimentat agent comercial. Prin întrebări abil alese el reușește să configureze un model al cumpărătorului și, prin aceasta, să vină

în întâmpinarea dorințelor sale, construind oferte care să maximizeze șansele de vânzare a produselor;

- **configurarea calculatoarelor** – un dialog cu clientul poate duce la determinarea configurației de calculator personal care să răspundă cel mai adecvat nevoilor acestuia.

3. Sisteme expert de simulare: dacă în sistemele expert de diagnostic și construcție soluția era selectată ori respectiv asamblată, simularea servește numai pentru prezicerea efectelor anumitor presupuziții asupra unui sistem. Un sistem este privit ca o unitate a cărei comportare poate fi inferată din cunoașterea comportării părților componente. Simularea constă din determinarea valorilor unor parametri de ieșire din valorile date ale unor parametri de intrare. Adesea o simulare este cerută pentru a verifica dacă soluția oferită de un sistem expert proiectat pentru funcționa în diagnostic sau construcție este într-adevăr cea dorită.

2.4. Exemple de sisteme expert

DENDRAL. Creație a unei echipe de la Universitatea Stanford [4], [17], [25], conduse de Edward Feigenbaum, DENDRAL apare într-o primă versiune în 1965. Intenția construirii sistemului a fost de a demonstra că metodologia domeniului inteligenței artificiale poate fi utilizată pentru formalizarea cunoașterii științifice, domeniul de aplicabilitate al sistemului însă chimia organică. DENDRAL a oferit o demonstrație convingătoare a puterii sistemelor expert bazate pe reguli. Implementând o căutare de tipul plan-generare-test asupra datelor din spectroscopia de masă și din alte surse, sistemul era capabil să prezică structuri candidate plauzibile pentru compuși necunoscuți, pentru anumite clase de compuși performanța sa rivalizând cu aceea a unor experți umani. Dezvoltări ulterioare au dus la crearea sistemului GENOA – un generator interactiv de structuri (1983). În DENDRAL însă cunoașterea expert era mixată cu algoritmica de rezolvare a problemei. Separarea completă a cunoașterii declarative de cea executorie a însemnat un pas mare înainte înspre definirea unor tehnologii rapide de dezvoltare a sistemelor expert.

META-DENDRAL (1970-76) [5] este un program care formulează reguli pentru DENDRAL, aplicabile domeniului spectroscopiei de masă. El a reușit să redescopere reguli cunoscute despre compuși chimici dar a formulat și reguli complet noi. Experimentele cu META-DENDRAL au confirmat că inducția poate fi automatizată ca un proces de căutare euristică și că, pentru eficiență, căutarea poate fi despărțită în doi pași: o fază de aproximare a soluției și una de rafinare a ei.

MYCIN. Proiectat de Buchanan și Shortliffe la mijlocul anilor 1970 [6], [7] pentru a ajuta medicul în diagnosticul și tratamentul meningitelor și al infecțiilor bacteriene ale sângelui. Adesea diagnosticarea unei infecții, mai ales apărută în

urma unei operații, este un proces laborios și de lungă durată. MYCIN a fost creat pentru a scurta acest interval și a furniza indicații de diagnostic și tratament chiar în lipsa unor teste complete de laborator. Este remarcabilă includerea în sistem a unei componente care să explice motivațiile răspunsului dat. Constructiv, MYCIN implementa o strategie de control cu înlănțuire înapoi ghidată de scop.

Rezultat al experienței dobândite cu MYCIN, autorii lui creează apoi **EMYCIN** (*Empty MYCIN* sau *Essential MYCIN*), obținut din MYCIN prin golirea sa de cunoștințe dependente de domeniu. Acesta a fost considerat primul *shell* de sisteme expert, așadar un mediu de dezvoltare a acestora, cuprinzând motorul de inferențe și utilitare de dezvoltare și consultare a bazei de cunoștințe. Sistemul a fost intens utilizat în Statele Unite și în afara lor, una dintre aplicații fiind sistemul Personal Consultant dezvoltat de Texas Instruments.

MYCIN și EMYCIN a stimulat crearea unei pleiade întregi de sisteme expert sau medii de asistență în dezvoltarea sistemelor expert:

TEIRESIAS [11] – asistent de achiziție a cunoștințelor de tip MYCIN;

PUFF [19], [1] – primul sistem construit cu EMYCIN, dedicat interpretării testelor funcționale pulmonare pentru bolnavii cu afecțiuni de plămâni, în folosință la Pacific Medical Center din San Francisco;

VM [12] – Ventilator Manager, program de interpretare a datelor cantitative în unitățile de terapie intensivă din spitale, capabil să monitorizeze un pacient în evoluția lui și să modifice tratamentul corespunzător;

GUIDON [19] – sistem utilizat în structurarea cunoașterii reprezentate prin reguli pentru scopuri didactice, în realizarea de sesiuni interactive cu studenții – domeniu cunoscut sub numele Instruire Inteligentă Asistată de Calculator (ICAI), experiența cu GUIDON a demonstrat necesitatea de a explicita cunoașterea depozitată în reguli pentru ca ea să devină efectivă pentru scopuri didactice).

AM [2], [11], [23] este un program de învățare automată prin descoperiri utilizat în domeniul matematicilor elementare. Folosind o bază de 243 de euristici AM a propus concepte matematice plauzibile, a obținut date asupra lor, a observat regularități și, completând ciclul demonstrațiilor din matematică, a găsit calea de a scurta unele demonstrații propunând noi definiții. AM nu a reușit însă să găsească el însuși noi euristici pentru a-și perfecționa, într-un fel de cerc vicios al câștigului, propria personalitate. Acest eșec, pus pe seama principiilor sale constructive, a stimulat cercetările pentru crearea unui sistem care să combine capacitatea de a face descoperiri automate, a lui AM, cu trăsătura de a formula noi euristici.

S-a ajuns în acest fel la **EURISKO** (1978-1984) [24]. În orice domeniu este aplicat sistemul are trei niveluri la care poate lucra: cel al domeniului, pentru rezolvarea problemei, cel al inventării de noi concepte ale domeniului și cel al sintezei de noi euristici care sunt specifice domeniului. A fost aplicat în matematica elementară, în programare pentru descoperirea de erori de programare, în jocuri strategice navale și în proiectarea VLSI.

Foarte multe sisteme expert au fost folosite cu succes în discipline ale pământului, ca geologia, geofizica ori pedologia. O anumită vâlvă, în anii '80, a stârnit sistemul **PROSPECTOR** [21] când s-a anunțat că datorită lui s-a reușit descoperirea unui depozit mineral valorând 100.000.000 USD. Mult mai recent, **COAMES** (COAstal Management Expert System) este un sistem expert dezvoltat de Plymouth Marine Laboratory cu intenția de a studia zonele de coastă de o manieră holistică, prin coroborarea datelor de natură biologică, chimică și fizică ce completează tabloul riveran, maritim și atmosferic al acestora. Se așteaptă ca acest sistem să contribuie la dimensionarea corectă a managementului mediului [30].

Sute de sisteme expert au fost descrise în cărți sau reviste dedicate domeniului, cele mai importante dintre reviste fiind:

- *Expert Systems with Applications*, Pergamon Press Inc.
- *Expert Systems: The International Journal of Knowledge Engineering*, Learned Information Ltd.
- *International Journal of Expert Systems*, JAI Press Inc.
- *Knowledge Engineering Review*, Cambridge University Press,
- *International Journal of Applied Expert Systems*, Taylor Graham Publishing

Pentru cercetătorii acestui domeniu este din ce în ce mai evident că problema fundamentală în înțelegerea inteligenței nu este identificarea câtorva tehnici foarte puternice de prelucrare a informației, ci problema reprezentării și manipulării unor mari cantități de cunoștințe de o manieră care să permită folosirea lor efectivă și inter-corelată. Această constatare caracterizează și tendințele fundamentale de cercetare: ele nu sunt îndreptate atât spre descoperirea unor tehnici noi de raționament, cât spre probleme de organizare a bazelor de cunoștințe foarte mari ori de formalizare a cunoștințelor “disipate” în baze de date în sisteme de reguli (*data mining*).

Nu este, desigur, lipsită de interes și problema achiziției cunoașterii din medii naturale, cu precădere din experiența umană [3]. Problema aici este cum ar putea fi identificată, formalizată și transpusă în reguli expertiza specialiștilor umani? Responsabilul cu această sarcină este, în general, cunoscut sub numele de **inginer de cunoaștere** sau **inginerul bazei de cunoștințe**. El este cel care trebuie să găsească limbajul comun cu specialiști din domeniul viitorului sistem expert, care trebuie să-i convingă să colaboreze și să poarte un dialog cu ei, pentru ca apoi să aducă la o formă convenabilă și să introducă în sistemul artificial informațiile furnizate de aceștia [22].

2.5. Evaluarea oportunității sistemelor expert

Construcția unui sistem expert care să facă față unei probleme reale este un proces, cel mai adesea, laborios. De aceea el presupune o fază de pregătire în care să se aprecieze gradul în care acest efort se justifică.

Tabelul de mai jos este reprodus din [32] (care la rândul său preia din [34]) conține 40 de criterii ponderate de evaluare a oportunității construirii unui sistem expert. Criteriile sunt clasificate în esențiale și dezirabile, cele esențiale având ponderi mai mari decât cele dezirabile.

Criterii esențiale

Nr.	Pondere	
1	1	Utilizatorii sistemului așteaptă beneficii mari în operații de rutină.
2	1	Utilizatorii au pretenții realiste asupra mărimii și limitelor sistemului.
3	1	Proiectul are un mare impact managerial.
4	1	Sarcina nu necesită o procesare a limbajului natural.
5	0,7	Sarcina este bazată pe cunoaștere într-o măsură destul de mare, dar nu exagerată.
6	0,8	Sarcina este de natură euristică.
7	1	Sunt disponibile cazuri de test pentru toate gradele de dificultate.
8	0,7	Sistemul se poate dezvolta în etape (sarcina e divizibilă).
9	1	Sarcina nu necesită de loc aprecieri de bun-simț, sau necesită numai foarte puține.
10	0,8	Nu se cer soluții optime ale problemelor.
11	1	Sarcina va fi relevantă și în viitorul apropiat.
12	0,7	Nu e esențial ca sistemul să fie gata în scurt timp.
13	0,8	Sarcina e ușoară, dar nu foarte ușoară pentru sisteme expert.
14	1	Există un expert.
15	1	Expertul este cu adevărat valoros.
16	1	Expertul este implicat în proiect pe toată durata lui.
17	0,8	Expertul e cooperant.
18	0,8	Expertul e coerent în exprimare.
19	0,8	Expertul are o anumită experiență în proiectare.
20	0,8	Expertul folosește raționament simbolic.
21	0,7	Cunoașterea expert este dificil, dar nu foarte dificil de transferat (de exemplu ea ar putea fi predată).
22	1	Expertul rezolvă problemele prin aptitudini cognitive, nu motorii sau senzoriale.

23	1	Experți diferiți sunt de acord asupra a ceea ce înseamnă o soluție bună.
24	1	Expertul nu trebuie să fie creativ în rezolvarea problemei.

Criterii dezirabile

25	0,8	Conducerea va susține proiectul după terminarea lui.
26	0,4	Introducerea sistemului nu va necesita multă reorganizare.
27	0,4	Utilizatorul poate interacționa cu sistemul.
28	0,4	Sistemul își poate explica raționamentul utilizatorului.
29	0,4	Sistemul nu pune prea multe întrebări și nu pune întrebări ce nu sunt necesare.
30	0,4	Sarcina era cunoscută anterior ca fiind problematică.
31	0,4	Soluțiile problemelor sunt explicabile.
32	0,5	Sarcina nu necesită un timp de răspuns prea scurt.
33	0,8	Există sisteme expert de succes care se aseamănă cu sistemul planificat.
34	0,5	Sistemul planificat poate fi folosit în mai multe locuri.
35	0,3	Sarcina e primejdioasă sau, cel puțin, neatractivă pentru oameni.
36	0,4	Sarcina include și cunoaștere subiectivă.
37	0,3	Expertul nu va fi disponibil în viitor (de exemplu, datorită pensionării).
38	0,4	Expertul e capabil să se identifice intelectual cu proiectul.
39	0,4	Expertul nu se simte amenințat.
40	0,2	Cunoașterea folosită de expert la rezolvarea problemelor este slab structurată sau deloc.

Pentru a aprecia oportunitatea construirii unui sistem expert, trebuie atribuite scoruri între 1 și 10 răspunsurilor la întrebări; scorurile trebuie ponderate cu coeficienții atașați și calculată media. Dacă nota obținută este peste 7,5 înseamnă că efortul merită a fi făcut. O notă între 6 și 7,5 semnaleză costuri mari și productivitate mică, invitând la meditație și la încercarea de a reprojeta specificațiile pentru a îmbunătăți acele răspunsuri care au avut cea mai mare contribuție la acest rezultat slab. O notă mai mică de 6 indică cu tărie renunțarea la tentativă.

Partea a II-a

Organizarea și funcționarea unui sistem expert

Organizarea unui sistem expert

Înlănțuirea regulilor în motoarele de inferență

Regimul de lucru tentativ

Confruntarea rapidă de șabloane: algoritmul RETE

Capitolul 3

Organizarea unui sistem expert

Premisa principală pe care se bazează concepția constructivă a sistemelor expert este aceea că un expert uman își construiește soluția la o problemă din piese elementare de cunoaștere, stăpânite de acesta anterior enunțului problemei, și pe care expertul le selectează și le aplică într-o anumită secvență. Pentru a furniza o soluție coerentă la o problemă dată, cunoașterea cuprinsă într-un anumit domeniu trebuie să fi fost inițial formalizată, apoi reprezentată într-o formă adecvată proceselor de inferență și, în final, manipulată în conformitate cu o anumită metodă de rezolvare de probleme. Se pune astfel în evidență diviziunea dintre secțiunea care păstrează reprezentarea cunoașterii asupra domeniului cât și a datelor problemei – **baza de cunoștințe** – și secțiunea responsabilă cu organizarea proceselor inferențiale care să implice aceste cunoștințe – **sistemul de control** (sau **motorul de inferențe**). Acestea sunt, tradițional, cele două module principale ale unui sistem expert (v. Figura 5).

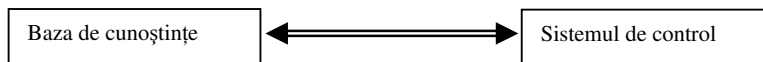


Figura 5: Componentele principale ale unui sistem expert

Aceste două module formează inima unui sistem expert. În jurul lor alte componente realizează funcționalitatea complexă a unui sistem expert, după cum urmează:

- **interfața de comunicații** controlează dialogul sistemului expert cu exteriorul. Prin intermediul acesteia, sistemul expert preia parametrii problemei pe măsură ce componenta de control are nevoie de ei, fie interogând un utilizator uman, fie preluându-i direct dintr-un proces industrial. Dacă interfața realizează un dialog cu un partener uman, atunci sistemul se numește interactiv, în caz contrar el este un sistem implantat (embedded system);

- **componenta de achiziție a cunoștințelor** permite introducerea și, ulterior, modificarea cunoștințelor din bază. O funcționalitate minimă trebuie să permită selecția, introducerea, modificarea și tipărirea regulilor și a faptelor. Procesul de achiziție a cunoștințelor fiind esențial pentru reușita unei aplicații, componentele de

achiziție atașate shell-urilor de sistem expert devin tot mai evaluate, putând încorpora trăsături de învățare, de generalizare, sau putând lansa un proces inferențial în scopul verificării consistenței ori completitudinii bazei;

- **componenta explicativă** asigură “transparența” în funcționare a sistemului expert. Ea ajută atât utilizatorul care caută o fundamentare a soluției oferite de sistem, cât și pe inginerul bazei de cunoștințe în faza de depistare a erorilor.

Baza de cunoștințe, care este componenta sistemului responsabilă cu depozitarea de informații asupra domeniului de expertiză și a problemei de rezolvat, este la rândul ei formată din:

- **baza (ori colecția) de fapte**: memorează datele specifice problemei de rezolvat, informații de natură, în general, volatilă, pentru că, pe parcursul unui proces de inferență, se pot adăuga fapte noi, iar altele vechi pot fi retrase.

- **baza (ori colecția) de reguli**: cuprinde piesele de cunoaștere care vor ghida procesele de raționament ale sistemului. Recunoaștem aici o secțiune principală, care trebuie să cuprindă cunoașterea specifică domeniului, și o secțiune dedicată meta-cunoașterii (sau regulilor de control) – acel set de reguli care realizează tranzițiile între faze ori ajută la derularea operațiilor auxiliare.

- **agenda** (numită și **memoria de lucru**): depozitează informații legate de activarea regulilor, instanțe de reguli, legări de variabile etc, toate acestea fiind necesare derulării proceselor de inferență.

Sistemul de control este responsabil cu desfășurarea proceselor de inferență. El codifică una sau mai multe strategii de aplicare a regulilor (v. și [20]).

3.1. Baza de cunoștințe: faptele

Faptele din bază codifică obiecte, configurația lor de însușiri și valori particulare ale acestor însușiri. În general, există două tipuri de cunoștințe ce se pot memora în secțiunea de fapte a bazei de cunoștințe: declarații asupra structurii obiectelor (precum numele generice ale obiectelor, configurația lor de attribute și tipurile de valori pe care le pot avea attributele) și declarații asupra conținutului informațional al obiectelor (numele ce le individualizează, valorile atributelor).

Dintr-un alt punct de vedere, informațiile conținute în baza de cunoștințe pot fi permanente (cum sunt de exemplu declarații care configurează obiectele, dar și declarații de conținut asupra unor obiecte imuabile în timpul procesării) sau volatile (date ce sunt de utilitate temporară sau care contribuie la derularea procesului de inferență în anumite faze și trebuie apoi șterse pentru a nu încărca inutil memoria).

Astfel, pentru un domeniu medical, anumite reguli pot memora diverse simptome ale bolnavilor. Ca să exprimăm faptul că pacientul Ionescu are gâtul țeapăn, putem utiliza un fapt care să aibă codificate într-un anume fel informațiile privitoare la numele pacientului – Ionescu, la simptomul de interes – mobilitate_gât

și la valoarea acestuia – mică. Maniera practică de codificare diferă atât de la un limbaj bazat de reguli la altul, cât și în funcție de opțiunea de a utiliza codificări nestructurate sau obiectuale (în spiritul paradigmei orientate obiect). Într-un fel sau altul, codificarea trebuie să cuprindă următoarele informații:

```
[nume_pacient = Ionescu, mobilitate_gât = mică]
```

Într-o aplicație de construcții geometrice în care avem nevoie să codificăm puncte și triunghiuri, de exemplu, am putea exprima faptul că A, B și C sunt puncte și că ABC este un triunghi, într-o multitudine de feluri, dar, oricum ar fi ea, această informație trebuie să ne permită să identificăm numele punctelor și numele triunghiului:

```
[punct = A]
[punct = B]
[punct = C]
[triunghi = ABC]
```

Într-o aplicație de fizică ce-și propune să calculeze circuite în curent alternativ se operează cu impedanțe, capacități și rezistențe. Cum cea mai firească reprezentare a acestora este ca numere complexe, trebuie avută în vedere o manieră de a reprezenta diverse numere complexe. Dacă $100 + 80 * i$ este un astfel de număr, atunci o reprezentare trebuie să cuprindă minimum partea reală și pe cea imaginară, adică:

```
[100, 80]
```

Importantă pentru organizarea faptelor din baza de cunoștințe este observația că un fapt este întotdeauna reprezentat cu unicitate. Această observație are două implicații: una legată de maniera de operare a sistemului de control – care nu va permite includerea simultană în bază a două fapte identic structurate – și una legată de păstrarea consistenței informaționale a bazei și care adaugă o responsabilitate programatorului: acesta trebuie să aibă grijă ca baza să nu cuprindă informații contradictorii.

Legat de prima observație, în exemplul medical considerat sistemul nu va permite mobilarea bazei cu două fapte identice (în sens sintactic), ca de exemplu:

```
[pacient = Ionescu, mobilitate_gât = mică]
[pacient = Ionescu, mobilitate_gât = mică]
```

Legat de a doua observație, programatorul nu trebuie să permită existența simultană a unor fapte de forma:

```
[pacient = Ionescu, mobilitate_gât = mică]
[pacient = Ionescu, mobilitate_gât = normală]
```

care, altfel, ar putea fi acceptate de sistem, pentru că, din punct de vedere strict sintactic, ele sunt diferite. Dacă însă considerente legate de modelarea evoluției simptomelor pacienților într-o perioadă de timp impun necesitatea de a avea ambele fapte în bază, pentru că, de exemplu, Ionescu avea inițial gâtul țeapăn pentru ca, în urma tratamentului, acesta să fi revenit la normal, atunci trebuie adăugată o informație asupra datei preluării observațiilor, adică ceva de genul:

```
[data = 22-dec-01, pacient = Ionescu,
                               mobilitate_gât = mică]
[data = 18-ian-02, pacient = Ionescu,
                               mobilitate_gât = normală]
```

Observația asupra unicității înregistrărilor din baza de fapte obligă la o reconsiderare a modului de reprezentare a numerelor complexe în exemplul preluat din fizică, pentru că foarte repede vom constata că o structură a faptului ce memorează numere complexe în care putem identifica doar componenta reală și pe cea imaginară face imposibilă existența simultană în bază a două numere complexe egale (de exemplu, caracterizând două componente electrice de impedanțe egale). Devine astfel evident că trebuie adăugată în reprezentare o informație care să identifice unic un număr complex:

```
[complex = c1, real = 100, imaginar = 80]
[complex = c2, real = 100, imaginar = 80]
```

3.2. Regulile

Regulile codifică transformările ce trebuie operate asupra obiectelor din baza de cunoștințe. O regulă este o entitate de forma:

```
<nume regulă>: [<comentariu>]
dacă <condiții> atunci <acțiuni>
```

Adesea condițiile se mai numesc și partea stângă a regulii, iar acțiunile – partea sa dreaptă. Activitatea sistemului de control legată de o regulă poate fi exprimată simplificat astfel: dacă partea de condiții este satisfăcută de faptele existente în baza de cunoștințe, atunci acțiunile pot fi executate. Acțiunile dictează modificările ce trebuie operate asupra bazei.

Iată un exemplu de regulă dintr-un domeniu medical (exemplu adaptat după [32]):

```

diagnostic_meningită:
dacă
    pacientul are gâtul țeapăn,
    pacientul are temperatură mare,
    pacientul are dese pierderi de cunoștință,
atunci
    introdu în bază informația că pacientul e suspect de
    meningită.

```

Pentru ca această regulă să fie luată în considerare de către componenta de control în vederea aplicării, sistemul trebuie să găsească în baza de cunoștințe trei fapte care, într-un fel sau altul, să precizeze că mobilitatea gâtului pacientului X este mică, că temperatura pacientului X este mare și că pierderile de cunoștință ale pacientului X sunt frecvente, adică ceva de genul:

```

[pacient = Ionescu, mobilitate_gât = mică]
[pacient = Ionescu, temperatură = mare]
[pacient = Ionescu, pierderi_cunoștință = frecvente]

```

În continuare, dacă sistemul de control decide să aplice această regulă, în baza de fapte va fi adăugat un nou fapt, ce va memora că diagnosticul posibil al pacientului Ionescu este meningită:

```

[pacient = Ionescu, diagnostic_posibil = meningită]

```

În aplicația de construcții geometrice sugerată mai sus, următoarea indicație de construcție:

```

construcție_triunghi:
dacă
    A, B și C sunt trei puncte diferite între ele
atunci
    introdu în bază informația că ABC este un triunghi

```

poate fi exprimată ca o regulă. Aplicarea ei, amorsată de descoperirea în bază a trei obiecte de tip punct geometric, ar trebui să ducă la construirea unui obiect de tip triunghi care să aibă vârfurile constituite din cele trei obiecte de tip punct.

În aplicația de fizică, următoarea afirmație poate fi pusă sub forma unei reguli:

```

adunare_numere_complexe:
dacă
    c1 = a1 + i*b1 este un număr complex și

```

```

    c2 = a2 + i*b2 este un număr complex, diferit de
    primul și
    se dorește efectuarea sumei dintre c1 și c2
atunci
    introdu în bază informația că suma dintre c1 și c2
    este dată de numărul complex  $c = (a1 + a2) + i*(b1 + b2)$ .

```

Aplicarea ei, motivată de găsirea în bază a două numere complexe și de o aserțiune care dictează necesitatea efectuării sumei lor, ar trebui să adauge în baza de cunoștințe un nou fapt de tip număr complex și care să aibă calculate părțile reală și imaginară în maniera știută.

3.3. Variabile și șabloane în reguli

În general, în definițiile de reguli apar variabile. Domeniul lexical al unei variabile (zona din program în care semnificația variabilei este aceeași) este strict limitat la o regulă. Astfel, în definiția regulii din exemplul medical, apare firesc să notăm într-un anumit fel pacientul; în exemplul preluat din geometrie, numele punctelor vor fi date de variabile, iar în exemplul din fizică, atât numerele complexe cât și părțile lor reale și imaginare vor fi notate cu variabile.

Este clar că regula de diagnostic medical dată mai sus nu este specifică pacientului Ionescu, și nici unui alt pacient anume, ci este aplicabilă oricărui pacient. Acest “oricare pacient”, trebuie, pe de o parte, să apară în exprimarea regulii, pentru ca cele trei fapte ce se caută în bază să fie legate între ele prin informația comună asupra numelui pacientului (ar fi neplăcut ca gâtul țepăn al lui Ionescu să fie coroborat cu temperatura mare a lui Popescu și cu pierderile de cunoștință ale lui Georgescu pentru a trage concluzia că cineva are meningită). Pe de altă parte, acest “oricare pacient”, care a fost găsit în bază ca satisfăcând simultan condițiile părții stângi, este exact acela asupra căruia se va trage concluzia de meningită. Cu alte cuvinte, regula dată mai sus ar trebui să arate cam așa:

```

diagnostic_meningită:
dacă
    [pacient = X, mobilitate_gât = mică]
    [pacient = X, temperatură = mare]
    [pacient = X, pierderi_cunoștință = frecvente]
atunci
    introdu în bază
    [pacient = X, diagnostic_posibil = meningită]

```

X, în această exprimare, este o variabilă. Condiția $[pacient = X, mobilitate_gât = mică]$ din partea stângă a regulii se aseamănă cu faptul $[pacient = Ionescu, mobilitate_gât = mică]$ din bază dar nu este

identică cu el. Această condiție constituie un șablon. Suntem în cazul unui șablon cu variabile. El exprimă succint condiția ca în bază să existe un fapt care “să se potrivească” cu modelul indicat de șablon. Dacă un șablon nu conține variabile, el trebuie să fie identic cu un fapt din bază pentru ca partea corespunzătoare din condiția regulii să fie verificată.

Sistemul de control încearcă să confrunte partea de condiții a regulii cu faptele din bază. Regula se va considera “aplicabilă” dacă în bază se găsesc simultan trei fapte care să se potrivească cu cele trei condiții, cum sunt cele date mai sus asupra pacientului Ionescu. Dacă acest lucru se întâmplă, atunci un fapt nou este adăugat în bază, conform șablonului din partea de acțiuni a regulii. Variabila joacă acum rolul de căraș (transportor) de informații din partea stângă spre partea dreaptă.

Regula din aplicația de construcții geometrice, poate fi acum rescrisă utilizând un format al condițiilor apropiat de cel în care au fost definite faptele:

```
construcție_triunghi:
dacă
    [punct = A]
    [punct = B] și B ≠ A
    [punct = C] și C ≠ A și C ≠ B
atunci
    introdu în bază [triunghi = ABC]
```

unde A, B și C sunt variabile.

La fel, în aplicația de fizică, regula de adunare a două numere complexe poate fi exprimată într-o formă apropiată de următoarea:

```
regula_adunare_numere_complexe:
dacă
    [complex = c1, real = a1, imaginar = b1]
    [complex = c2, real = a2, imaginar = b2] și
    c1 ≠ c2
    se dorește efectuarea sumei dintre c1 și c2
atunci
    introdu în bază informația că suma dintre c1 și c2
    este dată de numărul complex [complex = c, real = a1+a2,
    imaginar = b1+b2]
```

iar simbolurile c1, c2, c, a1, a2, b1 și b2 sunt toate variabile.

3.4. Legări de variabile și instanțe de reguli

În general sunt trei situații în care se pot afla faptele din baza de cunoștințe față de condițiile din partea stângă a unei reguli: nu există o combinație de fapte din bază care să verifice condițiile, există exact un mod în care faptele din bază verifică condițiile sau există mai multe moduri în care acestea să le verifice. Procesul de verificare a părții de condiții a unei reguli în inteligența artificială este numit **confruntare de șabloane**.

Astfel, pentru exemplul de diagnostic medical de mai sus, presupunând o bază de fapte consistentă și care nu deține informații temporale (interesează, spre exemplu, datele asupra pacienților la momentul internării într-o secție a unui spital), vor exista atâtea posibilități de aplicare a regulii câți pacienți ce manifestă simultan simptomele menționate în regulă sunt înregistrați în bază.

În cazul exemplului de construcție geometrică, dacă baza de fapte cuprinde mai puțin de trei definiții de puncte, regula nu poate fi aplicată. Dacă baza include exact trei definiții de puncte atunci regula se poate aplica în nu mai puțin de șase moduri. Într-adevăr, presupunând punctele notate în bază M, N și P, atunci cu ele se pot forma triunghiurile MNP, MPN, NMP, NPM, PMN și PNM, în funcție de ordinea în care se consideră realizate identificările dintre numele punctelor geometrice din bază și notațiile punctelor în regulă, respectiv A, B și C.

Totodată, dacă în exemplul de adunare a numerelor complexe de mai sus, presupunem existența în bază a două definiții de numere complexe, să zicem:

```
[complex = z1, real = 200, imaginar = 150] și
[complex = z2, real = 300, imaginar = 150]
```

atunci vor exista două maniere de realizare a sumei, după cum c1 din definiția condițiilor regulii va fi identificat cu z1 și c2 cu z2 sau invers.

Așadar, este ca și cum o regulă ar fi multiplicată de către componenta de control în mai multe instanțe. Ele apar întotdeauna când în partea de condiții a unei reguli se regăsesc mai multe condiții care sunt exprimate în forme similare (șabloane structural identice³).

Practic o **instanță** a unei reguli R constă din numele regulii împreună cu faptele ce verifică condițiile regulii și cu legările variabilelor la valori. Aceste legări au loc la confruntarea părții de condiții a regulii cu o combinație de fapte din bază. Vom nota o instanță de regulă între paranteze unghiulare, în forma:

```
<nume_regulă; f1, ... fn; v1→a1, ... vk→ak>
```

³ Două șabloane sunt structural identice dacă ele sunt identice cu excepția numelor variabilelor.

unde prin f_1, \dots, f_n vom înțelege faptele ce verifică condițiile regulii, iar prin $v_i \rightarrow a_i$ vom înțelege legarea variabilei v_i la valoarea a_i ($i \in \{1, \dots, k\}$).

Astfel, considerând faptele de mai sus ce definesc simptomele pacientului Ionescu și regula `diagnostic_meningită`, sistemul de control va produce o unică instanță a acesteia:

```
<diagnostic_meningită, X→Ionescu>
```

În același mod, faptele ce descriu punctele A, B și C împreună cu regula `construcție_triunghi` provoacă instanțierile:

```
<construcție_triunghi, A→M, B→N, C→P>
<construcție_triunghi, A→M, B→P, C→N>
<construcție_triunghi, A→N, B→M, C→P>
<construcție_triunghi, A→N, B→P, C→M>
<construcție_triunghi, A→P, B→M, C→N>
<construcție_triunghi, A→P, B→N, C→M>
```

iar faptele ce definesc numerele complexe $z1$ și $z2$ împreună cu regula `adunare_numere_complexe` duc la apariția instanțelor:

```
<adunare_numere_complexe, c1→z1, a1→200, b1→150,
c2→z2, a2→300, b2→150>
<adunare_numere_complexe, c1→z2, a1→300, b1→150,
c2→z1, a2→200, b2→150>
```

În Capitolul 7 – *Primii pași într-un limbaj bazat pe reguli: CLIPS* din partea a treia a cărții se vor detalia acțiunile ce au loc în componenta de control la instanțierea regulilor.

3.5. Agenda

Agenda este structura de date care memorează la fiecare moment instanțele regulilor. Aceste instanțe sunt dispuse într-o listă, instanța de regulă aflată pe prima poziție fiind aceea ce va fi apoi utilizată, așa cum vom vedea în secțiunea următoare.

Există două criterii care dictează ordinea instanțelor regulilor din agendă. Primul este **prioritatea declarată a regulilor**, al doilea **strategia de control**. Urmând aceste două criterii, instanțele regulilor ce-și satisfac condițiile la un moment dat sunt întâi ordonate în agendă în ordinea descrescătoare a priorităților declarate, iar cele de priorități egale, în ordinea dată de strategia de control.

3.6. Motorul de inferențe

Componenta de control a unui sistem expert mai este numită și motor de inferențe pentru că, în cursul execuției, la fel ca într-un proces inferențial, sistemul este capabil să genereze fapte noi din cele cunoscute, aplicând reguli.

Un mecanism elementar de aprindere a regulilor lucrează conform următorului algoritm, care descrie cel mai simplu ciclu al unui motor de inferențe ca un motor în trei timpi:

- **faza de filtrare:** se determină mulțimea tuturor instanțelor de reguli filtrate (MIRF) corespunzătoare regulilor din baza de reguli (BR) care își pot satisface condițiile pe faptele din baza de fapte (BF). Dacă nici o regulă nu a putut fi filtrată, atunci motorul se oprește. Dacă există cel puțin o instanță de regulă filtrată, atunci se trece în faza următoare;

- **faza de selecție:** se selectează o instanță de regulă $R \in \text{MIRF}$. Dacă MIRF conține mai mult de o singură instanță, atunci selecția se realizează prin aplicarea uneia or a mai multor strategii de conflict (v. secțiunea rezoluția conflictelor), după care se trece în faza următoare;

- **faza de execuție:** se execută partea de acțiuni a regulii R, cu rezultat asupra bazei de fapte. Se revine în faza de filtrare.

Anumite *shell*-uri de sisteme expert recunosc o sub-fază a fazei de filtrare, numită *faza de restricție*, în care se determină, plecând de la baza de fapte BF și de la baza de reguli BR, submulțimile $BF_I \subseteq BF$, respectiv $BR_I \subseteq BR$, care, *a priori*, merită să participe la acest ciclu inferențial. Apoi MIRF este determinată ca mulțimea instanțelor regulilor aparținând BR_I care-și satisfac condițiile pe faptele din BF_I . O tehnică curentă pentru realizarea restricției constă a împărți cunoștințele disponibile în familii de reguli, care să fie apoi utilizate în funcție de contextul în care a ajuns raționamentul (de exemplu într-un sistem de urmărire a funcționării unui proces pot exista etapele *detecție*, *diagnostic* și *intervenție*) sau în funcție de tipul specific de problemă ce trebuie soluționată (de exemplu, într-o aplicație de diagnostic medical regulile ce țin de bolile infantile pot fi despărțite de cele ce analizează boli ale adulților, iar faptele ce țin de analiza sângelui pot fi separate de celelalte). *Shell*-urile evaluate de sisteme expert oferă inginerului bazei de cunoștințe posibilitatea de separare în clase a setului de reguli și, analog, a mulțimii de fapte. Cel mai adesea, pentru mai multe cicluri consecutive se folosesc reguli aparținând aceleiași submulțimi. În consecință, pentru a economisi timp, faza de restricție poate fi exterioară unui ciclu, ea indicând motorului să lucreze cu o secțiune sau alta a bazei de reguli, respectiv de fapte, pe o etapă anumită a derulării raționamentului.

Motoarele de inferență implementează o proprietate numită **refractabilitate**, care se manifestă în faza de filtrare: o regulă nu este filtrată mai mult de o singură dată pe un set anume de fapte. Fără această proprietate, sistemele expert ar fi

angrenate adesea în bucle triviale ce ar apărea ori de câte ori acțiunile părții drepte ale unei reguli nu ar produce modificări în bază.

Capitolul 4

Înlănțuirea regulilor în motoarele de inferență

4.1. Căutarea soluției în problemele de inteligență artificială

Problemele de inteligență artificială se aseamănă prin aceea că toate presupun existența unei stări inițiale cunoscute și a uneia sau mai multor stări finale precizate exact sau doar schițate prin condiții pe care ele ar trebui să le satisfacă, iar rezolvarea problemei constă în identificarea unui traseu între starea inițială și o stare finală. Uneori găsirea soluției înseamnă descifrarea unui drum care să unească o stare inițială de una finală, iar alteori numai descoperirea stării finale (v. Figura 6).

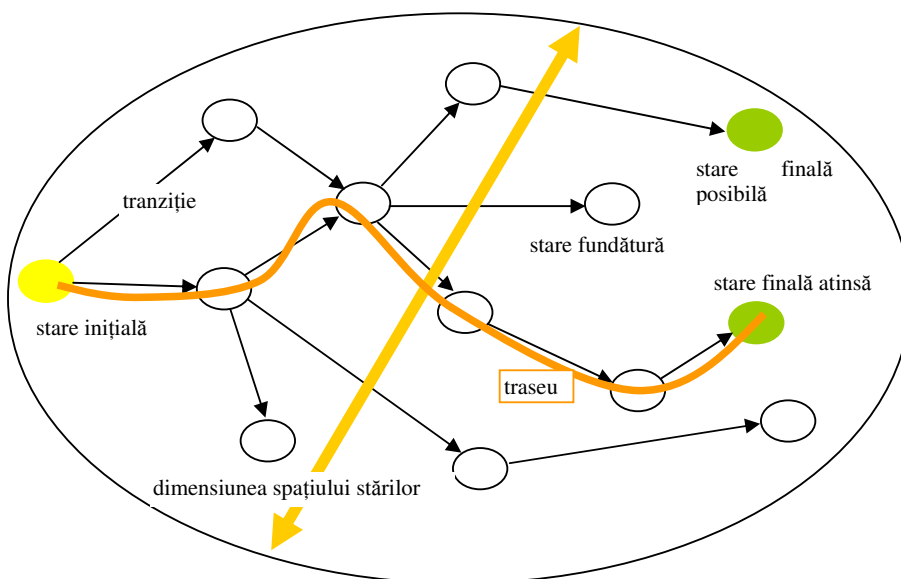


Figura 6: Spațiul stărilor în problemele de inteligență artificială

Acest lucru face ca o problemă de inteligență artificială să fie una de căutare într-un spațiu al stărilor. În funcție de dimensiunea acestui spațiu se utilizează

diferite metode pentru a eficientiza procesul de navigare. În limbajul domeniului inteligenței artificiale, aceste metode se mai numesc și **strategii**. Există mai multe tipuri de strategii, dar ele sunt clasificate în două mari clase: **irevocabile** și **tentative**. O strategie irevocabilă este una în care orice mișcare în spațiul stărilor este ireversibilă, în care nu există cale de întoarcere în caz de greșeală. O alegere o dată făcută rămâne definitivă, iar dacă într-un pas ulterior unei astfel de alegeri motorul ajunge într-un impas, el se va opri, așadar, fără a oferi posibilitatea de a mai fi explorate alte căi spre soluție. Dimpotrivă, o strategie tentativă este una șovăitoare, în care o mișcare ce se dovedește greșită poate fi îndreptată. Aplicarea unei strategii este necesară pentru a decide ce e de făcut în situațiile în care dintr-o stare anumită există mai multe căi de a trece într-o altă stare și în situațiile în care, deși nu s-a ajuns într-o stare finală, nu mai există nici o posibilitate de a face o altă tranziție.

Totodată, parcurgerea spațiului stărilor poate fi făcută plecând de la stările inițiale pentru a avansa către cele finale, și atunci avem de a face cu o **căutare** (sau **înlănțuire**) **înainte**, invers, plecând de la ceea ce se consideră a fi o stare finală pentru a regăsi starea inițială, și atunci avem de a face cu o **căutare** (sau **înlănțuire**) **înapoi**, sau combinând o căutare înainte cu una înapoi într-o **căutare** (sau **înlănțuire**) **mixtă**. În cele ce urmează vom discuta în amănunt tipul de înlănțuire a regulilor înainte și vom face o prezentare scurtă a celorlalte două tipuri de căutări.

Termenul de “înlănțuire” folosit în sintagmele de mai sus este utilizat în directă legătură cu aplicarea domeniului inteligenței artificiale la sistemele expert, unde tranziția între două stări înseamnă aplicarea unei reguli. Faptele din baza de cunoștințe configurează o stare a sistemului în orice moment din dezvoltarea inferențelor. Regulile bazei de reguli își verifică partea de condiții pe faptele din baza de cunoștințe, deci pe starea curentă. Rezultă un număr oarecare de instanțe de reguli, adică de reguli potențial aplicabile. Aplicarea lor ar configura “evantaiul” de stări în care se poate tranzita din starea curentă. În Figura 7 stările sunt notate ca noduri, starea din care se pleacă este numită **stare sursă**, iar starea în care se ajunge după aplicarea regulii este numită **stare destinație**. Este posibil ca instanțe diferite, aplicate stării sursă, să producă aceeași stare destinație, ceea ce înseamnă că tranzițiile între stări sunt caracterizate, în general, de mulțimi de instanțe. De aceea etichete ale arcelor în notația de graf pot fi liste de instanțe. Mulțimea etichetelor arcelor care pleacă din nodul stare curentă este, astfel, agenda.

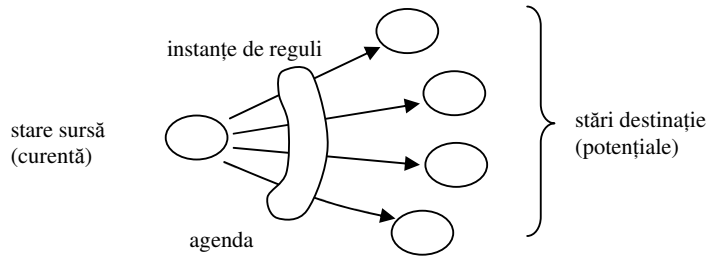


Figura 7: Instanțele regulilor aplicabile în starea curentă pot tranzita starea sistemului într-o mulțime de stări potențiale

4.2. Înlănțuirea înainte

Într-un motor cu înlănțuire înainte se pleacă cu fapte ce precizează ipoteze inițiale de lucru și se cere să se ajungă la o concluzie. Aceasta poate fi cunoscută, și atunci cerința este de a o verifica, sau necunoscută, și atunci cerința este de a o găsi. În general verificarea concluziilor este o caracteristică a motoarelor implantate pe sisteme de diagnostic sau simulare, pe când găsirea unei stări noi este o caracteristică a sistemelor expert de construcție. Într-un astfel de sistem, în principiu, nu se cunoaște *a priori* configurația spațiului stărilor, dar o parte a acestui spațiu este relevantă de rulare.

Pentru a exemplifica funcționarea unui motor în înlănțuire înainte, să considerăm următoarea colecție de reguli (exemplu adaptat):

Ex. 1

- R1: **dacă** A, **atunci** E
- R2: **dacă** B, E, **atunci** F
- R3: **dacă** C, E, **atunci** G
- R4: **dacă** B, D, **atunci** G
- R5: **dacă** F, G, **atunci** H
- R6: **dacă** I, **atunci** H

și următoarea colecție de fapte inițiale: {A, B, C, D, I}. Cerința este de a proba faptul H.

Existența unui nume de fapt în partea de condiții a unei reguli (partea stângă) semnifică cerința ca acel fapt să se găsească în bază. Existența lui în partea dreaptă înseamnă că, la aprinderea regulii, acel fapt este introdus în bază.

Putem simula activitatea sistemului de control cu înlănțuire înainte pe acest exemplu utilizând un graf ȘI-SAU⁴ care să modeleze baza de reguli, ca în Figura 8:

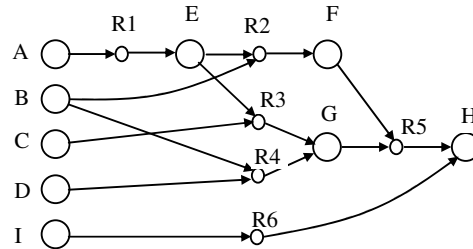


Figura 8: Graful ȘI-SAU atașat bazei de reguli a exemplului Ex. 1

În această figură am folosit următoarele convenții de notație: cu cerceulețe mari am notat faptele, cu cerceulețe mici – regulile, săgețile care înțeapă o regulă semnifică un ȘI logic pentru realizarea părții de condiții a regulii (pentru că o regulă poate fi aplicată doar dacă toate condițiile date de partea ei stângă sunt realizate) iar săgețile care înțeapă un fapt semnifică un SAU logic pentru introducerea faptului în bază (pentru că un fapt poate fi introdus în bază de oricare dintre regulile care îl specifică în partea dreaptă). Un fapt odată introdus în bază se consideră a fi adevărat. Cu aceste convenții nodurile-regulă sunt noduri ȘI iar nodurile-fapte sunt noduri SAU. În secvența de grafuri din Figura 9 se indică o posibilă ordine în aplicarea regulilor. Pe fiecare rând este desenat graful ȘI-SAU, care oglindește situațiile din faza de selecție, urmat de situația existentă imediat după execuție. Primul graf arată așadar situația în faza de selecție inițială. Faptele aflate în bază sunt prezentate într-o culoare închisă, regulile potențial active (filtrate) în gri deschis, iar cele efectiv aprinse (selectate și executate) în negru.

⁴ Grafurile ȘI-SAU, grafuri orientate folosite în logică pentru demonstrarea valorilor de adevăr ale formulelor logice, sunt formate din noduri ȘI și noduri SAU. Un nod ȘI este adevărat dacă toate nodurile din care pleacă arcele ce-l înțeapă sunt adevărate. Un nod SAU este adevărat dacă măcar unul dintre nodurile din care pleacă arcele ce-l înțeapă este adevărat.

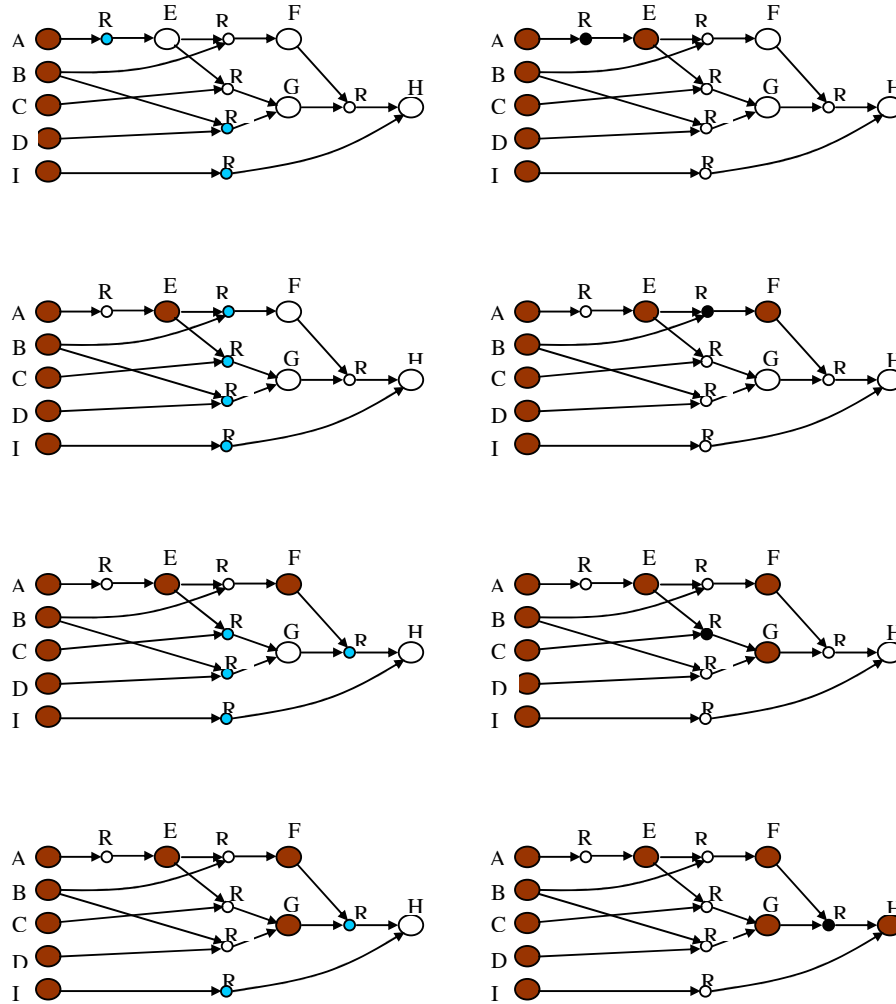


Figura 9: O secvență posibilă de aprinderi de reguli și includeri de fapte în bază pentru exemplul Ex.1

Trebuie evitată confuzia de a asimila reprezentarea sub formă de graf a spațiului stărilor, prezentat la începutul acestui capitol, cu graful ȘI-SAUI pe care l-am utilizat în notarea bazei de cunoștințe inițiale și în simularea aprinderii regulilor. Pentru a clarifica total lucrurile, să elaborăm o parte a spațiului stărilor pentru exemplul de mai sus, relevant de secvența de aprinderi de reguli indicată în Figura 9 (vezi Figura 10). O stare este dată de mulțimea faptelor din bază, deci

starea inițială va fi {A, B, C, D, I}. Secvența de reguli aplicate este R1, R2, R3, R5, indicată în figură prin săgețile îngroșate. Stările finale sunt marcate prin cercuri duble și sunt cele ce conțin faptul H, ce trebuia dovedit. Figura prezintă o strategie neeficientă, pentru că starea finală, deși posibil de atins după un singur pas (ciclu), după doi pași, sau după trei pași, a fost atinsă numai după patru cicluri ai motorului.

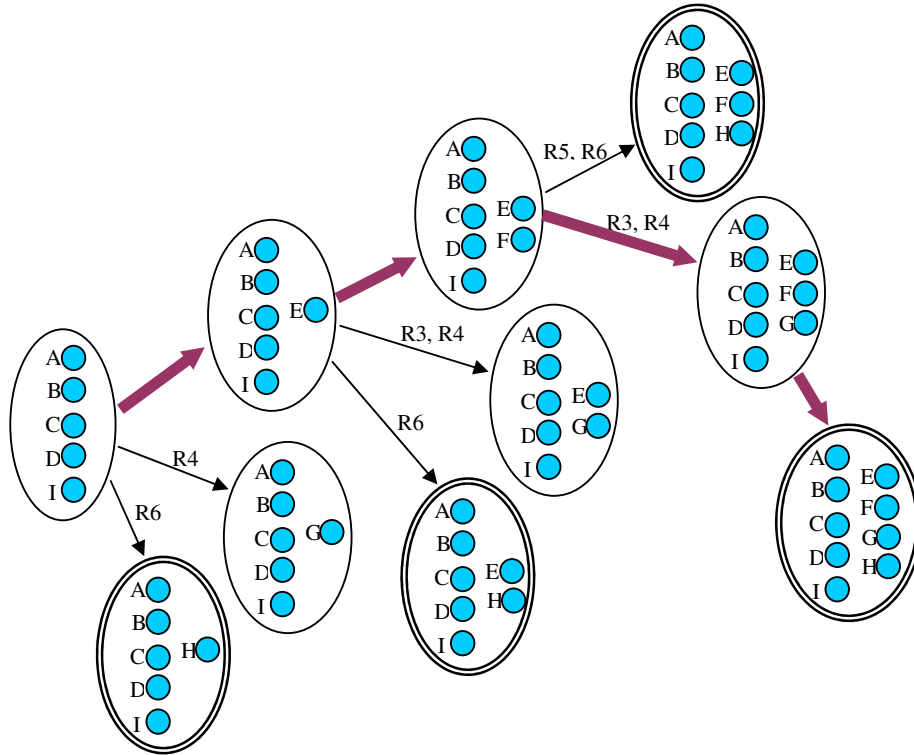


Figura 10: O porțiune a spațiului stărilor și realizarea tranzițiilor pentru exemplul de mai sus

În cele ce urmează vom investiga câteva proprietăți ale motoarelor de inferență.

4.3. Despre prezumția de lume deschisă/închisă

Până acum am asociat un fapt din baza de cunoștințe unui ansamblu de proprietăți definite asupra unui obiect. Bazat pe reprezentarea proprietăților obiectelor de acest fel, am asociat satisfacerea unei condiții din partea stângă a unei reguli cu găsirea unui fapt în bază care să corespundă acelei descrieri de condiții.

Dar cum s-ar putea face ca o regulă să verifice, dimpotrivă, o condiție negată, adică inexistența unui obiect de un anumit fel în bază? Dacă afirmarea proprietăților unor obiecte este o îndeletnicire rezonabilă și finită în timp și spațiu, afirmarea explicită a proprietăților pe care nu le au obiectele este sortită eșecului. Astfel, dacă ar trebui să definesc proprietățile scaunului pe care stau acum, când scriu aceste rânduri, ar trebui să spun ceva de genul: scaun cu patru picioare, din lemn sculptat, de culoare maro închis, cu spetează înaltă, îmbrăcat în pluș roșu și prevăzut cu brațe. Imaginați-vă însă ce ar însemna să definesc explicit proprietățile pe care scaunul meu nu le are. E lesne de înțeles că ar trebui să încep cam așa: nu este din făcut din coajă de ou și nici nu e prevăzut cu pene, nu are coadă sau dinți, nu are roți, bujii și faruri, nu are pagini ca o carte, nu cântă, nu are bani în bancă, și nu are nevoie să fie udat periodic. Adică, dacă cineva m-ar întreba asupra existenței acestor proprietăți la scaunul meu aș răspunde negativ la ele, dar cine știe câte întrebări ar mai putea cineva să irosească relativ la nevinovatul obiect pe care stau?

Analog am putea să ne întrebăm cum definim într-o bază de cunoștințe obiectele care populează un micro-univers: asertându-le pe cele care fac parte din el sau infirmându-le pe cele care nu fac parte? Într-adevăr, uitându-mă în jur văd că obiectele de pe masa mea de lucru sunt: un calculator, un telefon fix, unul mobil, o ceșcuță cu cafea, o vază de flori, un ceas, un toc de ochelari, 18 cărți, o mapă, hârtii, două dischete, un deschizător de plicuri, un CD, două pixuri și o cutie cu agrafe. În mod clar nu se află acolo: doi elefanți, nici măcar unul, un Audi automatic, un microscop electronic, un tablou de Chagall, o poartă de biserică, un vierme de mătase pistruiat și o pisică siameză răgușită și cu ochii albaștri.

Cu alte cuvinte, ce putem face dacă acțiunea de a cumpăra o pisică siameză cu ochi albaștri ar trebui să fie condiționată de neexistența ei la mine în casă? Trebuie ca baza mea de date să o menționeze explicit ca nefiind în posesia mea, așa cum se întâmplă în cazul tuturor regulilor discutate până acum, ce se aprind numai dacă un obiect care se potrivește peste condiția specificată în regulă este găsit în bază? Dacă aș fi pregătit pentru această întrebare, mi-aș prevedea un fapt care să nege explicit existența pisicii amintite la mine în casă, dar, adoptând această politică, câte întrebări trebuie anticipate înainte de a le fi auzit?

Se pare așadar că putem conveni asupra a două opțiuni de a satisface un șablon negat din corpul unei reguli: să dispunem ca el să fie satisfăcut numai dacă un fapt care să afirme neexistența unui obiect cu anumite trăsături sau neexistența anumitor trăsături ale unui obiect este inclus în bază, sau, dimpotrivă, să dispunem ca un șablon negat să fie satisfăcut numai dacă un fapt cu proprietățile cerute în șablon nu este găsit în bază. În primul caz spunem că lucrăm în prezumția de lume deschisă, în cel de al doilea – că lucrăm în prezumția de lume închisă. Prezumția de lume deschisă este echivalentă cu presupunerea unui univers infinit, deci un univers în care atât existența cât și neexistența unui obiect sau a unei proprietăți a unui obiect trebuie declarate explicit pentru a fi siguri că el există sau nu acolo. Dimpotrivă, prezumția de lume închisă este echivalentă presupunerii unui univers

finit, ce poate fi descris exhaustiv, astfel încât dacă un obiect nu este declarat înseamnă că el nu face parte din univers.

Definiție: Spunem că un motor de inferențe lucrează în **prezumția de lume deschisă** dacă un șablon negat $\text{not}(P)$, este evaluat la *true* numai atunci când în baza de fapte **există declarat un fapt $\text{not}(P')$ peste care P să se potrivească**.

Definiție: Spunem că un motor de inferențe lucrează în **prezumția de lume închisă** dacă orice șablon aflat în partea de condiții a unei reguli, de forma $\text{not}(P)$, este evaluat la *true* atunci când în baza de fapte **nu există nici un fapt P' peste care P să se potrivească**.

4.4. Despre monotonie

Se spune despre un sistem expert că funcționează **monoton**, sau că este monoton, dacă:

- nici o cunoștință (fapt stabilit sau regulă) nu poate fi retrasă din bază și
- nici o cunoștință adăugată la bază nu introduce contradicții.

Dimpotrivă un sistem poate să funcționeze și **nemonoton** dacă poate suprima definitiv sau inhiba provizoriu cunoștințe. Monotonia este o caracteristică a bazei de cunoștințe, mai precis a modului în care sunt proiectate regulile.

Să presupunem cazul unui motor cu înlănțuire înainte și care lucrează în prezumția de lume deschisă pe o bază monotonă. Cum, de la un ciclu la următorul, numărul faptelor dovedite (afirmative sau negative) crește, rezultă că și numărul regulilor posibil de aplicat de la un ciclu la următorul, după faza de filtrare, crește sau, cel puțin, rămâne constant. Atunci, dacă într-un ciclu n avem mulțimea $MIRF^n$ de instanțe de reguli filtrate, în ciclul $n+1$ vom avea mulțimea $MIRF^{n+1} \supseteq MIRF^n$, deci aceleași posibilități de aplicare a regulilor care au funcționat la pasul n se regăsesc și la pasul $n+1$, eventual mai multe. De aceea, apariția unui eșec într-un ciclu k nu se poate datora decât epuizării tuturor instanțelor de reguli din $MIRF^k$ (care include $MIRF^{k-1}$ ș.a.m.d.) și deci nu ne putem aștepta ca revenirea într-o stare anterioară, prin *backtracking*, să deschidă posibilități noi.

Aceste considerente justifică următoarea **proprietate**: dacă, pentru un sistem de reguli monoton, o problemă are o soluție, atunci un sistem care funcționează în înlănțuire înainte în prezumția de lume deschisă o poate găsi lucrând în regim irevocabil (v. Figura 11).

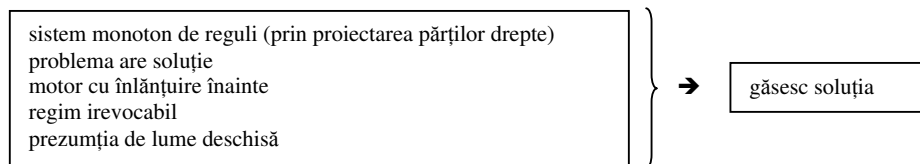


Figura 11: Asigurarea soluției în premisa de lume deschisă

O altă ordine de aplicare a regulilor în acest caz poate însemna o dinamică diferită de adăugare a faptelor în bază. Deși, în principiu, în cursul rulării tot mai multe reguli devin potențial active, terminarea procesului apare datorită epuizării instanțelor de reguli utilizabile prin restricția de unică filtrare a regulilor pe o configurație anume de fapte (proprietatea de refractabilitate). Deci, dacă există, mai devreme ori mai târziu o soluție va fi găsită.

Restricția privitoare la prezumția de lume deschisă este importantă pentru că semnificația șabloanelor negate în părțile de condiții ale regulilor este diferită în cele două prezumții. Astfel, pentru un motor care lucrează în prezumția de lume închisă, să presupunem existența unei reguli R care utilizează în partea de condiții un șablon negat $not(P)$. Aplicând definiția prezumției de lume închisă, dacă la un pas k o instanță a lui R făcea parte din $MIRF^k$ atunci înseamnă că nici un fapt P' peste care șablonul P se potrivește nu fusese introdus în bază până în acest moment. Datorită monotoniei sistemului este posibil însă ca un astfel de fapt să apară la un pas k' ulterior lui k , ceea ce va avea ca efect eliminarea lui R din $MIRF^k$. Ca urmare, pentru un motor care funcționează în prezumția de lume închisă, o condiție de suficiență a găsirii soluției trebuie enunțată cu o restricție suplimentară.

Proprietate: dacă, pentru un sistem monoton de reguli în care regulile nu conțin șabloane negate, o ipostază de problemă are o soluție, atunci un sistem care funcționează în înlănțuire înainte în prezumția de lume închisă o poate găsi lucrând în regim irevocabil.

Această concluzie este importantă pentru proiectarea sistemelor expert. Un shell de sisteme expert care funcționează ca motor cu înlănțuire înainte, în regim irevocabil și utilizând prezumția de lume închisă, este CLIPS, motorul de sistem expert pe care îl vom utiliza în restul cărții pentru experimentele noastre. Proprietatea de mai sus ne asigură de faptul că, proiectând regulile astfel încât acestea să fie monotone și să nu conțină șabloane negate, putem să garantăm găsirea soluției în problemele care au soluții (vezi Figura 12).

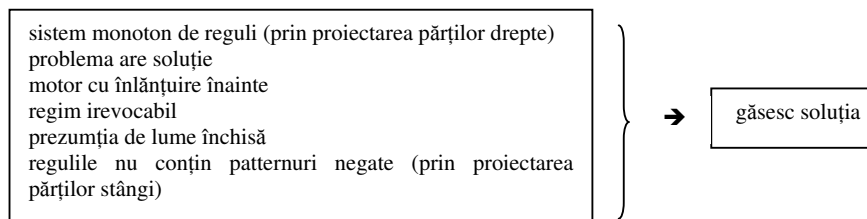


Figura 12: Restricții de proiectare a sistemelor expert pentru asigurarea soluției în premisa de lume închisă

4.5. Despre fapte negate și reguli cu șabloane negate

În premisa de lume deschisă, un fapt negat, fie el *not L*, care apare în partea de condiții a unei reguli, pentru a putea fi validat trebuie să fi fost asertat ca negat în bază. Ca urmare, pentru că necesități de consistență a bazei interzic prezența simultană atât a lui *L* cât și a lui *not L*, faptul *not L* ar putea fi înlocuit pur și simplu cu un fapt ce nu mai există, nenegat, $L' = \text{not } L$. Tragem de aici concluzia că faptele negate și, echivalent, condițiile negate din reguli, pot fi eliminate complet dintr-un sistem care lucrează în premisa de lume deschisă.

Nu se poate spune același lucru însă atunci când se lucrează în premisa de lume închisă. Aici, pentru ca o condiție negată să fie validată într-o regulă nu e necesar ca faptul negat să fie declarat în bază, ci e suficient ca faptul nenegat să nu apară. Așadar nici în modelul de lume închisă faptele negate nu-și au rostul.

Cele două raționamente de mai sus duc la concluzia că putem foarte bine să lucrăm numai cu fapte afirmative. De altfel un limbaj precum CLIPS nu acceptă declararea de fapte negate în baza de cunoștințe.

Dar care este atunci utilitatea negației în partea de condiții a regulilor? Să analizăm mai întâi ce se întâmplă cu un sistem monoton care are reguli ce conțin șabloane negate. Cum într-o bază de fapte monotonă faptele nu dispar niciodată, o regulă care conține șabloane negate în partea de premise are următoarea comportare pe parcursul procesului de inferență:

- dacă faptul pe care îl neagă premisa există inițial în bază, atunci regula nu este filtrată în ciclul I și nu va putea fi filtrată în nici un moment ulterior (pentru că acel fapt nu va dispărea și nici nu va putea fi negat). Acest lucru este valabil în ambele premise de lume;
- dacă însă faptul pe care îl neagă premisa nu există inițial (sau este negat inițial în premisa de lume deschisă), atunci regula este potențial activă inițial dar poate ulterior să dispară din agendă (o dată cu apariția faptului în baza de cunoștințe).

Comportamentul sistemului este așadar dependent de contextul în care se află motorul ceea ce poate fi exploatat de programator în proiectarea regulilor.

După cum am văzut, în anumite condiții e suficient să avem un motor care să funcționeze în regim irevocabil pe o bază de reguli monotone pentru ca să garantăm găsirea soluției atunci când ea există. În practică însă, cele mai frecvente sisteme sunt cele nemonotone care funcționează în prezumția de lume închisă. Retragera faptelor din bază este o operație des întâlnită. Din păcate utilizarea unui motor irevocabil nu mai garantează soluția la sistemele nemonotone, după cum nu o garantează nici la cele monotone în care regulile au șabloane negate.

O alternativă o oferă regimul tentativ, de care ne vom ocupa în capitoul următor.

Capitolul 5

Regimul de lucru tentativ

Un **regim tentativ** este acela în care, dacă într-un ciclu n se ajunge într-un punct mort (mulțimea *MIRF* vidă), deci nu se mai poate realiza faza de execuție, în ciclul de rang $n+1$ faza de restricție este sărită iar faza de filtrare restabilește direct mulțimea *MIRF* a regulilor din ciclul $n-1$, din care a fost îndepărtată regula R a cărei aplicare în ciclul $n-1$ a dus la eșec în ciclul n . Întoarcerea la contextul de lucru al ciclului anterior celui în care a avut loc blocajul presupune de asemenea restabilirea bazei de fapte caracteristice acelui moment. Procedura este aplicată recursiv de fiecare dată când apare un blocaj. Acest procedeu de întoarcere înapoi (*backtracking*) poate fi aplicat și în situațiile în care, o dată o soluție găsită, se dorește căutarea și a altor soluții.

Pentru a exemplifica funcționarea în cele două regimuri să considerăm mai întâi un set de reguli și o configurație a bazei de fapte inițiale, care duce la obținerea unei soluții într-un regim irevocabil (exemplu adaptat).

Ex. 2

R1: **dacă** A, **atunci** B
R2: **dacă** B, C, L, **atunci** D
R3: **dacă** D, E, **atunci** F
R4: **dacă** F, G, **atunci** H
R5: **dacă** D, I, **atunci** J
R6: **dacă** J, K, **atunci** H
R7: **dacă** B, L, **atunci** H

Baza de fapte inițiale este {A, I, C, L} iar scopul este H.

Tabela următoarea arată o posibilă funcționare a motorului pe această bază de cunoștințe.

Tabela 1

Starea	Baza de fapte înainte de aplicarea regulii	Reguli filtrate	Regula aplicată	Fapte adăugate	Baza de fapte după aplicarea regulii
inițială:1	A,I,C,L	R1	R1	B	A,I,C,L,B
2	A,I,C,L,B	R2,R7	R2	D	A,I,C,L,B,D
3	A,I,C,L,B,D	R5,R7	R5	J	A,I,C,L,B,D,J
4	A,I,C,L,B,D,J	R7	R7	H	A,I,C,L,B,D,J,H
5:succes	A,I,C,L,B,D,J,H	-			

O simplă modificare a unei reguli face acest sistem nemonoton. Regula pe care o vom modifica este R2. Să presupunem că ea devine:

R2: **dacă** B, C, L, **atunci** D, -L

în care -L are aici semnificația de retragere a faptului L.

După cum vedem din tabela de mai jos, aplicarea aceluiași motor pe baza de cunoștințe astfel modificată poate duce la un eșec:

Tabela 2

Starea	Baza de fapte înainte de aplicarea regulii	Reguli filtrate	Regula aplicată	Fapte adăugate	Fapte retrase	Baza de fapte după aplicarea regulii
inițială:1	A,I,C,L	R1	R1	B	-	A,I,C,L,B
2	A,I,C,L,B	R2,R7	R2	D	L	A,I,C,B,D
3	A,I,C,B,D	R5	R5	J	-	A,I,C,B,D,J
4:eșec	A,I,C,B,D,J	-	-	-	-	-

O soluție însă există, și ea ar fi obținută dacă în starea 2 s-ar prefera regula R7 în locul regulii R2. După cum știm, preferarea unei reguli din agendă în favoarea alteia este de competența strategiei de control. Numai că preferarea unei strategii în locul alteia poate să dea rezultate pentru anumite probleme și să fie inefectivă pentru altele. Numai un motor tentativ, în momentul epuizării agendei, poate reface condițiile de aplicare a unei reguli ce nu a fost aleasă într-un pas anterior, rejucând practic o carte și deschizând astfel o nouă cale spre succes. Iată ce s-ar întâmpla dacă am lăsa să ruleze un motor tentativ pe ultima bază de cunoștințe de mai sus:

Tabela 3

Starea	Baza de fapte înainte de aplicarea regulii	Reguli filtrate	Regula aplicată	Fapte adăugate	Fapte retrase	Baza de fapte după aplicarea regulii
inițială: 1	A,I,C,L	R1	R1	B	-	A,I,C,L,B
2	A,I,C,L,B	R2,R7	R2	D	L	A,I,C,B,D
3	A,I,C,B,D	R5	R5	J	-	A,I,C,B,D,J
4:eșec	A,I,C,B,D,J	-	-	-	-	-
revenire în starea 2	A,I,C,L,B	R7	R7	H	-	A,I,C,L,B,H
5:succes	A,I,C,L,B,H	-				

5.1. Simularea unui motor tentativ printr-un *shell* de motor irevocabil

Cu toată fragilitatea lor aparentă, *shell*-urile de sisteme expert ce implementează regimuri de lucru irevocabile sunt foarte răspândite (exemple sunt OPS5 [8], [13], CLIPS [16]). Din fericire, un comportament tentativ poate fi realizat pe o mașină irevocabilă pur și simplu prin proiectarea regulilor. În acest caz, practic se construiește un meta-sistem expert aflat deasupra sistemului expert dedicat domeniului avut în vedere. Faptele cu care lucrează meta-sistemul sunt regulile din domeniul aplicației.

În cele ce urmează vom proiecta meta-regulile unui sistem a cărui menire va fi să facă să funcționeze pe un motor irevocabil baza de cunoștințe din exemplul Ex. 2.

Vom considera următoarele tipuri generale de fapte (structuri de fapte⁵):

fapte care păstrează regulile domeniului (pentru că, așa cum spuneam, regulile domeniului trebuie privite acum drept fapte ale meta-sistemului), de forma:

```
(regula <eticheta> <prioritate> <listaPremise>
<listaActiuni>)
```

în care <eticheta> identifică unic o regulă, <prioritate> este un indicator al priorității regulii, iar <listaPremise> și <listaActiuni> păstrează premisele, deci partea stângă, și respectiv acțiunile, adică partea dreaptă, a regulii. Despre prioritatea regulilor, ca mijloc de a controla selecția, vom vorbi în mai multe rânduri în capitolele următoare.

⁵ Termenul englezesc: *templates*.

- stiva listelor de reguli deja utilizate este dată de un fapt de forma:

```
(stivaReguliAplicate <listaReguli>*)
```

unde <listaReguli> este o intrare în stivă și reprezintă o listă a etichetelor regulilor ce au fost deja aplicate, iar steluța semnifică, aici ca și mai departe, repetarea acestei structuri de zero sau mai multe ori. În această reprezentare, ca și în cele ce urmează, vom presupune că vârful stivei este prima listă de după simbolul care dă numele stivei, iar baza stivei este lista din extremitatea dreaptă a faptului. O ipostază a acestei stive, după al treilea ciclu al motorului care funcționează pe exemplul de mai sus, este:

```
(stivaReguliAplicate (R1 R2 R5) (R1 R2) (R1))
```

cu semnificația: la primul ciclu s-a aplicat regula R1, după al doilea erau aplicate R1 și R2, iar după al treilea – R1, R2 și R5.

- stiva listelor de reguli filtrate, dar neaplicate, este dată de un fapt de forma:

```
(stivaReguliDeAplicat <listaReguli>*)
```

unde <listaReguli> este o listă a etichetelor regulilor ce mai pot fi încă aplicate într-o stare dată. Pentru exemplificare, o ipostază a acestei stive, pe exemplul dat mai sus, după același al treilea ciclu al motorului, este:

```
(stivaReguliDeAplicat () (R7) ())
```

înțelegând prin aceasta că după primul pas nu mai rămăsese de aplicat nici o regulă dintre regulile filtrate, după al doilea pas mai rămăsese ca posibil de aplicat regula R7, iar după ciclul al treilea, nici o regulă. După cum se remarcă, presupunem că funcționează strategia “refractabilității”, ceea ce, simplificat, pentru cazul nostru, înseamnă că o regulă o dată aplicată nu mai poate fi filtrată.

- un fapt păstrează stiva configurațiilor bazei de fapte la momente anterioare aplicării regulilor, de forma:

```
(stivaBazelorDeFapte <listaFapte>*)
```

în care <listaFapte> este o listă a faptelor din bază înainte de derularea unui ciclu. O ipostază a acestei stive după același al treilea ciclu al motorului, pe exemplul de mai sus, este:

```
(stivaBazelorDeFapte (A I C B D) (A I C L B) (A I C L))
```

În orice moment în care trebuie să aibă loc o revenire, stiva regulilor de aplicat este memoria pe baza căreia se poate decide o întoarcere înapoi prin alegerea unei alte reguli în locul celei deja selectate la un pas anterior și care a dus motorul într-un impas. Prin urmare, dacă o intrare în această stivă este vidă, înseamnă că o întoarcere în starea corespunzătoare ei este inutilă întrucât nici o altă alegere nu mai este posibilă. Putem deci elimina intrările vide din stiva regulilor de aplicat fără a pierde nimic din coerență. Condiția este să păstrăm sincronismul sistemului de stive, pentru că altfel revenirea asupra alegerii unei reguli s-ar face într-un alt context decât cel avut în vedere. Cum cele trei stive trebuie să fie sincrone, deci să aibă în orice moment același număr de intrări (fie el, cândva, k), datorită dispariției intrărilor vide din stiva regulilor de aplicat, conform *simplificării* sugerate mai sus, numărul de cicluri pe care motorul le-a executat, în acest caz, va fi $\geq k$. Atunci, presupunând că intrarea a i -a din stiva de reguli de aplicat corespunde ciclului j , $j \geq i$, va trebui ca intrării a i -a din stiva de fapte să-i corespundă, de asemenea, configurația bazei de fapte de imediat înainte de ciclul j al motorului, iar intrarea a i -a a stivei de reguli aplicate să semnifice lista regulilor aplicate în aceeași situație. Această observație avertizează deci să avem grijă ca atunci când o intrare din stiva regulilor de aplicat dispăre, intrările corespunzătoare din stiva de fapte și din cea a regulilor aplicate să dispară de asemenea.

Aplicând această observație la exemplul tratat mai sus, dacă înainte de simplificare, la momentul de dinaintea ciclului al treilea al motorului, configurațiile celor trei stive erau:

```
(stivaReguliDeAplicat () (R7) ())
(stivaReguliAplicate (R1 R2 R5) (R1 R2) (R1))
(stivaBazelorDeFapte (A I C B D) (A I C L B) (A I C L))
```

după simplificare, acestea trebuie să arate astfel:

```
(stivaReguliDeAplicat (R7))
(stivaReguliAplicate (R1 R2))
(stivaBazelorDeFapte (A I C L B))
```

Pe Figura 14 se constată că această simplificare, efectuată înainte de ciclul al patrulea al motorului, semnifică o revenire direct în starea 2 din starea 4, fără a mai vizita starea 3, de unde nici o altă alegere n-ar mai fi fost posibilă.

În continuare, o seamă de fapte vor ține starea curentă a motorului:

- lista regulilor filtrate în ciclul curent se memorează într-un fapt de forma:

```
(reguliFiltrate <etichetaRegula>*)
```

unde <etichetaRegula> este un index de regulă.

- baza de fapte curentă este păstrată de un fapt de forma:

```
(bazaDeFapteCurenta <fapt>*)
```

unde <fapt> este un nume de fapt. Pe această bază de fapte are loc faza de filtrare. Baza de fapte este actualizată atât în timpul de execuție ai motorului, cât și în urma revenirilor.

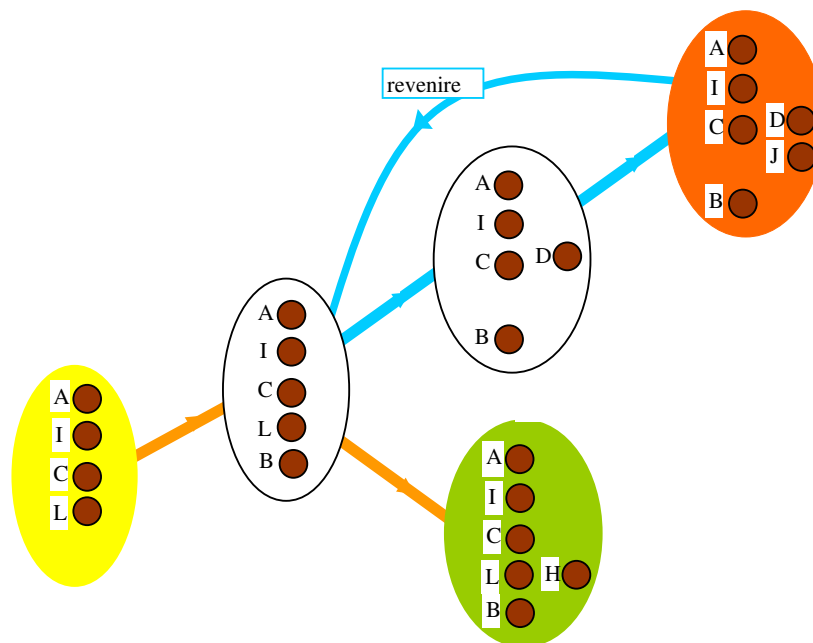


Figura 14: Scurt-circuitarea revenirii dintr-o stare de blocare în prima stare anterioară în care mai există o alternativă neexplorată

- lista regulilor deja aplicate la momentul curent este păstrată într-un fapt de forma:

```
(reguliAplicate <etichetaRegula>*)
```

- scopul căutat îl vom păstra într-un fapt de forma:

```
(scop <fapt>)
```

• mai avem nevoie de un fapt care precizează faza în derularea unui ciclu, de forma:

```
(faza <numeFaza>)
```

Configurația inițială într-o rulare a motorului tentativ este dată de:

- o mulțime de fapte (regula ...),
- un fapt (stivaReguliDeAplicat),
- un fapt (stivaReguliAplicate),
- un fapt (stivaBazelorDeFapte),
- un fapt (reguliFiltrate),
- un fapt (reguliAplicate),
- un fapt (bazaDeFapteCurenta <bazaDeFapteInitiala>),

unde <bazaDeFapteInitiala> reprezintă baza de fapte de plecare,

- un fapt (scop <faptDeDemonstrat>), unde <faptDeDemonstrat> este faptul de demonstrat,
- un fapt (faza filtrare), semnificând faptul că prima fază a motorului este filtrarea.

Descriem mai departe meta-regulile motorului tentativ. Vom începe prin a defini regulile responsabile cu terminarea procesului. Terminarea se poate face cu succes dacă în baza de fapte curentă se regăsește faptul definit drept scop, sau cu eșec dacă, la sfârșitul fazei de filtrare, nici o regulă nu a fost selectată iar stiva de reguli de aplicat e goală:

```
MT0: "Terminare cu succes, dacă în baza de date curentă se află faptul scop"
dacă      (faza filtrare) și
(scop UN-FAPT) și
(bazaDeFapteCurenta NISTE-FAPTE) și
UN-FAPT ∈ NISTE-FAPTE
atunci    elimină (faza filtrare)
anunță succes
```

Condiția de terminare este căutată imediat după execuție, deci după actualizarea bazei de date curente, adică în faza de filtrare. Terminarea se produce prin retragerea faptului (faza ...) care este referit de oricare meta-regulă. Prioritatea meta-regulii MT0 trebuie să fie cea mai mare din faza de filtrare pentru ca terminarea cu succes să se realizeze imediat după ce faptul scop a fost găsit în faza anterioară de execuție.

MT1: "Terminare cu eșec, dacă nici o regulă nu mai poate fi selectată și stiva de reguli rămase de aplicat e epuizată"

dacă (faza filtrare) **și**
 (reguliFiltrate) **și**
 (stivaReguliDeAplicat) **și**
 nici una din regulile fazei de filtrare, MT0 sau MT2, nu pot fi aplicate
atunci elimină (faza filtrare)
anunță eșec

La fel ca mai sus, terminarea se produce datorită retragerii faptului (faza ...) care este referit de oricare meta-regulă. Prioritatea meta-regulii MT1 trebuie să fie cea mai mică între regulile ce pot fi aplicate în faza de filtrare, pentru ca ea să fie luată în considerare doar după încheierea filtrării și înainte de tranziția în faza de selecție.

Următoarea meta-regulă descrie faza de filtrare: dintre faptele (regula ...) se selectează toate regulile aplicabile și care nu au mai fost aplicate, adică nu se regăsesc printre regulile menționate în lista de reguli aplicate (reguliAplicate ...). Condiția de aplicabilitate a unei reguli este ca faptele din partea de condiții a regulii să se regăsească între cele ale bazei de fapte (bazaDeFapteCurenta ...):

MT2: “Faza de filtrare”

dacă (faza filtrare) **și**
 (reguliFiltrate NISTE-REGULI-FILTRATE) **și**
 (bazaDeFapteCurenta NISTE-FAPTE) **și**
 (reguliAplicate NISTE-REGULI-APPLICATE) **și**
 (regula R PRI PREMISE ACTIUNI) **și**
 $R \notin \text{NISTE-REGULI-FILTRATE}$ **și**
 $R \notin \text{NISTE-REGULI-APPLICATE}$ **și**
 $\text{PREMISE} \subseteq \text{NISTE-FAPTE}$ **și**
 MT0 nu poate fi aplicată
atunci elimină (reguliFiltrate NISTE-REGULI-FILTRATE) **și**
adaugă (reguliFiltrate R NISTE-REGULI-FILTRATE)

Regula MT2 se aplică de atâtea ori câte reguli pot fi filtrate. La terminarea filtrării, motorul trece în faza de selecție:

MT3: “Tranziția în faza de selecție”

dacă (faza filtrare) **și**
 nici una din regulile MT0, MT1, sau MT2 nu (mai) poate fi aplicată
atunci elimină (faza filtrare) **și**
adaugă (faza selecție)

Meta-regula care urmează descrie un proces elementar de selecție, și anume unul bazat exclusiv pe prioritatea declarată a regulilor. În cazul mai multor reguli

de prioritate maximă, ultima filtrată este cea selectată. Meta-regula MT4, în esență, procedează la o ordonare a etichetelor de reguli filtrate în ordine descrescătoare:

```
MT4: "Faza de selecție"
dacă      (faza selecție) și
(reguliFiltrate ... R1 R2 ...) și
(regula R1 PRI1 ...) și
(regula R2 PRI2 ...) și
PRI2 > PRI1
atunci elimină (reguliFiltrate ... R1 R2 ...) și
adaugă (reguliFiltrate ... R2 R1 ...)
```

Dacă în urma filtrării rezultă cel puțin o regulă, când ordonarea lor ia sfârșit se trece în faza de execuție:

```
MT5: "Tranziția normală în faza de execuție"
dacă      (faza selecție) și
regulile MT4 și MT6 nu (mai) pot fi aplicate
atunci elimină (faza selecție) și
adaugă (faza execuție)
```

Meta-regula următoare descrie revenirea (*backtracking*). Ea se aplică când lista filtrată e vidă dar stiva regulilor de aplicat indică încă alte posibilități. Figura 15 arată schematic operațiile de efectuat în acest caz.

MT6: "Revenire cu tranziție în faza de execuție, dacă nici o regulă nu a putut fi selectată"

```
dacă      (faza selecție) și
(reguliFiltrate) și
(reguliAplicate NISTE-REGULI) și
(bazaDeFapteCurenta NISTE-FAPTE-CURENTE) și
(stivaReguliDeAplicat O-LISTA-DE-APL REST-STIVA-DE-APL) și
(stivaReguliAplicate O-LISTA-APL REST-STIVA-APL) și
(stivaBazelorDeFapte NISTE-FAPTE REST-BAZE-DE-FAPTE)
atunci elimină (faza selecție)
elimină (reguliFiltrate) și
elimină (reguliAplicate NISTE-REGULI) și
elimină (bazaDeFapteCurenta NISTE-FAPTE-CURENTE) și
elimină (stivaReguliDeAplicat O-LISTA-DE-APL REST-STIVA-
DE-APL) și
elimină (stivaReguliAplicate O-LISTA-APL REST-STIVA-APL)
și
elimină (stivaBazelorDeFapte NISTE-FAPTE REST-BAZE-DE-
FAPTE) și
```


adaugă (faza execuție) **și**
adaugă (reguliFiltrate O-LISTA-DE-APL) **și**
adaugă (reguliAplicate O-LISTA-APL) **și**
adaugă (bazaDeFapteCurenta NISTE-FAPTE) **și**
adaugă (stivaReguliDeAplicat REST-STIVA-DE-APL) **și**
adaugă (stivaReguliAplicate REST-STIVA-APL) **și**
adaugă (stivaBazelorDeFapte REST-BAZE-DE-FAPTE)

Meta-regula MT6 descrie maniera în care se actualizează memoriile ce păstrează starea curentă a sistemului, respectiv regulile filtrate vor fi date de lista din vârful stivei regulilor de aplicat, regulile aplicate vor fi luate din vârful stivei regulilor aplicate, iar baza de fapte curentă – din vârful stivei bazelor de fapte. Toate stivele sunt simultan decrementate.

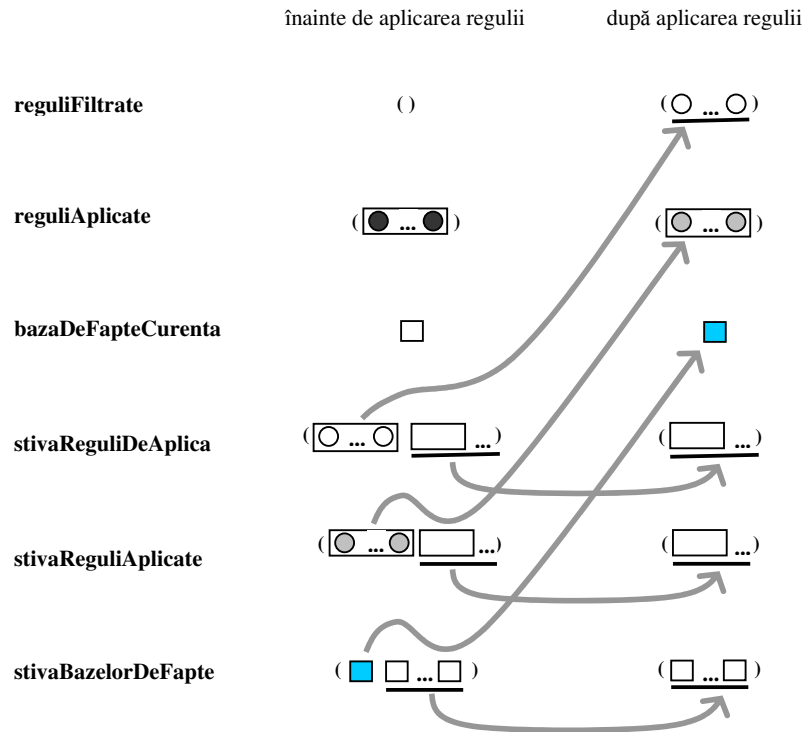


Figura 15: Operațiile în revenire (*backtracking*)

În faza de execuție, cea mai bine plasată regulă dintre cele filtrate este executată:

MT7: “Faza de execuție”

```

Dacă      (faza execuție) și
(reguliFiltrate R REST-FILTRATE) și
(regula R PRI PREMISE ACTIUNI) și
(reguliAplicate O-LISTA-REG-APL) și
(bazaDeFapteCurenta NISTE-FAPTE)
(stivaReguliDeAplicat LISTE-REG-DE-APL) și
(stivaReguliAplicate LISTE-REG-APL) și
(stivaBazelorDeFapte LISTE-DE-FAPTE) și
atunci    execută ACTIUNI în contextul NISTE-FAPTE ce, la rândul lor,
sunt transformate în
NOILE-FAPTE și
elimină   (faza selecție) și
elimină   (reguliFiltrate R REST-FILTRATE) și
elimină   (reguliAplicate O-LISTA-REG-APL) și
elimină   (bazaDeFapteCurenta NISTE-FAPTE) și
elimină   (stivaReguliDeAplicat LISTE-REG-DE-APL) și
elimină   (stivaReguliAplicate LISTE-REG-APL) și
elimină   (stivaBazelorDeFapte LISTE-DE-FAPTE) și
adaugă   (faza filtrare) și
adaugă   (reguliFiltrate) și
adaugă   (reguliAplicate R O-LISTA-REG-APL) și
adaugă   (bazaDeFapteCurenta NOILE-FAPTE) și
adaugă   (stivaReguliDeAplicat REST-FILTRATE LISTE-REG-DE-
APL) și
adaugă   (stivaReguliAplicate O-LISTA-REG-APL LISTE-REG-
APL) și
adaugă   (stivaBazelorDeFapte NISTE-FAPTE LISTE-DE-FAPTE)

```

Figura 16 descrie operațiile necesare actualizării memoriilor de lucru și a stivelor ce păstrează configurația curentă, pentru cazul în care regula ce se execută nu este singura în lista de reguli filtrate. R, regula cea mai prioritară, așadar cea aflată pe prima poziție în lista regulilor filtrate (reguliFiltrate R REST-FILTRATE), se aplică asupra faptelor din baza de fapte curente, NISTE-FAPTE, producând noua bază de fapte, NOILE-FAPTE, ce vor fi utilizate în faza următoare de filtrare. În plus, lista faptelor filtrate este resetată, pentru a pregăti această nouă fază de filtrare, iar regula R este adăugată listei regulilor aplicate în firul curent de procesare O-LISTA-REG-APL. Simultan, restul regulilor filtrate REST-FILTRATE, ce nu vor mai fi folosite pe firul curent de procesare, sunt “împinse” în stiva de reguli de aplicat, care va deveni acum (stivaReguliDeAplicat REST-FILTRATE LISTE-REG-DE-APL).

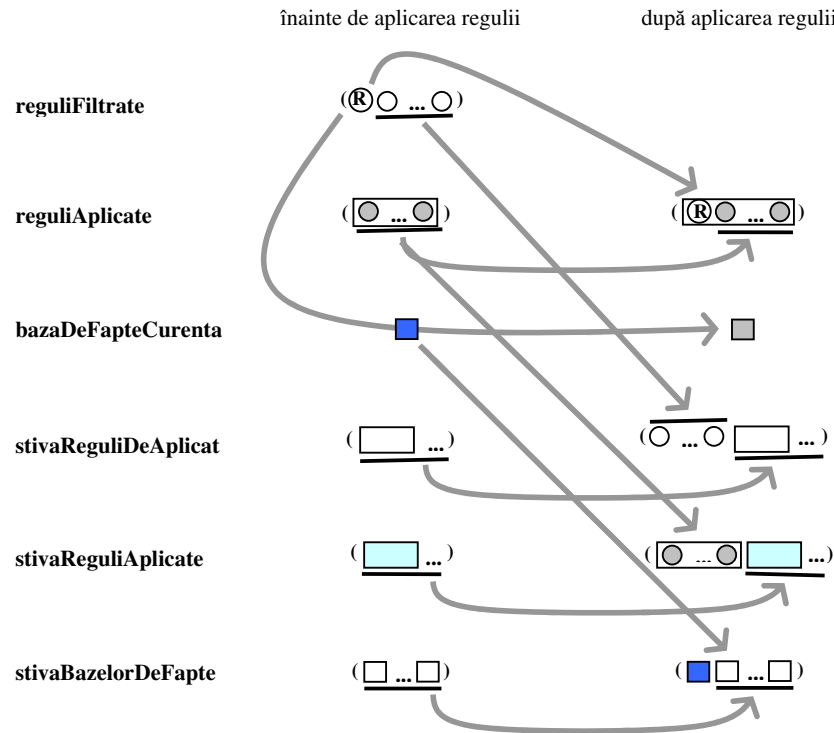


Figura 16: Faza de execuție în cazul mai multor reguli filtrate

După cum știm, cele trei stive trebuie să funcționeze sincron în privința numărului de intrări. Stiva regulilor de aplicat, a celor aplicate, ca și stiva bazelor de fapte trebuie să facă posibilă revenirea în starea în care se găsea sistemul înainte de executarea regulii curente R , dacă, la un moment ulterior, o reluare a procesării din acest loc este necesară, simultan cu încercarea unei alte reguli din cele filtrate dar ignorate pentru moment. Ca urmare, la stiva regulilor aplicate, ce ține “istoria” regulilor aplicate în firul curent de procesare, se adaugă lista regulilor aplicate înainte de execuția lui R , adică $O\text{-}LISTA\text{-}REG\text{-}APL$, iar la stiva bazelor de fapte se adaugă baza de fapte de dinaintea aplicării regulii R , respectiv $NISTE\text{-}FAPTE$.

Dacă R este, însă, singura regulă filtrată, atunci nici una din stive nu este incrementată. Meta-regula MT8 și Figura 17 descriu operațiile corespunzătoare acestei situații.

MT8: “Faza de execuție când stivele rămân neschimbate”

Dacă (faza execuție) **și**
 (reguliFiltrate R) **și**
 (regula R PRI PREMISE ACTIUNI) **și**

```

(reguliAplicate O-LISTA-REG-APL) și
(bazaDeFapteCurenta NISTE-FAPTE)
(stivaReguliDeAplicat LISTE-REG-DE-APL) și
(stivaReguliAplicate LISTE-REG-APL) și
(stivaBazelorDeFapte LISTE-DE-FAPTE) și
atunci   execută ACTIUNI în contextul NISTE-FAPTE ce, la rândul lor,
sunt transformate în
NOILE-FAPTE și
elimină (faza selecție) și
elimină (reguliFiltrate R) și
elimină (reguliAplicate O-LISTA-REG-APL) și
elimină (bazaDeFapteCurenta NISTE-FAPTE) și
adaugă (faza filtrare) și
adaugă (reguliFiltrate) și
adaugă (reguliAplicate R O-LISTA-REG-APL) și
adaugă (bazaDeFapteCurenta NOILE-FAPTE)

```

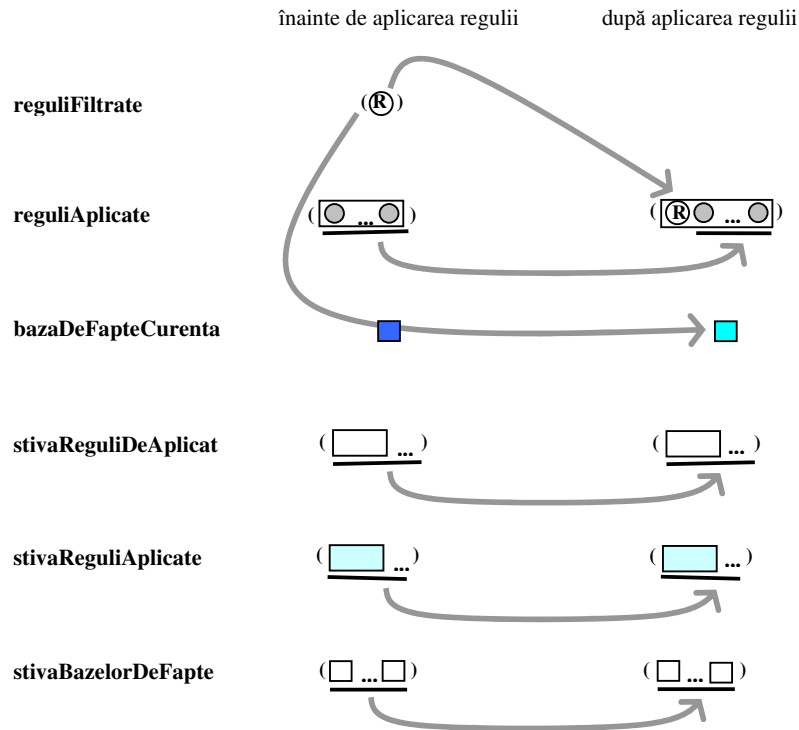


Figura 17: Execuția în cazul unei singure reguli filtrate

Capitolul 6

Confruntarea rapidă de șabloane: algoritmul RETE

Programatorii experimentați recunosc ușurința de concepere a programelor într-un limbaj bazat pe reguli. Faptul că elementele bazei de cunoștințe ce fac obiectul prelucrării nu trebuie identificate printr-un proces de căutare explicit ci sunt regăsite automat de sistem prin procesul “cablat” de confruntare de șabloane este un hocus-pocus extrem de convenabil programatorului. După cum bănuim însă, pentru că nimic pe lumea aceasta nu e pe gratis, un preț trebuie plătit pentru această conveniență: fie un timp de calcul mai lung, fie un consum de memorie mai mare. În acest capitol vom face cunoștință cu un algoritm de confruntare a regulilor cu faptele, datorat lui Forgy [14], [27], [16] care este eficient, dar care nu e tocmai ieftin în privința utilizării memoriei (pentru alte abordări, a se vedea [9], [28], [29], [31]).

Oricare din regulile unui sistem bazat pe reguli este formată dintr-o parte de condiții și o parte de acțiuni. Partea de condiții constă într-o colecție de șabloane ce trebuie confruntate cu obiecte din baza de fapte. Regulile considerate active, deci cele ce vor popula agenda la fiecare pas, sunt depistate în faza de filtrare. Nu e de mirare, așadar, că cel mai mult timp în rularea unui sistem bazat pe reguli se consumă în faza de filtrare, responsabilă, la fiecare ciclu, cu identificarea instanțelor de reguli, deci a tripletelor: reguli filtrate, fapte folosite în filtrări, legări ale variabilelor la valori.

Problema poate fi definită astfel: se dau un număr de reguli, fiecare având în partea stângă un număr de șabloane, și o mulțime de fapte și se cere să se găsească toate tripletetele <regulă, fapte, legări> ce satisfac șabloanele. O manieră de calcul imediată ar fi ca pentru fiecare șablon al fiecărei reguli să se parcurgă baza de fapte pentru a le găsi pe acelea ce verifică șablonul. Simpla iterare a setului de părți stângi ale regulilor peste setul de fapte din bază pentru găsirea acelor fapte care se potrivesc peste șabloanele regulilor este extrem de costisitoare, pentru că ar trebui să aibă loc la fiecare pas al motorului. Atunci când numărul regulilor și al faptelor este mare se pune așadar problema reducerii timpului fazei de filtrare. Iterația șabloanelor peste fapte la fiecare pas este inefficientă, printre altele, și pentru că, de la o iterație la alta, în general, numărul faptelor modificate este mic și deci foarte multe calcule s-ar repeta fără rost. Calculul inutil poate fi evitat dacă la fiecare ciclu s-ar memora toate potrivirile ce au rămas nemodificate de la ciclul anterior, astfel încât să se calculeze numai potrivirile în care intervin faptele nou adăugate,

modificate ori șterse. Apare naturală ideea ca faptele să-și găsească regulile și nu invers.

În algoritmul RETE regulile sistemului sunt compilate într-o rețea de noduri, fiecare nod fiind atașat unei comparații (test) ce are loc datorită unui anumit șablon. În construcția rețelei compilatorul ține cont de similaritatea structurală a regulilor, minimizând astfel numărul de noduri. Similaritatea a două șabloane înseamnă identitatea lor modulo numele variabilelor. Șabloanelor similare aflate în secvențe identice în mai multe reguli le corespund aceleași noduri în rețea.

Algoritmul exploatează relativa stabilitate a faptelor de la un ciclu la altul. Nodurile rețelei sunt supuse unui proces de activare-dezactivare ce se propagă dinspre rădăcină spre frunze la fiecare ciclu al motorului. Porțiunile din rețea ce nu sunt afectate de schimbările din baza de fapte prin aplicarea ultimei reguli rămân în aceeași stare. Un nod din rețea ce-și schimbă starea antrenează însă modificări în toată structura aflată sub el. Prin aceasta, porțiuni largi din rețea rămân imobile între o execuție și alta. În extremis, modificarea întregii baze de fapte antrenează recalcularea întregii rețele ca și cum toate faptele ar fi fost comparate cu toate șabloanele regulilor.

O regulă este considerată potențial activă, deci inclusă în agendă, atunci când toate șabloanele părții ei stângi sunt verificate pe fapte din bază (în sensul că se potrivesc asupra lor). Dacă, de exemplu, de la un ciclu la următorul, șablonul al treilea din patru al unei reguli își modifică starea în raport cu baza (de exemplu încetează a mai fi verificat), atunci rețeaua trebuie să păstreze informația că primele două șabloane sunt încă verificate și o dezactivare de la nivelul șablonului al treilea trebuie să se propage mai jos. Invers, dacă un șir de patru șabloane al unei reguli era doar parțial verificat la un anumit ciclu, să zicem până la nivelul celui de al doilea șablon inclusiv, pentru ca la următorul ciclu o modificare în bază să provoace verificarea și a următoarelor două, atunci un semnal în rețea corespunzând acestor ultime două șabloane trebuie să se propage pentru a anunța totodată aprinderea regulii.

Această observație este de natură să atragă atenția asupra **potrivirilor parțiale**. O potrivire parțială a unei reguli este orice combinație contiguă de șabloane satisfăcute începând cu primul șablon al regulii. O potrivire dintre un șablon și un fapt o vom numi **potrivire simplă**.

Fie, de exemplu, faptele următoare:

```
f1: [alpha a 3 a]
f2: [alpha a 4 a]
f3: [alpha a 3 b]
f4: [beta 3 a 3]
f5: [beta 3 a 4]
f6: [beta 4 a 4]
f7: [gamma a 3]
f8: [gamma b 4]
```

și o regulă:

```
R1:
dacă    [alpha X Y X] și
          [beta Y X Y] și
          [gamma X Y]
atunci ...
```

în care X și Y sunt variabile. Au loc următoarele potriviri simple și parțiale:

- potriviri simple pentru primul șablon:
 - $f_1; X \rightarrow a, Y \rightarrow 3$
 - $f_2; X \rightarrow a, Y \rightarrow 4$
- potriviri simple pentru al doilea șablon:
 - $f_4; Y \rightarrow 3, X \rightarrow a$
 - $f_6; Y \rightarrow 4, X \rightarrow a$
- potriviri simple pentru al treilea șablon:
 - $f_7; X \rightarrow a, Y \rightarrow 3$
 - $f_8; X \rightarrow b, Y \rightarrow 4$
- potriviri parțiale pentru primele două șabloane:
 - $f_1, f_4; X \rightarrow a, Y \rightarrow 3$
 - $f_2, f_6; X \rightarrow a, Y \rightarrow 4$
- potriviri parțiale pentru toate trei șabloanele:
 - $f_1, f_4, f_7; X \rightarrow a, Y \rightarrow 3$

Ultima potrivire parțială corespunde activării:

$\langle R1; f_1, f_4, f_7; X \rightarrow a, Y \rightarrow 3 \rangle$.

Rețeaua oglindește două tipuri de comparații dintre fapte și șabloane: intra-șablon și inter-șabloane, care definesc și două rețele așezate în cascadă.

Există un unic nod rădăcină al rețelei comparațiilor intra-șablon. Pe nivelul aflat imediat sub rădăcină sunt noduri în care se realizează confruntări simple, corespunzând potrivirilor elementare dintre șabloane și fapte. În fapt, fiecare nod al rețelei de șabloane implementează câte un test elementar, care este responsabil de verificarea diverselor inter-relații intra-șablon. Aceste noduri sunt plasate în cascadă, oglindind ordinea operațiilor de comparație corespunzătoare confruntărilor șablon-fapt. În felul acesta, fiecare coloană de noduri corespunde câte unui șablon, de fapt, datorită similarității structurale, câte unui set de șabloane structural similare aparținând unor reguli diferite. Nodurile terminale ale acestui nivel corespund, fiecare, unui set de potriviri simple similare.

În Figura 18 este arătată o rețea care corespunde compilării regulii R1 de mai sus și R2 de mai jos:

R2:
dacă [alpha Z U Z] **și**
 [beta U Z U] **și**
 [gamma Z 4]
atunci ...

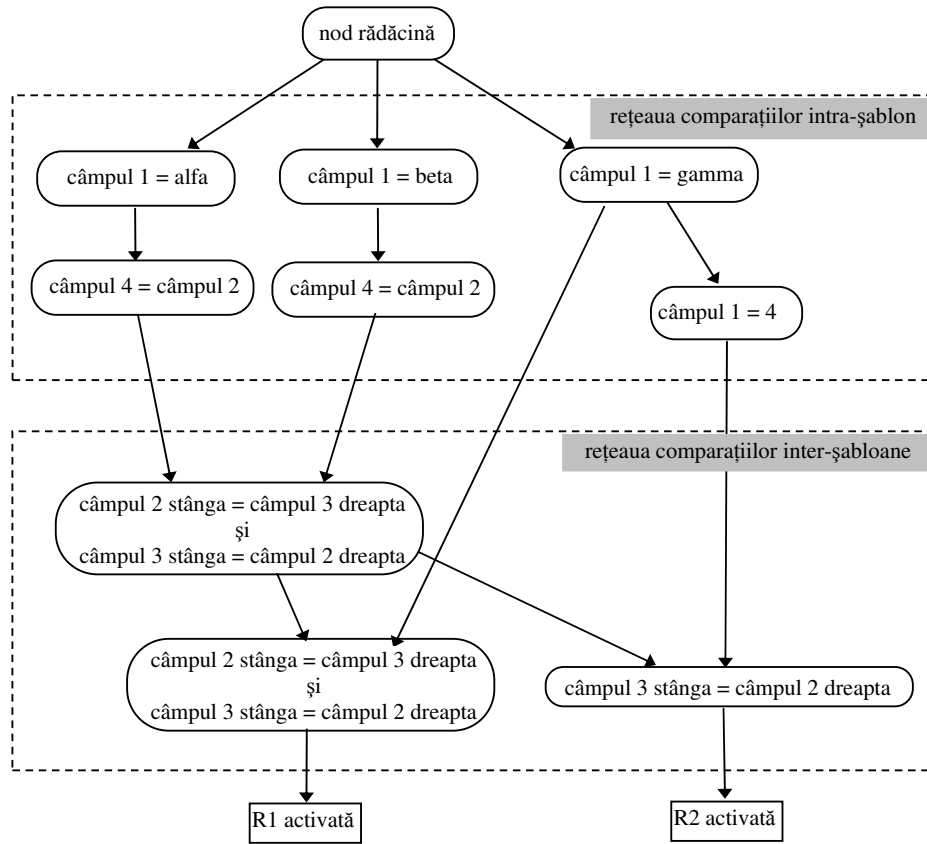


Figura 18: Rețeaua corespunzând regulilor R1 și R2

Cu excepția numelor variabilelor, primele două șabloane ale regulilor R1 și R2 sunt similare pentru că, redenumind variabilele Z și U din R2 respectiv în X și Y, obținem aceleași șabloane. Acest lucru este oglindit în primele două coloane ale

sub-rețelei comparațiilor intra-șablon care conțin noduri utilizate în comun de ambele reguli. Astfel primul nod al primei coloane exprimă testul de identificare a faptelor care au pe primul câmp simbolul α . Următorul câmp al primului șablon conține variabila X în $R1$, respectiv Z în $R2$. Acest câmp nu generează nici un test pentru că orice valoare pe poziția corespunzătoare lui într-un fapt ce a trecut cu bine de primul test este acceptată. Câmpul al treilea al primului șablon al regulii $R1$ conține variabila Y , respectiv U în $R2$, pentru care, din nou, nu se generează nici un test din același motiv. Câmpul al patrulea al primului șablon al lui $R1$ evocă variabila X din nou, respectiv variabila Z în $R2$. Pentru acest câmp trebuie generat un test pentru că apariția a doua oară a aceleiași variabile într-un șablon indică necesitatea identității valorilor de pe pozițiile corespunzătoare ale faptului supus confruntării. Ca urmare, rețeaua include un nod care testează egalitatea câmpului 4 al faptului cu câmpul 2 al faptului, lucru valabil pentru ambele reguli.

Coloana următoare a rețelei exprimă testele intra-șablon pentru șabloanele secunde ale lui $R1$ respectiv $R2$: primul câmp din fapt trebuie să conțină simbolul β iar câmpul 4 al faptului, din nou, să fie egal cu câmpul 2 al faptului (datorită apariției a doua oară a variabilei Y în $R1$ respectiv a variabilei U în $R2$). Coloana a treia a rețelei, exprimând testele intra-șablon pentru șabloanele de pe poziția a treia a regulilor $R1$ respectiv $R2$, are un nod comun pentru ambele reguli, corespunzând primului câmp din fapt ce trebuie testat la egalitate cu simbolul γ . Mai departe numai regula $R2$ mai necesită un test intra-șablon, corespunzând necesității ca pe poziția câmpului al treilea din fapt să se regăsească valoarea 4.

Sintetizând, potrivirile simple ale regulii $R1$, datorate secvenței celor trei șabloane, sunt testate în primele două noduri ale coloanei 1, primele două noduri ale coloanei 2 și, respectiv, primul nod al coloanei 3. Analog, potrivirile simple ale regulii $R2$ sunt testate în primele două noduri ale coloanei 1, primele două noduri ale coloanei 2 și, respectiv, al doilea nod al coloanei 3. După cum se poate observa și pe figură, fiecărei potriviri simple îi corespunde o ieșire din sub-rețeaua comparațiilor intra-șablon.

Imediat sub rețeaua comparațiilor intra-șablon se află rețeaua comparațiilor inter-șabloane. Prin intermediul ei sunt reprezentate testele datorate unor variabile partajate de mai multe șabloane. Fiecare nod al acestei rețele are două intrări. Un nod reprezintă aici o potrivire parțială, cele două intrări corespunzând, una potrivirii parțiale de rang imediat inferior și cealaltă – șablonului următor supus testului. Ultimul nivel al rețelei corespunde nodurilor reguli.

Astfel, în sub-rețeaua comparațiilor inter-șabloane din Figura 18, regulii $R1$ îi corespund două noduri iar regulii $R2$ de asemenea două noduri, deși numărul total de noduri ale sub-rețelei este de trei, pentru că un nod este partajat de cele două reguli.

Figura 19 detaliază maniera în care trebuie considerate intrările nodurilor comparație ale sub-rețelei inter-șabloane a regulii $R1$. Luând ca exemplu primul

nod, este clar că în prima comparație am fi putut considera câmpul al patrulea al intrării stânga în locul câmpului al doilea, întrucât în ambele câmpuri în șablon apare aceeași variabilă iar egalitatea celor două câmpuri a fost testată în sub-rețeaua comparațiilor intra-șablon, în cel de al doilea nod de pe prima coloană.

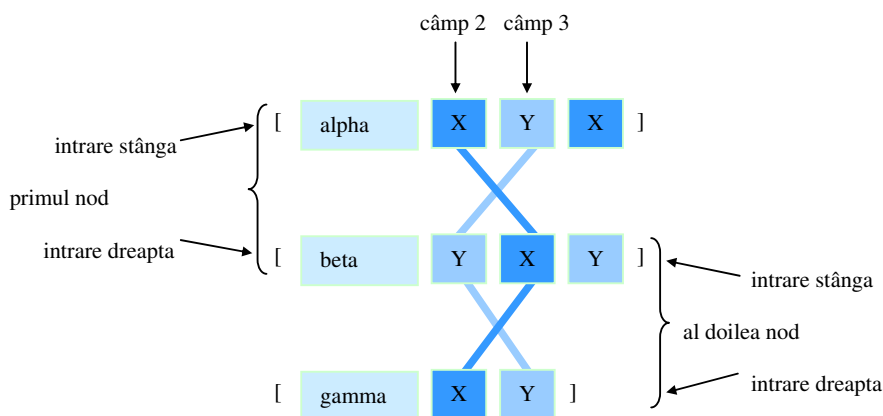


Figura 19: Corelațiile inter-șabloane ale regulii R1

O rețea ca cea descrisă are rolul de a păstra între cicluri succesive ale funcționării motorului informații despre potrivirile simple și parțiale și de a propaga modificările ce apar în baza de fapte, ca urmare a execuției regulilor. În felul acesta rețeaua va ține evidența instanțierilor de reguli. Pentru a realiza acest lucru, fiecare nod al rețelei are atașată o memorie ce va memora potrivirile, simple ori parțiale, corespunzătoare nivelului pe care este plasat nodul.

Nodul rădăcină al rețelei are rolul de a colecta modificările ce apar în baza de date la fiecare ciclu și a le distribui nodurilor aflate imediat sub el. În conformitate cu tipurile de acțiuni pe care le-am admis până acum asupra faptelor din bază, vom considera că există doar două tipuri de modificări ce pot apărea în rețea: adăugarea unui fapt și ștergerea unui fapt. O vom numi pe prima – modificare plus (+) și pe cea de a doua – modificare minus (–).

Secvența de figuri care urmează arată propagarea modificărilor în rețea pe măsură ce faptele sunt introduse în bază. Amorsarea propagărilor se face prin inserția în nodul rădăcină a unor dublete de forma $\langle \text{index-fapt}, + \rangle$. Semnul + semnifică aici modificarea plus. Ori de câte ori un fapt este retras din bază rădăcina va propaga o modificare minus, de forma: $\langle \text{index-fapt}, - \rangle$. Pe aceste figuri ultimele noduri aprinse în propagare sunt întunecate.

Astfel, Figura 20 reprezintă situația memoriei atașate rețelei după propagarea secvenței de modificări: $\langle f_1, + \rangle$, $\langle f_2, + \rangle$, $\langle f_3, + \rangle$. Faptele f_1 : [alpha a 3 a] și f_2 : [alpha a 4 a] ating nivelul nodului secund al primei coloane, pentru că ambele teste sunt satisfăcute de câmpurile acestor fapte. Totodată, pentru că ele nu satisfac condițiile din nodurile de pe celelalte două coloane, ele nu pot trece de nici unul din aceste teste. Faptul f_3 : [alpha a 3 b] nu poate trece mai jos de primul nod al primei coloane.

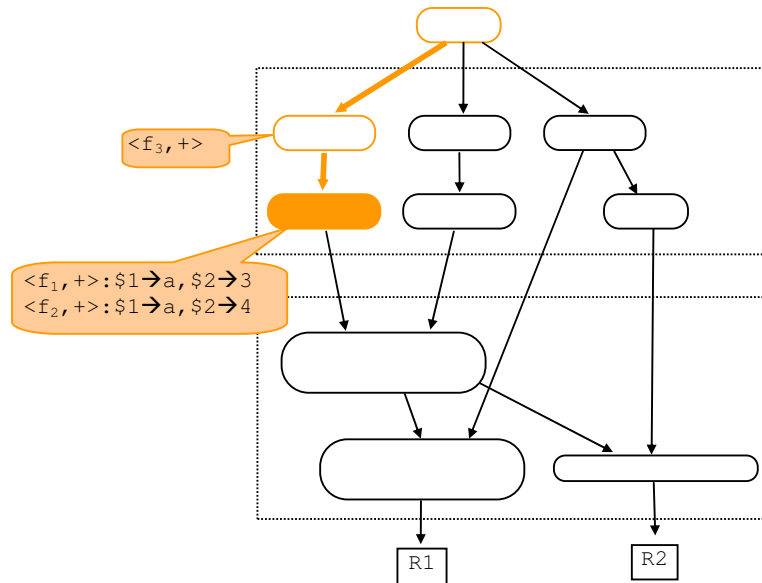
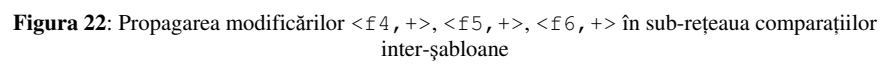
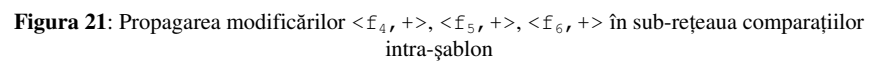


Figura 20: Propagarea modificărilor $\langle f_1, + \rangle$, $\langle f_2, + \rangle$, $\langle f_3, + \rangle$

Simultan cu propagarea modificărilor în josul rețelei, trebuie să aibă loc o memorare a legării câmpurilor variabile din șabloanele regulilor la valori, în cazurile de potriviri. Vom vizualiza această memorie numai în nodurile terminale ale rețelei comparațiilor intra-șablon. Cum nodurile din rețea pot fi partajate de mai multe reguli, fiecare numind altfel propriile variabile, notarea variabilelor în aceste asignări trebuie să fie neutră față de notațiile utilizate în reguli. De aceea, pe figuri, aceste variabile s-au notat prin \$1, \$2... Semnificația acestora este însă diferită de la un nod la altul.

Considerând că faptele f_4 , f_5 și f_6 sunt propagate în rețea ca modificări plus, Figura 21 arată cum faptul f_5 nu poate trece mai jos de primul nod al coloanei a doua, în timp ce modificările plus datorate faptelor f_4 și f_6 ajung la baza rețelei comparațiilor intra-șablon pe coloana a doua.



În acest moment două intrări ale primului nod al sub-rețelei comparațiilor inter-șabloane sunt active. Figura 22 arată potrivirea parțială ce se produce la nivelul acestui nod, prin selectarea respectiv a perechilor de două fapte $\langle f_1, f_4 \rangle$ și $\langle f_2, f_6 \rangle$. Acesta este cel mai jos punct în care modificările pot fi încă propagate.

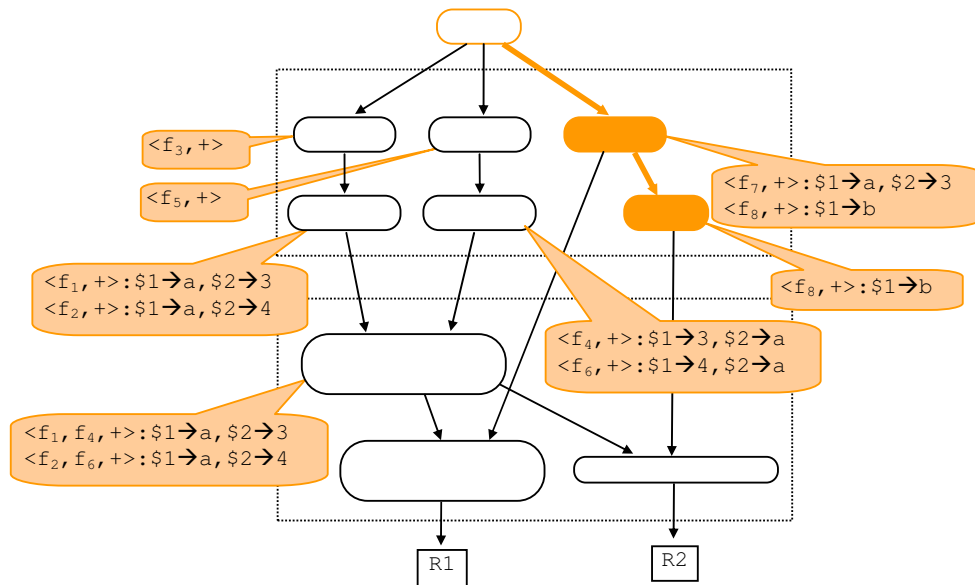


Figura 23: Propagarea modificărilor $\langle f7, + \rangle$, $\langle f8, + \rangle$ în rețeaua comparațiilor intra-șablon

Propagarea în sub-rețeaua comparațiilor intra-șabloane a modificărilor plus, datorate apariției în bază a faptelor \mathfrak{f}_7 și \mathfrak{f}_8 sunt arătate în Figura 23.

În fine, Figura 24 pune în evidență propagarea ultimelor modificări în sub-rețeaua comparațiilor inter-șablon. Se constată că numai regula R1 poate fi filtrată prin satisfacerea condițiilor nodului înnegrit de către tripletul de fapte $\langle f_1, f_4, f_7 \rangle$. Instanța ce va popula agenda va fi în acest caz: $\langle R1; f_1, f_4, f_7; X \rightarrow a, Y \rightarrow 3 \rangle$. Dimpotrivă, nici unul din tripletele $\langle f_1, f_4, f_8 \rangle$ ori $\langle f_2, f_6, f_8 \rangle$ ce alimentează intrările ultimului nod dreapta al rețelei inferioare nu satisfac condiția din nod, ceea ce face ca regula R2 să nu poată fi filtrată.

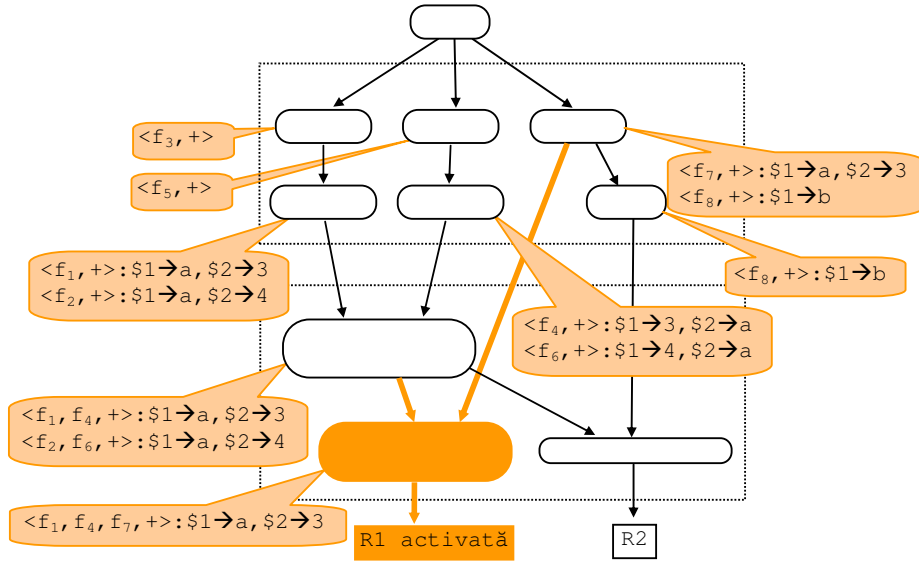


Figura 24: Propagarea modificărilor $\langle f_7, + \rangle$, $\langle f_8, + \rangle$ în rețeaua comparațiilor inter-șablon

Să presupunem, în continuare, că regula R1, filtrată așa cum s-a arătat mai sus, conține o parte dreaptă ca mai jos:

R1:
dacă [alpha X Y X] și
 [beta Y X Y]
 [gamma X Y]
atunci **șterge** [gamma X Y]
 adaugă [gamma X 4]

Execuția părții drepte a lui R1 injectează în nodul rădăcină al rețelei modificările $\langle f_7, - \rangle$, $\langle f_9, + \rangle$, unde f_9 este faptul [gamma a 4]. Propagarea modificării minus în rețea este indicată în Figura 25. Traseul modificărilor și nodurile afectate sunt desenate într-o culoare întunecată.

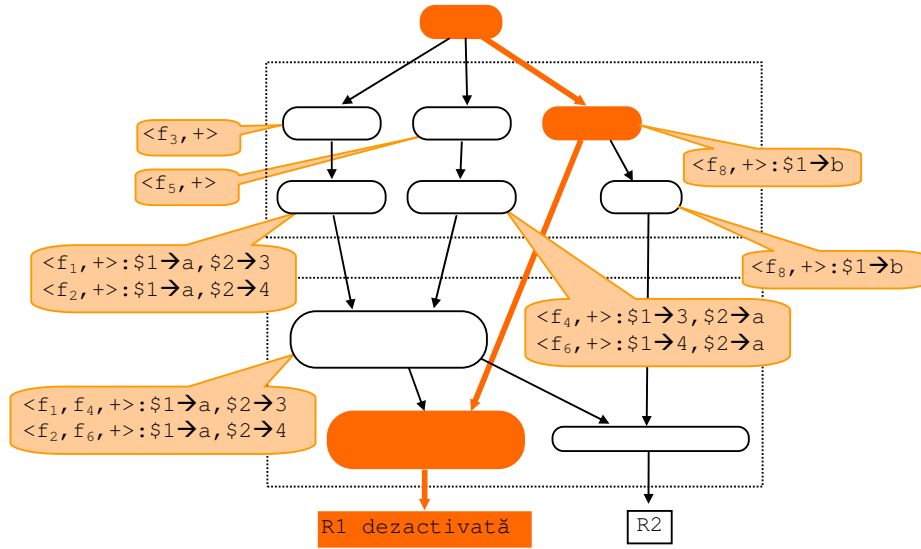


Figura 25: Propagarea modificării minus <f7, ->

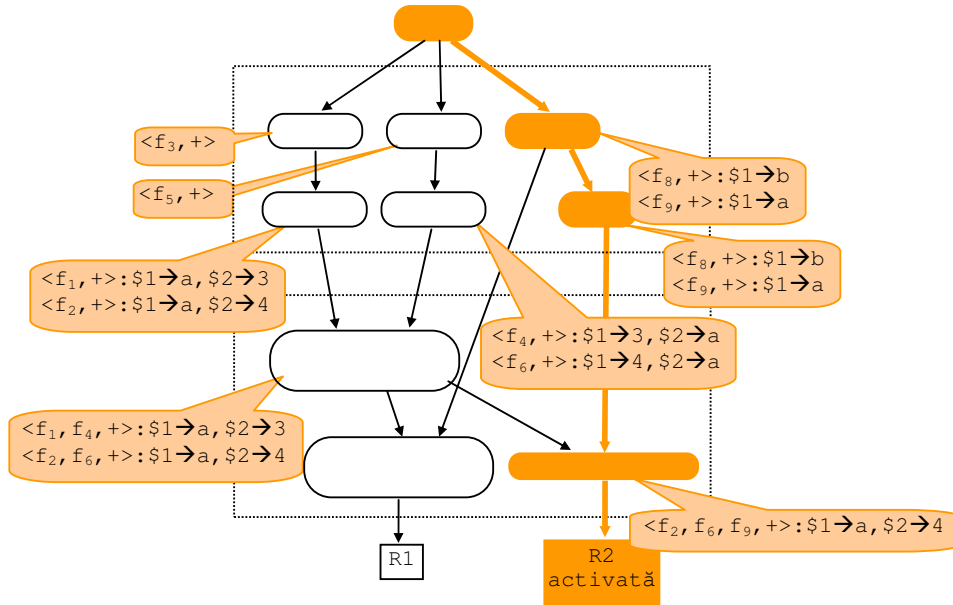


Figura 26: Propagarea modificării plus

În fine, propagarea modificării plus produse de aceeași execuție este indicată în Figura 26, unde, din nou, pe traseul propagării modificării, nodurile afectate sunt desenate într-o culoare întunecată.

6.1. Importanța ordinii șabloanelor

Considerațiile de mai sus privind construcția rețelei de șabloane a regulilor și maniera în care are loc propagarea modificărilor duc la formularea unor importante concluzii asupra eficientizării rulării. Cea mai importantă dintre ele se referă la ordinea șabloanelor în părțile stângi ale regulilor. Aceste concluzii au toate la bază maniera de calcul a potrivirilor parțiale: trecerea de la considerarea primelor k șabloane la considerarea primelor $k+1$ șabloane ale părții stângi poate adăuga suitei de potriviri parțiale datorate primilor k șabloane până la produsul dintre numărul acestora și numărul de potriviri elementare ale șablonului al $k+1$ -ulea. Deși reguli stricte în privința ordinii șabloanelor sînt dificil de formulat, explozia potrivirilor parțiale poate fi controlată într-o oarecare măsură respectând câteva recomandări:

- șabloanele cărora le corespund cele mai puține apariții de fapte în bază trebuie să apară pe primele poziții în părțile stângi ale regulilor. În felul acesta nodurilor finale ale rețelei comparațiilor intra-șablon le vor corespunde puține fapte verificate în baza de fapte. Astfel, nu este indiferent dacă primul șablon realizează 1 sau 100 de potriviri elementare, pentru că acestea vor intra în calculul potrivirilor parțiale ale rețelei comparațiilor inter-șabloane;
- șabloanele mai specifice trebuie să le preceadă pe cele mai generale. Cu cât un șablon are mai multe câmpuri libere sau conține mai multe variabile nelegate, cu atât el trebuie să apară mai jos în partea stângă a regulii. Ambele recomandări duc la micșorarea numărului de potriviri parțiale atașate nodurilor superioare din rețeaua comparațiilor inter-șabloane;
- șabloanele corespunzătoare faptelor celor mai volatile (ce sunt retrase și asertate des în bază) trebuie plasate la urmă. În felul acesta modificările propagate de includerea sau eliminarea faptelor corespunzătoare lor se vor manifesta în rețea doar dacă toate celelalte șabloane ale regulii se verifică;
- nu trebuie exagerată folosirea variabilelor multi-câmp și cu precădere a celor anonime (ele vor fi introduse în capitolul 7). Acestea duc de obicei la instanțieri multiple ale regulii, ce se pot apoi ușor multiplica prin instanțierile parțiale.

Uneori aceste recomandări pot fi contradictorii și uneori numai experimentând, după ce raționamentul este verificat, se poate găsi ordonarea optimă. Experiența programatorului poate fi aici de mare preț.

Partea a III-a

Elemente de programare bazată pe reguli

Primii pași într-un limbaj bazat pe reguli: CLIPS

Constrângeri în confruntarea șabloanelor

Despre controlul execuției

Recursivitatea în limbajele bazate pe reguli

Capitolul 7

Primii pași într-un limbaj bazat pe reguli: CLIPS

CLIPS este un *shell* evoluat pentru dezvoltarea de sisteme expert. El se încadrează în paradigma limbajelor bazate pe reguli și implementează o **căutare înainte irevocabilă**. CLIPS a fost dezvoltat inițial de Software Technology Branch la NATO Lyndon B. Johnson Space Center. Prima versiune a apărut în 1984, iar în 2002 a ajuns la versiunea 6.2, ca produs de domeniu public, această evoluție semnificând un șir impresionant de îmbunătățiri și extinderi ale limbajului.

Pentru a rezolva o problemă în CLIPS programatorul definește baza de cunoștințe și alege strategia de căutare, iar sistemul dezvoltă un proces de aplicare a regulilor care se perpetuează atât timp cât mai pot fi aplicate reguli pe faptele existente în bază.

Prezentarea care urmează nu este propriu-zis o introducere în CLIPS, cât o introducere în programarea bazată pe reguli. Ca urmare, nu vom insista asupra elementelor de limbaj, ce pot fi oricând găsite în documentația interactivă a limbajului sau în alte materiale orientate cu precădere asupra prezentării limbajului (v. de exemplu [15] ori situl CLIPS de la <http://www.ghg.net/clips/CLIPS.html>) cât asupra manierei în care se rezolvă probleme utilizând paradigma de programare bazată pe reguli de producție.

Prezentarea care urmează este una tributară credinței mele că un limbaj nou se învață cel mai eficient și mai rapid când ai de rezolvat o problemă. Ca urmare, elementele limbajului vor fi introduse într-o ordine indusă de exemplele pe care le vom aborda, pe măsură ce complexitatea mereu crescândă a problemelor tratate ne-o va impune. Cititorul chițibușar va găsi cu prisosință motive de nemulțumire în cuprinsul părților următoare ale cărții, care, adesea, vor anunța titluri de probleme în locul unor elemente de limbaj, așa cum se procedează în orice carte dedicată prezentării unui limbaj de programare. Pentru a ușura totuși accesul la elementele de limbaj introduse al acelui tip de cititor grăbit să programeze și care nu este dispus să caute aiurea o documentație, vom marca totuși pasajele ce povestesc câte ceva despre limbaj printr-o bară verticală pe marginea stângă a textului, ca aici.

CLIPS, ca și alte *shell*-uri de sisteme expert, este un limbaj interpretat. Aceasta înseamnă că o comandă dată la *prompter* este executată imediat și rezultatul întors utilizatorului. Interpretorul transformă codul sursă al programului

într-o rețea a regulilor ce va fi executată apoi de algoritmul RETE, așa cum am văzut în Capitolul 6.

Prompter-ul CLIPS este:

```
CLIPS>
```

7.1. Să facem o adunare

Comanda dată direct la prompter

Cum scriem în CLIPS că $2+1=3$? Operația de adunare se realizează prin operatorul de adunare +, scriind:

```
CLIPS> (+ 2 1)
```

urmat de un **<Enter>** (orice comandă se termină cu **<Enter>**). Interpretorul va întoarce:

```
3
```

după care vom avea din nou un *prompter*:

```
CLIPS>
```

Reținem așadar că sintaxa comenzilor este în paranteze rotunde, cuvântul cheie semnificativ trebuind dat pe prima poziție.

Fapte

Să presupunem acum că dorim să introducem numerele pe care vrem să le adunăm ca date de intrare. Precizarea intrării într-un program CLIPS se face fie direct de la tastatură, fie citindu-le dintr-un fișier, fie prin declarații de **fapte**. De exemplu, am putea avea două fapte ce conțin numerele de adunat:

```
(numar 2)  
(numar 1)
```

Un fapt poate fi de două feluri: o construcție multi-câmp, în care câmpurile nu au nume și ordinea lor este semnificativă – **fapte ordonate**, sau una în care câmpurile au nume și, ca urmare, ordinea lor nu mai e importantă – **obiecte**. În acest capitol vom lucra numai cu fapte ordonate.

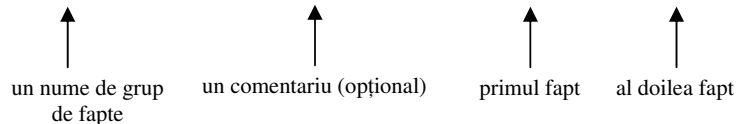
Un fapt este reprezentat cu unicitate în bază, acest lucru însemnând că sistemul va împiedica introducerea în bază a unui fapt identic cu unul deja existent. Aceasta nu înseamnă că o regulă care ar încerca o astfel de acțiune nu se aprinde. Ea lucrează, dar fără să afecteze baza.

Dar cum facem ca interpretorul să ia cunoștință de fapte? Cum la *prompter* el se așteaptă să primească comenzi, încercarea de a le scrie direct acolo va eșua pentru că el nu cunoaște nici o comandă `numar`:

```
CLIPS> (numar 2)
Missing function declaration for numar.
```

Comanda care comunică interpretorului un grup de fapte este `deffacts`:

```
(deffacts numere "numerele de adunat" (numar 2) (numar 1))
```



Grupul de fapte și numele lui nu au semnificație pentru program. Dar el poate fi de folos programatorului din rațiuni de structurare a datelor. Scriind la *prompter* acest rând, aparent nu se întâmplă nimic. Faptele sunt interpretate dar încă nu sunt depozitate în baza de fapte a sistemului. Interpretului trebuie să i se comande explicit ca faptele declarate prin definițiile de grupuri de fapte `deffacts` să fie depozitate în baza de fapte.

Comanda `(reset)` obligă includerea faptelor declarate în baza de fapte și alimentează rețeaua regulilor cu fapte, unul câte unul, în ordinea în care acestea sunt declarate și care va fi și ordinea de indexare a lor în bază. Verificarea conținutului bazei de fapte se face cu comanda `(facts)`.

Deci, să reluăm secvența:

```
CLIPS> (deffacts numere "numerele de adunat" (numar
2) (numar 1))
CLIPS> (reset)
CLIPS> (facts)
f-0      initial-fact
f-1      (numar 2)
f-2      (numar 1)
CLIPS>
```

Observăm că, în afara faptelor pe care le-am depus noi, ne este raportat încă un fapt asupra căruia nu avem nici un merit: `initial-fact`. Utilitatea acestui fapt, pe care sistemul îl introduce întotdeauna, stă în filtrarea regulilor

fără parte stângă. Sistemul înlocuiește partea stângă a regulilor care nu prevăd nici un șablon printr-un șablon `initial-fact`, ce va fi satisfăcut întotdeauna de acest fapt introdus din oficiu, făcând în acest fel posibilă și aprinderea acestor reguli.

Acum avem faptele în bază. Cum procedăm pentru a aduna cele două numere? Va trebui să construim o regulă capabilă să preia numerele din cele două fapte și să le adune. Această regulă va trebuie să fie atât de generală încât ori de câte ori conținutul acestor două fapte din bază s-ar modifica, ea să producă noua sumă.

Reguli

Pentru a defini o regulă folosim construcția `defrule`:

```
(defrule <nume-regulă> "comentariu" <șablon>* => <acțiune>*)
```

↑
orice regulă trebuie
să aibă un nume

↑
pot să scriu un
comentariu care să
descrie ce face regula
(opțional)

↑
partea stângă: un
șir de șabloane

↑
partea dreaptă: un
șir de acțiuni

Un **șablon** este o construcție care “imită”, mai mult sau mai puțin, un fapt ce ar trebuie să fie găsit în bază. În extremis, un șablon poate să fie identic cu un fapt, caz în care el se va **potrivi** numai cu acesta. Cel mai adesea însă un șablon permite anumite grade de libertate în structura faptelor cu care se intenționează să se potrivească. Operația de căutare în bază a faptelor care se potrivesc cu un șablon o vom numi **confruntare**.

Un șablon pentru fapte ordonate este o construcție sintactică de genul:

```
șablon ::= (<element-șablon>*)
element-șablon ::= <atom> | ? | ?<var> | $? | $?<var>
```

Construcția de mai sus (dată într-un format *Baccus-Naur*) exprimă succint faptul că un șablon este format din zero sau mai multe elemente-șablon, iar un element-șablon poate fi un atom sau o construcție în care apare semnul `?`. Un element-șablon atomic poate fi un întreg, un real, sau un simbol. Celelalte construcții se referă la câmpurile ce pot fi variabile în șablon. Confruntarea dintre un șablon și un fapt înseamnă o parcurgere a șablonului în paralel cu faptul și confruntarea fiecărui câmp din șablon cu unul sau mai multe câmpuri din fapt, în funcție de forma elementului-șablon. O regulă poate fi instanțiată diferit în funcție de legările care se realizează între variabilele șabloanelor și

câmpuri ale faptelor din bază. O instanțiere eșuează ori de câte ori elementele șablonului se epuizează înainte de epuizarea câmpurilor faptului sau invers:

- dacă elementul-șablon e un atom, atunci operația de confruntare reușește dacă câmpul respectiv din fapt este identic cu elementul-șablon;
- dacă elementul-șablon e ?, atunci operația reușește;
- dacă elementul-șablon e ?<var>, unde <var> este un simbol, și variabila ?<var> nu e legată, atunci operația reușește iar variabila ?<var>, ca efect colateral, va fi legată la valoarea din câmp;
- dacă elementul-șablon e ?<var>, și variabila ?<var> e deja legată, atunci, dacă valoarea din câmp este identică cu cea la care este legată variabila, atunci operația reușește, altfel ea eșuează;
- dacă elementul-șablon e \$?, atunci operația reușește și un număr de instanțieri ale regulii egal cu numărul câmpurilor rămase necercetate în fapt plus unu sunt produse. În fiecare instanțiere confruntarea continuă de la o poziție diferită din fapt. Astfel, în prima instanțiere confruntarea continuă din chiar poziția curentă a faptului, în a doua – de la o poziție situată un câmp mai la dreapta, ș.a.m.d.;
- dacă elementul-șablon e \$?<var> și variabila ?<var> nu e legată, atunci operația reușește și au loc aceleași acțiuni ca și în cazul elementului-șablon \$?. În plus, ca efect colateral, în fiecare instanțiere variabila ?<var> va fi legată la o secvență de câmpuri din fapt, cele peste care se face avansarea;
- dacă elementul-șablon e \$?<var> și variabila \$?<var> e deja legată la o valoare multi-câmp, atunci, dacă această valoare multi-câmp concordă cu secvența de câmpuri următoare din fapt, atunci operația reușește, altfel ea eșuează.

De remarcat din definițiile de mai sus că numele variabilei este ?<var> sau \$?<var>. Se interzice utilizarea în aceeași regulă a unui simbol de variabilă pentru a desemna simultan o variabilă simplu-câmp și una multi-câmp.

Până acum am definit construcții ce pot apărea în partea stângă a unei reguli. Ce acțiuni pot să apară în partea dreaptă? Pentru moment, ieșirea programelor se va manifesta prin modificări asupra bazei de fapte. Putem modifica colecția de fapte din bază în două moduri: adăugând noi fapte sau retrăgând fapte existente acolo.

■ Ca să adăugăm, scriem: (assert <fact>*). Ca să retragem, trebuie însă să precizăm despre ce fapt e vorba. Nu se pune problema să încercăm identificarea din nou a unui fapt în partea dreaptă a regulii prin intermediul unui șablon: un șablon nu poate acționa decât în partea stângă. Ca urmare faptul ce trebuie retras trebuie să fi fost deja identificat printr-un șablon și o adresă a lui să fi fost reținută. Ca să reținem adresa unui șablon, ori indexul lui, folosim construcția:

```
?<var-idx> <- <șablon>
```

care poate să apară doar în partea stângă a unei reguli. Dacă șablonul se potrivește peste un fapt din bază, atunci variabila `?<var-idx>` se va lega la indexul faptului. Cu aceasta, retragerea simultan a unui număr oarecare de fapte poate fi făcută printr-o comandă: `(retract ?<var-idx>*)`.

Revenind la exemplul de adunare a celor două numere, o primă tentativă ar putea fi următoarea:

```
(defrule aduna (numar ?x) (numar ?y) => (assert (suma = (+
?x ?y))))
```

Faptul nou introdus cu comanda `(assert ...)` conține activarea unei evaluări (semnul `=` în fața apelului de funcție) ce utilizează operatorul de adunare. Cititorul va realiza că în aceeași manieră se poate forța o evaluare cu orice alt operator, ori de câte ori se dorește ca valoarea unui câmp dintr-un fapt adăugat să rezulte dinamic.

Până acum programul nostru a constatat dintr-o singură regulă. Pe aceasta am putut-o scrie direct la *prompter*. Dar când dimensiunea unui program e mai mare, maniera firească este să-l edităm într-un fișier și, ca urmare, să-l încărcăm de acolo.

Comanda pentru încărcarea unui șir de declarații dintr-un fișier este `(load <nume-fișier>)`.

Scriind o secvență de declarații `defrule` la *prompter*, sau comandând încărcarea lor dintr-un fișier, se construiește baza de reguli, ceea ce revine la construcția rețelei. Comanda `(reset)` forțează apoi propagarea modificărilor plus în rețeaua de reguli prin alimentarea nodului rădăcină. Prin aceasta se realizează fazele de filtrare și selecție. Lansarea efectivă în execuție a regulii selectate se realizează prin comanda `(run)`. Această comandă amorsează procesul de inferență.

În capitolele 3 și 6 am arătat maniera în care se realizează legările variabilelor din părțile stângi ale regulilor la valori, ce sunt instanțele de regulă și cum se calculează acestea. Bănuim, probabil, că nu e cazul să ne așteptăm ca lansarea programului construit mai sus să producă o bază de fapte de genul:

```
CLIPS> (facts)
f-0      initial-fact
f-1      (numar 2)
f-2      (numar 1)
f-3      (suma 3)
CLIPS>
```

Într-adevăr, afișarea bazei de fapte, cu comanda `(facts)`, relevă existența în bază a încă două fapte sumă:


```

f-0      initial-fact
f-1      (numar 2)
f-2      (numar 1)
f-3      (suma 2)
f-4      (suma 3)
f-5      (suma 4)

```

Cum de au apărut acolo și sumele 2 și 4? Pentru a răspunde, să revedem mecanismul de aplicare a regulilor. Să presupunem că o regulă își satisface toate șabloanele asupra unei configurații de fapte din bază. Spunem că s-a creat o **activare** a respectivei reguli. În acest caz, cel puțin o **instanță** a regulii trebuie să apară în agendă. Așa cum știm, o instanță a unei reguli este formată dintr-un triplet ce asociază numelui regulii o secvență de fapte, câte unul pentru fiecare șablon din partea de condiții a regulii, și o configurație de legări a variabilelor proprii la valori. Aceeași regulă, împreună cu o altă secvență a unor fapte ce se potrivesc cu secvența de șabloane a părții stângi, va genera o altă instanță. E posibil deci ca în instanțe diferite să participe aceleași fapte dar într-o altă ordine.

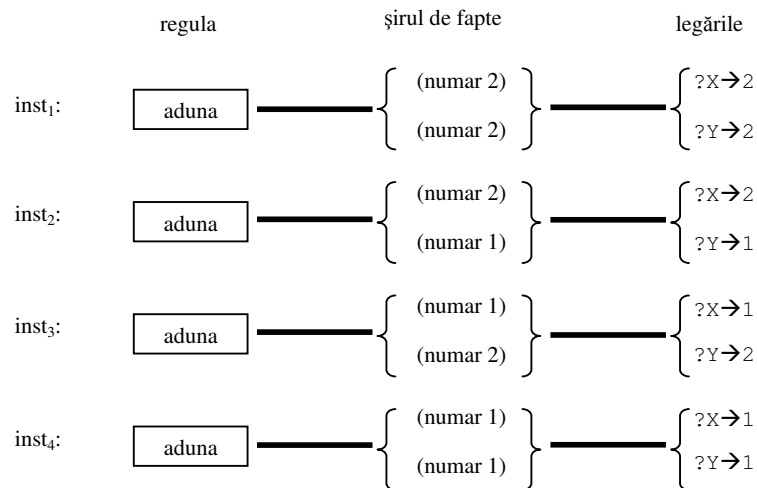


Figura 27: Instanțele regulii aduna

În exemplul nostru, în care apar regula aduna și faptele inițiale (numar 2) și (numar 1), partea de condiții a regulii, respectiv perechea de șabloane (numar ?x) și (numar ?y), va crea instanțele schițate în Figura 27.

Ca urmare, pentru fiecare dintre aceste instanțe, presupunând că toate s-ar aplica, faptul calculat de partea dreaptă a regulii va fi:

```

inst1: ?X = 2, ?Y = 2 → (suma 4)
inst2: ?x = 2, ?y = 1 → (suma 3)
inst3: ?x = 1, ?y = 2 → (suma 3), deja existent
inst4: ?x = 1, ?y = 1 → (suma 1)

```

Un program care nu se mai termină

Modificând un singur simbol în regula `aduna` facem ca ea să se aplice la nesfârșit. Acest lucru se întâmplă, de exemplu, dacă în loc de fapte `(suma ...)` am crea tot fapte `(numar ...)`:

```

(defrule aduna (numar ?x) (numar ?y)=> (assert (numar =(+
?x ?y))))

```

Bucula infinită e datorată faptului că la crearea fiecărui nou element din bază de forma `(numar ...)`, acesta va participa în atâtea activări câte fapte sunt deja în bază, ceea ce va duce la apariția a tot atâtea fapte noi, ș.a.m.d.

Dacă programul a intrat într-o buclă infinită, nu avem altă soluție decât să oprim programul prin mijloace brutale (**<CTRL><C>** sau **<CTRL><ALT>**, de exemplu).

7.2. Cum realizăm o iterație?

Să calculăm suma elementelor unui vector

Primul exemplu de iterație ce se dă, de obicei, în orice carte de prezentare a unui limbaj este unul în care se parcurg elementele unei liste. Să încercăm și noi să realizăm o banală *adunare a unui șir de numere*. De data aceasta vom reprezenta șirul de numere printr-un unic fapt `numere`, care are aparența unui vector. Mai avem nevoie de un fel de registru care să memoreze rezultate parțiale ale sumei, la fiecare pas în iterație, și care să fie inițializat cu 0:

```

(def facts niste_numere
  (numere 2 7 5 3 4)
  (suma 0)
)

```

Regula următoare realizează iterația de sumare:

```

(defrule aduna
  ?r <- (numere ?x $?rest)
  ?s <- (suma ?y)

```

```
=>
(retract ?r ?s)
(assert (numere $?rest) (suma =(+ ?x ?y)))
)
```

Cum funcționează ea? Partea de condiție a regulii conține o secvență de două șabloane. Primul verifică existența în bază a unui fapt `numere` care e urmat de cel puțin un simbol (să observăm că nimic în regulă nu încearcă să valideze că ceea ce urmează după simbolul `numere` sunt simboluri numerice; vom risca să nu ne luăm această precauție). Condiția ca lungimea șirului de numere să fie cel puțin egală cu 1 este realizată prin variabila de șablon `?x` urmată de variabila multi-câmp `$?rest`. Rețineți această secvență, pentru că ea este comună multor operații de iterații pe șir. Șablonul `(numere ?x $?rest)` pune în evidență după simbolul `numere` obligator un simbol, eventual și altele. În urma confruntării, variabila simplă `?x` se va lega la primul număr din secvență, iar variabila multi-câmp `$?y` la restul lor.

Ce de-al doilea șablon caută un fapt `suma` și leagă variabila `?y` de suma declarată în acest fapt. Dacă aceste două condiții sunt îndeplinite, cele două fapte găsite sunt retrase și înlocuite cu un fapt `numere` mai scurt cu o poziție și, respectiv, o sumă actualizată.

Pentru orice ipostază a bazei de fapte există o unică activare, evident asociată acestei reguli, și, datorită acestui lucru, în lipsa conflictului, regula se aplică de fiecare dată. Aceasta se întâmplă până când faptul `numere` nu mai conține nici un număr, caz în care faptul `suma` va conține rezultatul.

Calculul unui maxim

Să presupunem că ne propunem să calculăm *maximul dintr-o secvență de numere*. Ca și mai sus, numerele sunt definite printr-un fapt `numere` care simulează un vector, iar un fapt `max` conține inițial primul număr din secvență. Prima soluție este următoarea:

```
(def facts initializari
  (numere 2 7 5 3 4)
  (max 2)
)

(defrule max1 "actualizeaza max"
  ?r<-(numere ?x $?rest)
  ?s<-(max ?y)
  (test (> ?x ?y))
=>
  (retract ?r ?s)
```

```

    (assert (max ?x) (numere $?rest))
  )

  (defrule max2 "max nu se schimba"
    ?r<-(numere ?x $?rest)
    (max ?y)
    (test (<= ?x ?y))
    =>
    (retract ?r)
    (assert (numere $?rest))
  )

```

Regula `max1` înlocuiește faptul `max` cu un altul dacă primul număr din secvență e mai mare decât vechiul maxim. Construcția `(test ...)` implementează o condiție prin evaluarea unei funcții predicat. În cazul regulilor de mai sus, aceste funcții realizează comparații. Un test este satisfăcut doar dacă expresia pe care o încorporează se evaluează la un simbol diferit de *false*, altfel testul nu este satisfăcut. O regulă care conține un test în partea stângă va fi activată doar dacă testul este satisfăcut împreună cu toate celelalte șabloane cuprinse în partea stângă. Elementul condițional `test` nu este propriu-zis un șablon pentru că lui nu îi corespunde un fapt din bază în situația în care condiția este satisfăcută. Regula `max2` verifică condiția complementară, situație în care lasă neschimbat vechiul maxim dar elimină un element din vector. În ambele reguli înlocuirea vectorului cu unul mai scurt cu un element se face prin retragerea faptului `(numere ...)` vechi și asertarea altuia nou care are toate elementele, cu excepția primului. În toate cazurile, variabila multi-câmp `$?rest` se leagă la restul elementelor vectorului.

Reguli ce se activează doar când altele nu se pot aplica

Ceea ce deranjează în această soluție este necesitatea de a preciza atât condiția `>` cât și pe cea complementară `≤`. Următoarea variantă de program elimină testul complementar, care este unul redundant:

```

(deffacts initializari
  (numere 2 7 5 3 4)
  (max 2)
)

(defrule max1
  (declare (salience 10))
  ?r<-(numere ?x $?rest)
  ?s<-(max ?y)
  (test (> ?x ?y))

```

```

=>
  (retract ?r ?s)
  (assert (max ?x) (numere $?rest))
)

(defrule max2
  ?r<-(numere ? $?rest)
  =>
  (retract ?r)
  (assert (numere $?rest))
)

```

Din regula `max2` a dispărut șablonul care identifică elementul `max` din bază cât și testul de \leq . Așadar, cum știm că `max2` se aplică exact în cazurile în care nu se poate aplica `max1`? Răspuns: făcând ca `max1` să fie mai prioritară decât `max2`.

Acesta este rostul declarației de *salience* din capul regulii `max1`. În CLIPS există 20.001 niveluri de prioritate, ce se întind în plaja $-10.000 \div 10.000$, cu certitudine mai multe decât ar putea vreodată să fie utilizate într-un program. O regulă fără declarație de *salience* are, implicit, prioritatea 0. Valorile în sine ale priorităților nu au importanță, ceea ce contează fiind doar prioritățile, unele în raport cu altele. Declarațiile de *salience* provoacă o sortare a regulilor activate la oricare pas. Făcând ca `max1` să fie mai prioritară decât `max2` forțăm ca `max2` să fie eliminată în faza de selecție care urmează filtrării, ori de câte ori se întâmplă să fie ambele active. Cum însă condițiile regulii `max2` sunt mai laxe decât cele ale regulii `max1`, ori de câte ori este activă `max1`, va fi activă și `max2`, iar prioritatea mai mare a lui `max1` va face ca aceasta să fie preferată întotdeauna lui `max2`. Rămâne că `max2` va putea fi aplicată numai când `max1` nu este activată, ceea ce ar corespunde exact condiției inverse, așa cum intenționăm.

Exemplul arată că, pentru a face ca o condiție de complementaritate să poată fi realizată printr-o declarație de prioritate, trebuie să facem ca regula cu prioritate mai mică să aibă o constrângere de aplicabilitate mai laxă decât a regulilor cu care intră în competiție. Numai în acest fel ne asigurăm că, în cazurile în care celelalte reguli nu se aplică, cea de prioritate mai mică încă își satisface condițiile pentru a putea fi aplicată. În cazul în care condiția aceasta nu e respectată, de exemplu când condiția implementată de regula mai puțin prioritară este disjunctă celei implementată de regula mai prioritară, atunci controlul prin declarații de prioritate este inutil, pentru că el poate fi realizat direct prin condiții și, ca urmare, trebuie evitat.

Întorcându-ne la exemplul nostru, condiția de terminare a iterației se realizează impunând ca faptul `numere` să mai conțină cel puțin o poziție (variabila `?x` din șablon). Doar acest lucru face ca cel puțin una din reguli să se aplice **atât** cât mai există măcar un număr neprocesat.

Revenind la o schemă logică imperativă

Un sistem bazat pe reguli e, în general, orientat spre alte tipuri de activități decât cele ce se pretează la o soluție imperativă. Am insistat suficient asupra acestui aspect în capitolul 1 al cărții. N-ar trebui însă să înțelegem prin aceasta că un sistem bazat pe reguli e incapabil să efectueze și tipuri de prelucrări specifice unui sistem imperativ. Să încercăm în această secțiune să înțelegem cum trebuie procedat dacă am dori să utilizăm un sistem bazat pe reguli pentru a implementa o secvență imperativă. Motivația acestei tentative este aceea că pe parcursul rezolvării unei probleme care necesită utilizarea mașinării bazate pe reguli poate să apară o subproblemă care necesită o soluție imperativă. Cum procedăm? Părăsim sistemul bazat pe reguli pentru a construi o procedură imperativă care să fie apelată din acesta? E și aceasta o posibilitate, dar uneori e mai comod să gândim în maniera bazată pe reguli însăși acea secvență.

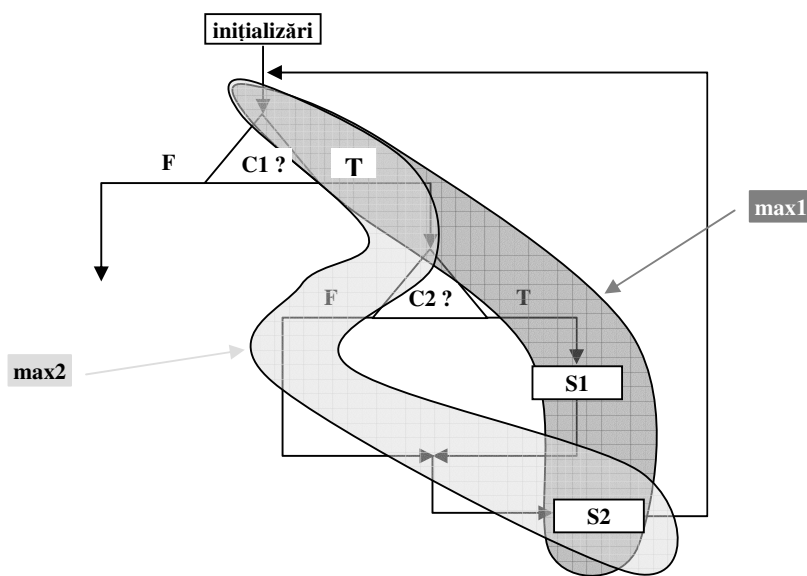


Figura 28: Decuparea de reguli dintr-o diagramă imperativă

Soluția imperativă a problemei de maxim este schițată în Figura 28. Aici, C1 este testul de terminare a iterației, C2 – cel care verifică $?x > ?y$, S1 reprezintă înlocuirea lui `max` din bază, iar S2 semnifică scurtarea vectorului de numere cu un

element. Decuparea acestui algoritm în cele două reguli este figurată în nuanțe diferite de gri. Într-adevăr `max1` și `max2` au în comun testul C1 cu ieșire *true* și secvența S2, în timp ce `max1` conține în plus secvența de actualizare a lui `max`. Inițializările sunt realizate direct de declarațiile `deffacts` de includere de fapte în bază iar terminarea iterației (ieșirea din testul C1 pe FALSE) se realizează automat în momentul în care nici una dintre regulile `max1` și `max2` nu-și mai satisfac condițiile, lucru care se întâmplă când faptul ce conține numerele a ajuns la forma: `(numere)`.

Cea de a treia soluție pe care o propunem izvorăște din observația că există o anumită redundanță a operațiunilor desfășurate în cele două reguli, manifestată în secvența S2. Preocuparea de a reduce redundanța, în cazul de față, poate părea exagerată, dar ea evidențiază o opțiune ce poate deveni necesitate în alte cazuri. Așadar să încercăm să reducem redundanța din cele două reguli.

Pentru aceasta va trebui să individualizăm într-o regulă separată operația comună, minisecvența S2 ce conține decrementarea vectorului de numere. Vom introduce o secvență `set-rest` care face acest lucru. Secvențierea regulilor devine acum cea din Figura 29, adică `max1` ori `max2` urmată de `set-rest`.

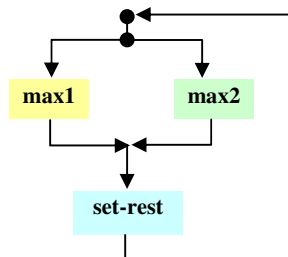


Figura 29: O altă diagramă de calcul al maximului

De data aceasta nu mai e posibil să umblăm la priorități pentru a impune o secvență, făcând de exemplu ca `set-rest` să aibă o prioritate mai mică atât decât `max1` cât și decât `max2`, pentru că stabilirea acestei secvențe nu se face în faza de selecție, ci ea ține de cicluri separate ale motorului. O soluție ar fi cea în care folosim un indicator (fanion), pe care `max1` și `max2` să-l seteze, iar `set-rest` să-l reseteze de fiecare dată, ca în Figura 30.

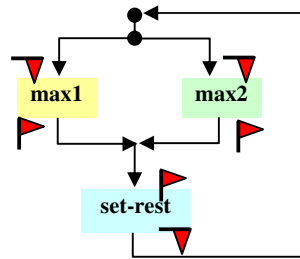


Figura 30: Utilizarea unui fanion în calculul maximului

În secvența CLIPS care urmează fanionul setat are valoarea 1 și resetat – 0.

```
(deffacts initializari
  (numere 2 7 5 3 4)
  (max 2)
  (fanion 0)
)

(defrule max1
  (declare (salience 10))
  (numere ?x $?rest)
  ?s <- (max ?y)
  (test (> ?x ?y))
  ?r <- (fanion 0)
  =>
  (retract ?s ?r)
  (assert (fanion 1) (max ?x))
)

(defrule max2
  ?r <- (flag 0)
  =>
  (retract ?r)
  (assert (fanion 1))
)

(defrule set-rest
  ?r <- (numere ?x $?rest)
  ?s <- (fanion 1)
  =>
  (retract ?r ?s)
  (assert (numere $?rest) (fanion 0))
)
```


În acest moment simțim nevoia să anunțăm un rezultat și altfel decât prin includerea în baza de fapte. Scrierea într-un fișier sau în terminal se face printr-un apel de forma:

```
(printout <dispozitiv-logic> <element-tipărit>*)
```

în care:

```
<dispozitiv-logic> ::= t | <nume-logic>
<element-tipărit> ::= <șir de caractere> | <variabilă>
| <constantă> | crlf
```

Simbolul `t` este folosit pentru scrierea în terminal. `crlf` este un simbol a cărui includere într-un apel al funcției de tipărire provoacă trecerea la un rând nou. Dacă se dorește ca ieșirea să apară într-un fișier, atunci fișierul trebuie deschis anterior printr-o comandă:

```
(open <nume-fișier> <nume-logic> "w")
```

După terminarea scrierii, fișierul trebuie închis prin comanda

```
(close <nume-logic>)
```

Cu acestea, anunțarea maximului, pentru oricare dintre soluțiile de mai sus, se poate face astfel:

```
(defrule tipareste-max
  (numere)
  (max ?max)
  =>
  (printout t "Maximul: " ?max crlf)
)
```


Capitolul 8

Constrângeri în confruntarea șabloanelor

În rularea oricărui program imaginabil pe un sistem de calcul există un cod și niște date. În general, într-o rulare clasică, procesarea datelor se face la inițiativa programului. Un fir de execuție principal, sau mai multe (în cazul unei execuții paralele) “cheamă” ori “apelează” alte componente de calcul, în funcție de nevoile dictate, la fiecare moment, de date. Componenta activă este programul și cea pasivă sunt datele. Într-o rulare bazată pe reguli rolurile se inversează: practic datele își cheamă codul ce le poate prelucra. E ca și cum regulile (ce formează corpul executabil) sunt “flămânde” să prelucreze faptele din bază, dar rămân inactive până ce apar date care să corespundă abilităților de prelucrare pentru care ele au fost concepute. Părțile drepte ale regulilor, ce formează componenta de prelucrare a sistemului, nu se activează decât atunci când părțile stângi găsesc exact acele date ce convin. Părțile stângi ale regulilor sunt “înghețate” în rețelele de șabloane, în timp ce modificări în bază, dictate de date, se propagă prin aceste rețele.

Nu e surprinzător, așadar, ca necesitatea de a depista date ce convin regulilor să fi impus dezvoltarea unor mecanisme de regăsire a datelor cât mai expresive. Cred, de aceea, că expresivitatea unui limbaj de programare bazat pe reguli trebuie apreciată în conformitate cu facilitățile pe care le oferă în descrierea șabloanelor. În acest capitol vom aprofunda descrierea de șablon pe care o oferă limbajul CLIPS, construind totodată o mică aplicație care să etaleze o seamă de necesități ce pot apărea în confruntarea șabloanelor regulilor cu faptele din bază.

Șabloanele sunt constituite din simboluri, numere și variabile (acestea fiind simple ori multi-câmp). Am văzut până acum cum putem indica acele părți constante, riguros exacte, pe care le dorim în faptele căutate, dar și cum putem lăsa elemente ale faptelor să nu fie supuse nici unor restricții, așadar să fie lăsate libere. Există însă o plajă întreagă de **constrângeri asupra câmpurilor** care să acopere plaja de la o valoare complet specificată la o valoare total liberă.

8.1. Interogări asupra unei baze de date

Fideli principiului că un limbaj se deprinde cel mai bine când avem în intenție construirea unei aplicații, *vom completa o mică bază cu informații despre*

niște piese de lego (reținând, de exemplu, un identificator al pieselor, forma lor, materialul din care sunt făcute – plastic sau lemn – suprafața, grosimea și culoarea lor), pentru ca, asupra ei, să putem organiza apoi interogări prin care să aflăm:

- toate piesele de altă culoare decât galben;
 - toate piesele de culoare fie roșu fie galben;
 - toate piesele care au grosimea mai strict mică de 5 unități (oricare ar fi ele), dar diferită de 2,
- dar să realizăm și interogări ceva mai complicate:
- toate piesele roșii din plastic și verzi din lemn;
 - toate piesele din plastic cu aceeași arie ca a unor piese din lemn dar mai subțiri decât acestea;
 - toate piesele de aceeași grosime, pe grupe de grosimi;
 - cele mai groase piese dintre cele de aceeași arie.

Cum informațiile pe care trebuie să le memorăm asupra pieselor cuprind o seamă de trăsături diferite, însă aceleași pentru toate, o reprezentare obiectuală a acestora este cea mai firească aici. CLIPS permite descrierea structurilor de obiecte prin declarații de *template* combinate cu instanțierea acestora ca fapte. Astfel, în mini-aplicația noastră vom reprezenta piesele ca obiecte lego cu câmpurile *id* (pentru identificator), *forma*, *material*, *arie*, *grosime* și *culoare*:

```
(deftemplate lego
  (slot id)
  (slot forma)
  (slot material)
  (slot arie)
  (slot grosime)
  (slot culoare)
)

(deffacts piese-lego
  (lego (id lego1) (forma cerc) (material plastic)
        (arie 10) (grosime 5) (culoare galben))
  (lego (id lego2) (forma cerc) (material plastic)
        (arie 10) (grosime 4) (culoare albastru))
  ...
)
```

Din punct de vedere sintactic, constrângerile se descriu cu ajutorul unor operatori ce se aplică direct valorilor unor câmpuri sau variabilelor ce stau pe poziția unor câmpuri în șabloane.

■ **Constrângerea NOT** se realizează cu semnul tilda (~) plasat în fața unui

literal constant sau a unei variabile. Dacă acest câmp se potrivește cu un câmp din fapt, atunci constrângerea NOT eșuează iar dacă elementul prefixat cu ~ nu se potrivește peste valoarea din fapt, atunci constrângerea reușește.

Utilizând constrângerea NOT putem răspunde la întrebarea *toate piesele de altă culoare decât galben* scriind un șablon:

```
(lego (culoare ~galben))
```

În plus, în cazul în care culoarea din acel câmp trebuie folosită undeva mai departe în regulă, putem împerechea un nume de variabilă cu constrângerea NOT:

```
(lego (culoare ?x&:~galben))
```

Simbolul **constrângerii OR** este bara verticală (|) și ea combină două constrângeri într-o disjuncție. Rezultatul confruntării reușește dacă cel puțin o constrângere dintre cele pe care le grupează reușește.

Toate piesele de culoare fie roșu fie galben, sunt găsite de un șablon:

```
(lego (culoare rosu|galben))
```

Simbolul **constrângerii AND** este *ampersand* (&) și el combină două constrângeri într-o conjuncție. Rezultatul confruntării reușește dacă ambele constrângeri pe care le grupează astfel reușesc.

De exemplu, pentru a căuta *piesele care au grosimea mai mică strict de 5, dar diferită de 2*, vom scrie:

```
(grosime ?x&:(< ?x 5)&~2)
```

Orice **combinare a constrângerilor** folosind ~, & și | poate fi realizată. Dacă în condiție apar variabile, ele vor fi legate numai dacă apar pe prima poziție în combinație. În toate cazurile în care o variabilă apare în interiorul unei constrângeri compuse, ea trebuie să fi fost anterior legată.

De exemplu, atunci când căutăm piese de grosime fie mai mică sau egală cu 3, fie egală cu 5, vom scrie:

```
(grosime ?x&:(<= ?x 3)|5)
```

În exemplele de mai sus s-au folosit câteva **funcții predicat** în scrierea constrângerilor. Iată o listă (incompletă) din care programatorul poate să aleagă: <, <=, <>, =, >, >=, and, or, not, eq, neq, evenp, oddp, integerp, floatp, stringp, symbolp etc.

Trecând la problemele mai pretențioase, o primă tentativă de a rezolva cererea *toate piesele roșii din plastic și verzi din lemn* duce la o buclă infinită:

```
(defrule selectie1-v1
; toate piesele din plastic rosii si din lemn verzi
  (or (lego (id ?i) (material plastic) (culoare rosu))
      (lego (id ?i) (material lemn) (culoare verde)))
  ?sel <- (selectie $?lis)
  =>
  (retract ?sel)
  (assert (selectie $?lis ?i))
)
```

Motivul, așa cum ne putem lesne da seama, este modificarea faptului *selectie* la fiecare aplicare de regulă, pentru că o modificare a unui fapt se face întotdeauna prin ștergerea lui și includerea în bază a unui fapt nou, deci cu un index nou. Acest lucru face ca perechea formată dintr-un obiect *lego* și faptul *selectie*, ce determină o activare a regulii, să fie mereu alta în orice fază de filtrare și, drept urmare, refractabilitatea să nu dea roade.

Putem repara această hibă în mai multe moduri. Cea mai simplă cale e să eliminăm obiectele din bază imediat după ce le-am folosit, ca în varianta de regulă de mai jos:

```
(defrule selectie1-v2
; toate piesele din plastic rosii si din lemn verzi
  (or ?l <- (lego (id ?i) (material plastic) (culoare
rosu))
      ?l <- (lego (id ?i) (material lemn) (culoare verde)))
  ?sel <- (selectie $?lis)
  =>
  (retract ?l ?sel)
  (assert (selectie $?lis ?i))
)
```

Exemplul mai pune în evidență și **elementul condițional OR**, ca și faptul că, sub el, pot avea loc inclusiv legări de variabile la indecși de fapte. Cum doar una dintre clauzele disjuncției va fi în final satisfăcută, e suficientă utilizarea unei singure variabile ce va fi legată la primul fapt, în ordinea din disjuncție, care se potrivește cu unul dintre șabloane. Pentru că regula se aplică de mai multe ori, până la urmă toate obiectele ce satisfac una sau alta dintre condițiile disjuncției vor fi procesate și, în final, eliminate.

O abordare de acest gen nu este întotdeauna agreată pentru că afectează baza în mod neuniform: în urma procesării unele obiecte vor fi dispărut. Ca urmare, ar trebui să refacem baza dacă dorim să o interogăm în continuare. Putem însă lesne

face ca baza să rămână intactă după interogare: introducem un test care să împiedice selectarea unui obiect ce se află deja în lista selectie:

```
(defrule selectie1-v3
; toate piesele din plastic rosii si din lemn verzi
  (or (lego (id ?i) (material plastic) (culoare rosu))
      (lego (id ?i) (material lemn) (culoare verde)))
  ?sel <- (selectie $?lis)
  (test (not (member$ ?i $?lis)))
=>
  (retract ?sel)
  (assert (selectie $?lis ?i))
)
```

În această regulă `test` este un element de realizare a condițiilor, ce poate fi utilizat în părțile stângi ale regulilor pentru a evalua expresii în care apar funcții predicat. Predicatul multi-câmp `member$`, verifică incluziunea primului argument în lista de câmpuri dată de cel de al doilea argument.

Să încercăm acum o soluție la cererea *toate piesele din plastic cu aceeași arie ca a unor piese din lemn, dar mai subțiri decât acestea*.

```
(defrule selectie2
; toate piesele din plastic cu aceeasi arie ca a unor
piese din
; lemn dar mai subtiri
  (lego (material lemn) (arie ?a) (grosime ?g1))
  (lego (id ?i) (material plastic) (arie ?a)
      (grosime ?g2&:(< ?g2 ?g1)))
  ?sel <- (selectie $?lis)
  (test (not (member$ ?i $?lis)))
=>
  (retract ?sel)
  (assert (selectie $?lis ?i))
)
```

Ca să răspundem la întrebarea *toate piesele de aceeași grosime, pe grupe de grosimi*, avem nevoie de două reguli. Prima construiește grupe de grosimi, iar a doua include piesele în grupe. Nu e nevoie ca aplicarea lor să fie dirijată în vreun fel. Faptele care descriu piesele pot fi considerate în orice ordine, în așa fel încât, ori de câte ori o piesă de o grosime pentru care nu există încă o grupă este luată în considerare, grupa acesteia să fie construită, pentru ca, ulterior, o piesă având aceeași grosime cu a unei grupe să fie inclusă în grupă:

```
(defrule selectie3-init-grupa
```

```

; initializeaza o grupa de grosimi
(lego (id ?i) (grosime ?g1))
(not (gros ?g1 $?))
=>
(assert (gros ?g1 ?i))
)

(defrule selectie3-includ-in-grupa
; include un obiect in grupa corespunzatoare
(lego (id ?i) (grosime ?g1))
?g <- (gros ?g1 $?lista)
(test (not (member$ ?i $?lista)))
=>
(retract ?g)
(assert (gros ?g1 $?lista ?i))
)

```

În sfârșit, pentru interogarea *cele mai groase piese dintre cele de aceeași arie*, următoarea variantă rezolvă doar parțial problema, pentru că întoarce câte un singur obiect care satisface condiția de a fi cel mai mare dintre cele de aceeași arie:

```

(defrule selectie4
; initializeaza o grupa de arii
(lego (id ?i) (arie ?a1))
(not (arie ?a1 $?))
=>
(assert (arie ?a1 ?i))
)

(defrule selectie4-includ-in-grupa
; include un obiect in grupa corespunzatoare
; daca e mai gros decit cel existent deja
(lego (id ?i) (arie ?a1) (grosime ?g1))
?a <- (arie ?a1 ?j)
(lego (id ?j) (grosime ?g2&:(> ?g1 ?g2)))
=>
(retract ?a)
(assert (arie ?a1 ?i))
)

```

Varianta următoare găsește toate obiectele care se presupune că au aceeași grosime maximă între cele de aceeași arie:

```

(defrule selectie4-init
; initializeaza o grupa de arii
(lego (id ?i) (arie ?a1))

```



```

    (not (arie ?a1 $?))
    =>
    (assert (arie ?a1 ?i))
  )

  (defrule selectie4-inlocuieste-in-grupa
    ; initializeaza o grupa cu un obiect daca are aria
    corespunzatoare
    ; grupei si daca e mai gros decit oricare dintre cele
    existente deja ; in grupa
    (lego (id ?i) (arie ?a1) (grosime ?g1))
    ?a <- (arie ?a1 $?list)
    (lego (id ?j) (grosime ?g2&:(> ?g1 ?g2)))
    (test (member$ ?j $?list))
    =>
    (retract ?a)
    (assert (arie ?a1 ?i))
  )

  (defrule selectie4-includ-in-grupa
    ; include un obiect in grupa corespunzatoare
    ; daca e la fel de gros decit oricare dintre cele
    existente deja
    (lego (id ?i) (arie ?a1) (grosime ?g1))
    ?a <- (arie ?a1 $?list)
    (test (not (member$ ?i $?list)))
    (lego (id ?j) (grosime ?g1))
    (test (member$ ?j $?list))
    =>
    (retract ?a)
    (assert (arie ?a1 $?list ?i))
  )

```

8.2. Un exemplu de sortare

Problema care urmează tratează un aspect des întâlnit în programare: sortarea listelor. Pretextul va fi *o mică aplicație ce-și propune să acorde gradații de merit profesorilor dintr-un liceu. Există mai multe criterii după care aceste gradații se acordă: gradul didactic, numărul de elevi îndrumați de profesor și care au participat la olimpiade, numărul de publicații ale cadrului didactic, vechimea lui etc. Se cunosc numerele de puncte ce se acordă pentru fiecare criteriu în parte. Datele despre personalul didactic sunt înregistrate într-o bază de date. Aplicația noastră va trebui să ordoneze profesorii în vederea acordării gradațiilor de merit, prin aplicarea acestor criterii.*

Aplicația începe prin definiția cadrelor de fapte care vor memora numele profesorilor și datele ce se cunosc despre ei. Aceste date vor fi folosite pentru aplicarea criteriilor de acordare a gradațiilor de merit. Exemplul nostru folosește patru profesori – ai căror parametri sunt memorați imediat după definițiile de *template*:

```
(deftemplate persoana
  (slot nume)
  (slot sex)
  (slot varsta)
  (slot vechime)
  (slot grad)
  (slot nr-ore)
  (multislot olimpiade)
  (slot publicatii)
)

(deffacts personal
  (persoana (nume Ionescu) (sex b) (varsta 30) (vechime
5) (grad 2) (nr-ore 18) (olimpiade 1999 3 1998 5 1997
2) (publicatii 2))
  (persoana (nume Popescu) (sex f) (varsta 35) (vechime
10) (grad 1) (nr-ore 17) (olimpiade 1999 4 1998 5 1997
3) (publicatii 1))
  (persoana (nume Georgescu) (sex f) (varsta 32) (vechime
7) (grad 2) (nr-ore 18) (olimpiade 1999 5 1998 6 1997
1) (publicatii 0))
  (persoana (nume Isopescu) (sex f) (varsta 40) (vechime
25) (grad 0) (nr-ore 20) (olimpiade 1999 1 1998 1 1997
1) (publicatii 0))
)
```

Se definește *template*-ul faptelor care vor înregistra punctajele persoanelor. Apoi aceste fapte sunt inițializate la valorile 0 ale punctajului:

```
(deftemplate punctaj
  (slot nume)
  (slot valoare)
)

(deffacts puncte
  (punctaj (nume Ionescu) (valoare 0))
  (punctaj (nume Popescu) (valoare 0))
  (punctaj (nume Georgescu) (valoare 0))
  (punctaj (nume Isopescu) (valoare 0))
)
```

Primele reguli calculează punctajele corespunzătoare activităților profesorilor. Regula `criteriu-grad` aplică criteriul grad didactic. La fiecare aplicare a acestei reguli într-un fapt punctaj al unei persoane se modifică câmpul `valoare` prin adăugarea unui număr de puncte corespunzător gradului didactic – indicat prin comentarii în antetul regulii. Pentru prima dată în această regulă avem un exemplu de folosire a comenzii `if`:

```
(defrule criteriu-grad
; doctor = 50 puncte
; grad 1 = 30 puncte
; grad 2 = 20 puncte
; definitivat = 10 puncte
; grad 0 = 0 puncte
  ?per <- (persoana (nume ?num) (grad ?grd&~folosit))
  ?pct <- (punctaj (nume ?num) (valoare ?val))
  =>
  (modify ?per (grad folosit))
  (if (eq ?grd doctor) then (modify ?pct (valoare =(+
?val 50)))
    else (if (eq ?grd 1) then (modify ?pct (valoare =(+
?val 30)))
      else (if (eq ?grd 2) then (modify ?pct (valoare =(+
?val 20)))
        else (if (eq ?grd definitivat) then (modify ?pct
(valoare =(+ ?val 10))))))
  )
```

Regula `criteriu-olimpiade` se aplică de un număr de ori egal cu numărul de profesori multiplicat cu numărul de olimpiade în care aceștia au avut elevi:

```
(defrule criteriu-olimpiade
; fiecare elev la olimpiada in ultimii 3 ani aduce 5
; puncte
  ?per <- (persoana (nume ?num)
              (olimpiade ?an ?elevi $?rest))
  ?pct <- (punctaj (nume ?num) (valoare ?val))
  =>
  (modify ?per (olimpiade $?rest))
  (modify ?pct (valoare =(+ ?val (* 5 ?elevi))))
  )
```

Regula criteriu-vechime se aplică câte o dată pentru fiecare persoană. Conform criteriului de vechime, la fiecare aplicare valoarea punctajului persoanei este incrementată cu un număr egal cu numărul anilor de vechime:

```
(defrule criteriu-vechime
; fiecare an de vechime aduce 1 punct
  ?per <- (persoana (nume ?num) (vechime ?ani&~folosit))
  ?pct <- (punctaj (nume ?num) (valoare ?val))
  =>
  (modify ?per (vechime folosit))
  (modify ?pct (valoare =(+ ?val (* 1 ?ani))))
)
```

Regula criteriu-publicatii se aplică din nou câte o dată pentru fiecare persoană. Conform criteriului publicațiilor, la fiecare aplicare valoarea punctajului persoanei este incrementată cu de două ori numărul publicațiilor acesteia:

```
(defrule criteriu-publicatii
; fiecare publicatie aduce 2 puncte
  ?per <- (persoana (nume ?num)
                  (publicatii ?publ&~folosit))
  ?pct <- (punctaj (nume ?num) (valoare ?val))
  =>
  (modify ?per (publicatii folosit))
  (modify ?pct (valoare =(+ ?val (* 2 ?publ))))
)
```

Și regula criteriu-nr-ore are un număr de aplicații egal cu numărul persoanelor. La fiecare aplicare valoarea punctajului persoanei este incrementată cu jumătate din numărul orelor prestate de aceasta săptămânal:

```
(defrule criteriu-nr-ore
; fiecare ora pe saptamina aduce 0.5 puncte
  ?per <- (persoana (nume ?num) (nr-ore ?ore&~folosit))
  ?pct <- (punctaj (nume ?num) (valoare ?val))
  =>
  (modify ?per (nr-ore folosit))
  (modify ?pct (valoare =(+ ?val (* 0.5 ?ore))))
)
```

Urmează trei reguli de sortare. Prioritățile acestor trei reguli sunt, toate, -100, deci mai mici decât ale regulilor de aplicare a criteriilor, date mai sus. Ca urmare, aceste trei reguli se vor aplica numai în cazul în care nici o regulă din cele de mai

sus nu mai poate fi aplicată. Acest lucru se întâmplă numai când toate câmpurile faptelor `persoană` sunt folosite deja. Cititorul a observat, cu siguranță, că am marcat faptul că s-a lucrat asupra unui câmp al unui fapt `persoana`, prin înregistrarea valorii folosit în acest câmp.

Regula `sortare-initializare` se aplică o singură dată, la începutul sortării persoanelor, și ea va introduce un fapt `lista-finală` – inițial vid. O pereche, formată dintr-un nume de profesor ales la întâmplare și valoarea punctajului său, va intra apoi în această listă. Faptul `punctaj` corespunzător este retras pentru a nu mai fi luat încă o dată în considerare:

```
(defrule sortare-initializare
; initializeaza lista finala in care persoanele vor fi
; asezate in ordinea descrescatoare a punctajelor
  (declare (salience -100))
  (not (lista-finala $?))
  ?pct <- (punctaj (nume ?num) (valoare ?val))
  =>
  (assert (lista-finala ?num ?val))
  (retract ?pct)
)
```

Vom sorta `lista-finală` în ordinea descrescătoare a punctajelor calculate. Regula `sortare-extrema-stinga` tratează cazul în care o valoare de punctaj a unui profesor este mai mare decât cea mai mare valoare înscrisă în listă, adică cea din extrema stângă a listei. Perechea formată din numele persoanei și valoarea punctajului său este inclusă acolo:

```
(defrule sortare-extrema-stinga
; plaseaza o persoana in capul listei
  (declare (salience -100))
  ?lst <- (lista-finala ?prim-nume ?prim-val $?ultime)
  ?pct <- (punctaj (nume ?num)
                (valoare ?val&:(> ?val ?prim-val)))
  =>
  (assert (lista-finala ?num ?val
                        ?prim-nume ?prim-val $?ultime))
  (retract ?lst ?pct)
)
```

Regula `sortare-extrema-dreapta` tratează cazul în care o valoare de punctaj a unui profesor este mai mică decât cea mai mică valoare înscrisă în listă, adică cea din extrema dreaptă:

```
(defrule sortare-extrema-dreapta
```

```

; plaseaza o persoana in coada listei
(declare (salience -100))
?lst <- (lista-finala $?prime ?ultim-nume ?ultim-val)
?pct <- (punctaj (nume ?num)
           (valoare ?val&:(< ?val ?ultim-val)))
=>
(assert (lista-finala $?prime ?ultim-nume ?ultim-val
                        ?num ?val))
(retract ?lst ?pct)
)

```

Ultimul caz tratat este cel al unei valori care trebuie plasată între două valori aflate în listă:

```

(defrule sortare-mijloc
; plaseaza o persoana intre alte doua persoane in lista
(declare (salience -100))
?lst <- (lista-finala $?prime
           ?un-nume
           ?o-val&:(numberp ?o-val)
           ?alt-nume
           ?alt-val
           $?ultime)

?pct <- (punctaj
        (nume ?num)
        (valoare ?val&:(and (< ?val ?o-val)
                             (> ?val ?alt-val)))))

=>
(assert (lista-finala $?prime
                    ?un-nume ?o-val
                    ?num ?val
                    ?alt-nume ?alt-val
                    $?ultime))
(retract ?lst ?pct)
)

```

Capitolul 9

Despre controlul execuției

9.1. Criterii utilizate în ordonarea agendei

Agenda este o structură folosită de motorul CLIPS pentru păstrarea informațiilor privitoare la activările regulilor de la un ciclu al motorului la următorul. Ea este o listă ordonată de **instanțe** (sau **activări**) **de reguli**. Plasarea unei activări de regulă în această listă se face după următoarele criterii, în ordine:

- regulile nou activate sunt plasate deasupra tuturor celor de o prioritate mai mică și dedesubtul tuturor celor de o prioritate mai mare;
- printre regulile de egală prioritate, plasarea se bazează pe strategia de rezoluție a conflictelor activată implicit sau explicit;
- dacă primele două criterii nu reușesc să stabilească o ordine totală între noile reguli activate de aceeași introducere ori retragere a unui fapt, atunci plasarea se face arbitrar (și nu aleator), adică este dependentă de implementare. Programatorul este sfătuit să nu se bazeze pe observații empirice asupra manierei de ordonare a agendei.

La momentul execuției, activarea aflată pe prima poziție în agendă va fi cea executată. Orice modificare ce se produce în agendă între două momente de execuție nu influențează în nici un fel rularea. O schiță a activităților motorului ce sunt amorate de comenzi este dată în Figura 31.

După cum se constată, primul criteriu al ordonării agendei în vederea selecției îl constituie prioritatea regulilor și abia al doilea este strategia de rezoluție. Programatorul trebuie să deprindă exploatarea corectă a priorității. Declarații de prioritate au fost utilizate încă din capitolul precedent. În acest capitol, cât și în cel ce urmează, motivația principală în prezentarea exemplelor va fi centrată pe utilizarea declarațiilor de prioritate. În primul exemplu vom utiliza însă o singură regulă, ceea ce face ca o declarație de prioritate să nu-și aibă sens. Trasarea rulării ne va ajuta să urmărim evoluția agendei.

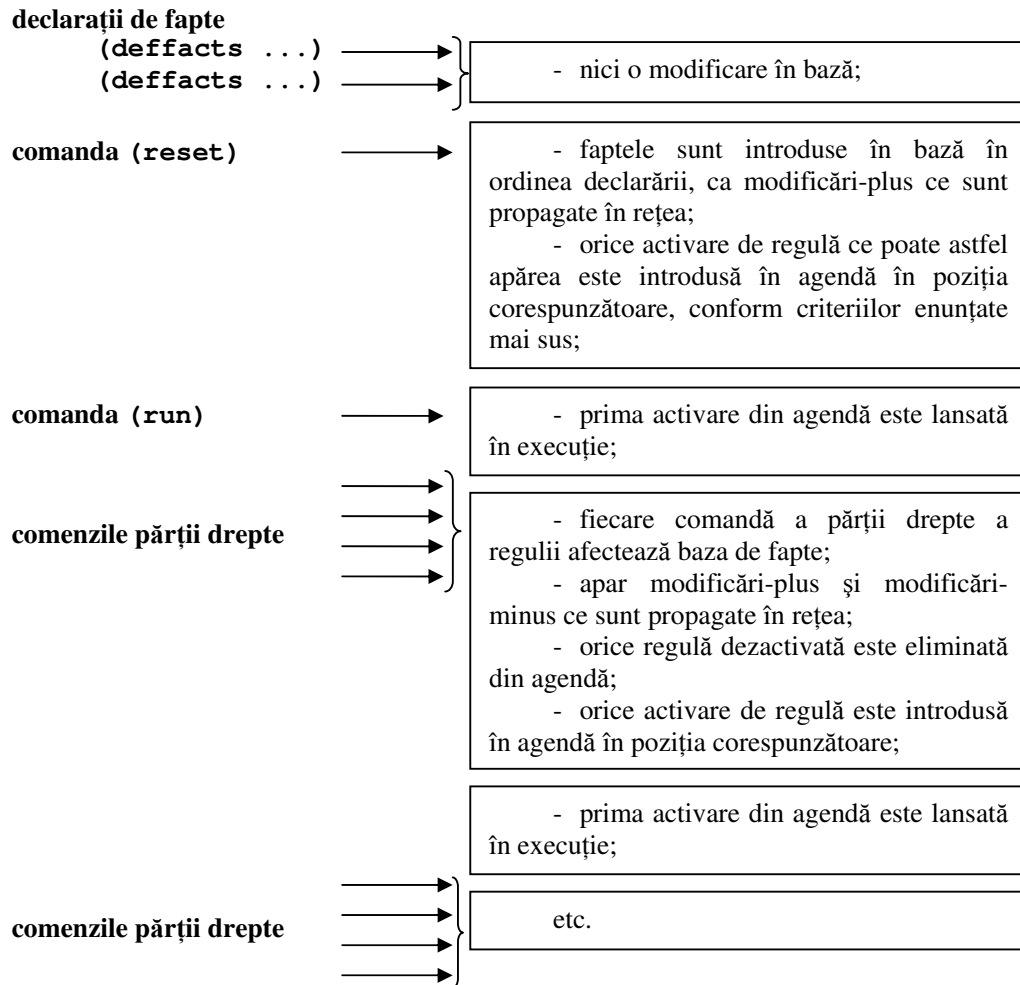


Figura 31: Activitățile motorului

Exemplele care urmează evidențiază diferite maniere de control al execuției. În primul, o declarație de prioritate a regulilor lipsește cu desăvârșire. Și totuși vom vedea că putem influența controlul prin ordinea comenzilor sau modificarea strategiei.

9.2. Urmărirea execuției

Să presupunem că avem o mini-bază de date cu prețul pe kilogram al unor fructe:

```
(deffacts preturi-fructe
  (pret mere 10000)
  (pret pere 18000)
  (pret struguri 24000)
  ...
)
```

Să mai presupunem că plecăm de acasă cu o listă a fructelor și a cantităților pe care dorim să le cumpărăm:

```
(deffacts de-cumparat
  (cumpar mere 5)
  (cumpar pere 3)
  (cumpar struguri 2)
)
```

și că o altă secțiune a datelor conține o variabilă ce ne va da, la întoarcerea acasă, cât am cheltuit:

```
(deffacts buzunar
  (suma 0)
)
```

Dorim ca valoarea sumei să fie actualizată corespunzător cumpărăturilor efectuate. Pentru a face lucrurile cât mai simple, vom admite că fondul de bani de care dispunem este nelimitat. Atunci, tot ce avem de făcut este să lăsăm să ruleze o regulă de felul:

```
(defrule cumpar_un_fruct
  ?c <- (cumpar ?fruct ?kg)
  (pret ?fruct ?lei)
  ?s <- (suma ?suma)
  =>
  (retract ?f ?s)
  (assert (suma =(+ ?suma (* ?kg ?lei))))
)
```

La fiecare aplicare, un fapt cumpar dispare din bază iar valoarea cheltuită pentru fructele corespunzătoare este adunată la suma deja cheltuită.

Mediul CLIPS are mai multe metode de a urmări o execuție. Una din ele este invocarea comenzii (`watch <parametri>`). Ea poate primi ca parametri: `activations`, `facts`, `rules`, `statistics` etc. Vom urmări efectul dării acestei comenzi înainte de rularea programului nostru pentru a înțelege mai bine maniera în care regulile se activează și faptele sunt retrase ori incluse în bază în cursul execuției. La încărcarea programului printr-o comandă (`load . . .`) sau la definirea la *prompter* a faptelor, acestea nu sunt încă incluse în bază. După cum arată și Figura 31, acest lucru se realizează o dată cu execuția comenzii (`reset`).

```
CLIPS> (reset)
==> f-0      (initial-fact)
==> f-1      (pret mere 10000)
==> f-2      (pret pere 18000)
==> f-3      (pret struguri 24000)
==> f-4      (cumpar mere 8)
==> f-5      (cumpar pere 3)
==> f-6      (cumpar struguri 2)
==> f-7      (suma 0)
```

Săgețile spre dreapta indică faptele introduse în bază. Faptul (`initial-fact`) are întotdeauna indexul `f-0`. Faptele incluse de program sunt cele cu indecșii de la `f-1` la `f-7`.

În continuare, trasarea indică trei activări de reguli, în fapt trei instanțe ale singurei reguli definite, `cumpar-un-fruct`:

```
==> Activation 0      cumpar-un-fruct: f-6, f-3, f-7
==> Activation 0      cumpar-un-fruct: f-5, f-2, f-7
==> Activation 0      cumpar-un-fruct: f-4, f-1, f-7
```

Indecșii care urmează numelui regulii sunt faptele care sunt satisfăcute de cele trei șabloane ale regulii, primul de tip (`cumpar...`), al doilea de tip (`pret...`) și al treilea – singurul fapt (`suma...`):

```
(defrule cumpar_un_fruct
  ?c <- (cumpar ?fruct ?kg)
  (pret ?fruct ?lei)
  ?s <- (suma ?suma)
  =>
  ...
)
```

Rezultă că prima activare corespunde configurației de fapte:

```
f-6      (cumpar struguri 2)
f-3      (pret struguri 24000)
f-7      (suma 0)
```

a doua – configurației:

```
f-5      (cumpar pere 3)
f-2      (pret pere 18000)
f-7      (suma 0)
```

iar a treia – configurației:

```
f-4      (cumpar mere 8)
f-1      (pret mere 10000)
f-7      (suma 0)
```

Dacă am considera o altă ordine de apariție a faptelor în bază decât cea dată, de exemplu plasând declarația de sumă înaintea declarațiilor fructelor:

```
(deffacts buzunar
  (suma 0)
)

(deffacts de-cumparat
  (cumpar mere 5)
  (cumpar pere 3)
  (cumpar struguri 2)
)
```

constatăm o ușoară modificare a trasării, în conformitate cu schema de propagare a modificărilor dată de algoritmul RETE, care este reluată schematizat în Figura 31:

```
CLIPS> (reset)
==> f-0      (initial-fact)
==> f-1      (suma 0)
==> f-2      (pret mere 10000)
==> f-3      (pret pere 18000)
==> f-4      (pret struguri 24000)
==> f-5      (cumpar mere 8)
==> Activation 0      cumpar-un-fruct: f-5,f-2,f-1
==> f-6      (cumpar pere 3)
==> Activation 0      cumpar-un-fruct: f-6,f-3,f-1
==> f-7      (cumpar struguri 2)
==> Activation 0      cumpar-un-fruct: f-7,f-4,f-1
```

CLIPS>

Întrucât faptele `suma` și `pret` sunt deja introduse, la fiecare apariție a unui fapt `cumpar`, o nouă activare apare în agendă.

Să presupunem, în continuare, că revenim la ordinea inițială de declarare a faptelor. Rularea, amorțată de comanda `(run)`, decurge astfel:

```
CLIPS> (run)
FIRE      1 cumpar-un-fruct: f-4,f-1,f-7
<== f-4      (cumpar mere 8)
<== f-7      (suma 0)
<== Activation 0      cumpar-un-fruct: f-5,f-2,f-7
<== Activation 0      cumpar-un-fruct: f-6,f-3,f-7
==> f-8      (suma 80000)
==> Activation 0      cumpar-un-fruct: f-6,f-3,f-8
==> Activation 0      cumpar-un-fruct: f-5,f-2,f-8
```

Trasarea indică prima “aprindere” a regulii `cumpar-un-fruct` datorită potrivirii configurației de fapte: `f-4, f-1, f-7` cu partea de condiții a regulii. Aprinderea în ordine inversă față de ordinea includerilor în agendă se datorează utilizării strategiei **depth**, care este implicită în lipsa unei declarații explicite de strategie. Dar asupra strategiilor utilizate în CLIPS vom reveni imediat.

În continuare, trasarea indică evenimente induse de executarea părții de acțiuni a regulii `cumpar_un_fru`:

```
...
=>
(retract ?f ?s)
(assert (suma =(+ ?suma (* ?kg ?lei))))
)
```

Săgețile spre stânga indică retrageri de fapte din bază și disparițiile unor activări din agendă. Așadar aprinderea instanței `f-4, f-1, f-7` a regulii `cumpar-un-fruct` duce la eliminarea faptelor cu indecșii `f-4` și `f-7`. Totodată, cum faptul `f-7` intra în condițiile ambelor instanțe rămase, dispariția lui din bază este resimțită imediat în agendă prin dezactivarea acestor instanțe.

Rezultatul executării comenzii `(assert ...)` este apariția unui nou fapt de tip sumă, de index `f-8`: `(suma 80000)`. Apariția acestui fapt, corelată cu existența în bază a celorlalte fapte, duce la activarea a două noi instanțe ale regulii `cumpar-un-fruct`, respectiv datorate secvențelor de fapte `f-6, f-3, f-8` și `f-5, f-2, f-8`.

Rularea continuă astfel:

```

FIRE      2 cumpar-un-fruct: f-5,f-2,f-8
<== f-5      (cumpar pere 3)
<== f-8      (suma 80000)
<== Activation 0      cumpar-un-fruct: f-6,f-3,f-8
==> f-9      (suma 134000)
==> Activation 0      cumpar-un-fruct: f-6,f-3,f-9
FIRE      3 cumpar-un-fruct: f-6,f-3,f-9
<== f-6      (cumpar struguri 2)
<== f-9      (suma 134000)
==> f-10     (suma 182000)

```

Dacă efectuarea unor statistici asupra rulării, printr-o comandă (`watch statistics`), a fost, de asemenea, invocată, un raport final încheie trasarea. El indică numărul total de reguli aprinse, timpul total de execuție în secunde, o estimare a numărului mediu de reguli aprinse pe secundă etc.

```

3 rules fired      Run time is 0.1850000000013096
seconds.
16.21621621610141 rules per second.
7 mean number of facts (8 maximum).
1 mean number of instances (1 maximum).
2 mean number of activations (3 maximum).
CLIPS>

```

9.3. Strategii de rezoluție a conflictelor

Atunci când, după faza de filtrare, agenda – structura responsabilă cu memorarea instanțelor regulilor potențial active – conține mai mult decât o singură regulă, iar declarațiile de prioritate nu sunt capabile să introducă o ordonare totală a regulilor candidate la a fi aprinse, selecția trebuie să decidă asupra uneia, care să fie apoi executată. Acest lucru se poate realiza simplu prin alegerea primei reguli găsite. De cele mai multe ori însă se preferă să se facă uz de anumite criterii – strategii de rezoluție a conflictelor – care ar mări șansele de găsire a soluției.

Prima strategie este mai mult un principiu, pentru că nu introduce o ordonare a instanțelor regulilor din agendă ci își propune să împiedice intrarea motorului de inferențe în bucle infinite, prin repetarea la nesfârșit a acelorași secvențe de reguli aplicate asupra acelorași fapte. Ea a fost amintită deja în capitolul 3 unde am vorbit despre fazele motorului. Este vorba despre principiul refractabilității, care nu permite includerea în agendă a acelor instanțe ce au fost deja procesate în cicluri anterioare ale rulării. *Shell*-urile consacrate de sisteme bazate pe reguli au această strategie “cablată”.

CLIPS implementează câteva strategii de rezoluție a conflictelor. Strategia implicită este cea în adâncime (*depth strategy*), dar ea poate fi schimbată prin comanda `setstrategy` (care reordonează agenda în conformitate):

Strategia în adâncime (*depth strategy*): regulile nou activate sunt plasate în agendă în fața celor mai vechi de aceeași prioritate. Este strategia implicită în CLIPS.

Strategia în lărgime (*breadth strategy*): regulile nou activate sunt plasate în agendă în spatele celor mai vechi de aceeași prioritate.

Strategia complexității (*complexity strategy*): regulile nou activate sunt plasate în agendă în fața tuturor activărilor de reguli de egală sau mai mică specificitate.

Această strategie dă prioritate regulilor *cu condiții mai complexe*. Motivația este desigur aceea că o condiție mai complexă va fi satisfăcută într-o situație mai specifică. O măsură aproximativă a complexității este dată de numărul de comparații ce trebuie realizate de partea stângă a regulii:

- fiecare comparare cu o constantă sau o variabilă deja legată contează ca un punct;
- fiecare apel de funcție făcut în partea stângă ca parte a unui element `:`, `=` sau `test` adaugă un punct;
- funcțiile booleene `and` și `or` nu adaugă puncte dar argumentele lor adaugă;
- apelurile de funcții făcute în interiorul altor apeluri nu contează.

Strategia simplității (*simplicity strategy*): între activările de aceeași prioritate declarată, regulile nou activate sunt plasate în agendă în fața tuturor activărilor regulilor de egală sau mai mare specificitate.

Strategia aleatorie (*random strategy*): regulile activate din agendă sunt ordonate aleator.

9.4. Importanța ordinii asertărilor

Până acum nu am dat nici o importanță ordinii în care am scris faptele noi în comenzile (`assert...`). Surprinzător, ori poate nu, ordinea de apariție a faptelor în bază contează.

Să considerăm următorul exemplu:

```
(def facts baza
  (fact a)
)

(defrule r1
  ?fa<-(fact a)
=>
```

```

    (printout t "r1 " crlf)
    (retract ?fa)
    (assert (fapt b) (fapt c))
  )

  (defrule r2
    (fapt b)
    =>
    (printout t "r2 " crlf)
  )

  (defrule r3
    (fapt c)
    =>
    (printout t "r3 " crlf)
  )

```

Ordinea asertărilor în regula `r1` este întâi faptul `b` și apoi faptul `c`. Urmarea este că în agendă se va introduce mai întâi activarea regulii `r2` și apoi cea a regulii `r3`. Utilizarea strategiei în adâncime face ca prima regula aprinsă să fie cea corespunzătoare ultimei activări introduse în agendă, deci `r3`. Ordinea aprinderilor va fi deci `r1`, `r3`, `r2`.

```

CLIPS> (load "ordineAssert1.clp")
TRUE
CLIPS> (reset)
==> f-0      (initial-fact)
==> f-1      (fapt a)
==> Activation 0      r1: f-1
CLIPS> (run)
FIRE      1 r1: f-1
r1
<== f-1      (fapt a)
==> f-2      (fapt b)
==> Activation 0      r2: f-2
==> f-3      (fapt c)
==> Activation 0      r3: f-3
FIRE      2 r3: f-3
r3
FIRE      3 r2: f-2
r2
CLIPS>

```

Schimbarea ordinii asertărilor în `r1`:

```
(defrule r1
  ?fa<-(fapt a)
  =>
  (printout t "r2 " crlf)
  (retract ?fa)
  (assert (fapt c) (fapt b))
)
```

face ca introducerea în agendă a activărilor să fie acum în ordinea r3, r2, deci ordinea aprinderilor să fie r1, r2, r3:

```
CLIPS> (load "ordineAssert2.CLP")
TRUE
CLIPS> (reset)
==> f-0      (initial-fact)
==> f-1      (fapt a)
==> Activation 0      r1: f-1
CLIPS> (run)
FIRE      1 r1: f-1
r1
<== f-1      (fapt a)
==> f-2      (fapt c)
==> Activation 0      r3: f-2
==> f-3      (fapt b)
==> Activation 0      r2: f-3
FIRE      2 r2: f-3
r2
FIRE      3 r3: f-2
r3
CLIPS>
```

Este evident că o schimbare a ordinii activărilor poate duce la execuții complet diferite.

9.5. Eficientizarea execuției prin schimbarea ordinii comenzilor `retract` și `assert`

Să revenim la exemplul gradațiilor de merit prezentat în capitolul 8. Într-un context în care regula `sortare-initializare` nu era prevăzută să se activeze, nedumerea numărul repetat de activări ale ei, ca în trasarea:

```
...
FIRE      30 sortare-extrema-stinga: f-64,f-49
<== f-64      (lista-finala Ionescu 88.0)
```



```

<== Activation -100    sortare-extrema-dreapta: f-64,f-21
<== Activation -100    sortare-extrema-stinga: f-64,f-35
==> Activation -100    sortare-initializare: f-0,,f-21
==> Activation -100    sortare-initializare: f-0,,f-35
==> Activation -100    sortare-initializare: f-0,,f-49
<== f-49      (punctaj (nume Popescu) (valoare 110.5))
<== Activation -100    sortare-initializare: f-0,,f-49
==> f-65      (lista-finala Popescu 110.5 Ionescu 88.0)
==> Activation -100    sortare-extrema-dreapta: f-65,f-21
<== Activation -100    sortare-initializare: f-0,,f-35
<== Activation -100    sortare-initializare: f-0,,f-21
FIRE    31 sortare-extrema-dreapta: f-65,f-21
...

```

Cauza e ordinea comenzilor retract și assert în părțile drepte ale regulilor de sortare: retract urmat de assert. Rezultatul acestei ordini este că între momentul în care faptul `lista-finala` este retras și cel în care un nou fapt `lista-finala` este asertat, regula `sortare-initializare`, ce verifică în partea ei stângă lipsa unui fapt `lista-finala` din bază, își satisface condițiile de aplicabilitate și o activare a ei este inclusă în agendă.

Fără ca acest lucru să influențeze execuția, pentru că imediat ce noul fapt `lista-finala` apare, regula `sortare-initializare` se dezactivează, lucrul este de natură a afecta viteza de execuție a programului.

Corecția constă în inversarea ordinii acestor două comenzi, ca în:

```

(defrule sortare-extrema-dreapta
; plaseaza o persoana in coada listei
  (declare (salience -100))
  ?lst <- (lista-finala $?prime ?ultim-nume ?ultim-val)
  ?pct <- (punctaj (nume ?num) (valoare ?val&:(< ?val
?ultim-val)))
  =>
  (assert (lista-finala $?prime ?ultim-nume ?ultim-val
?num ?val))
  (retract ?lst ?pct)
)

```

Întâi se asertează un al doilea fapt `lista-finala` și abia apoi cel vechi este retras. Ca urmare, imediat după introducerea pentru prima oară a unui astfel de fapt, nu va mai exista nici un moment în care el să lipsească din bază, ceea ce va face ca regula `sortare-initializare` să nu se mai activeze la fiecare pas al sortării. Aceeași secvență dintre două aprinderi de mai sus, va raporta acum:

```
...
FIRE    30 sortare-extrema-stinga: f-64,f-49
==> f-65      (lista-finala Popescu 110.5 Ionescu 88.0)
==> Activation -100  sortare-extrema-dreapta: f-65,f-21
<== f-64      (lista-finala Ionescu 88.0)
<== Activation -100  sortare-extrema-dreapta: f-64,f-21
<== Activation -100  sortare-extrema-stinga: f-64,f-35
<== f-49      (punctaj (nume Popescu) (valoare 110.5))
FIRE    31 sortare-extrema-dreapta: f-65,f-21
...
```

Capitolul 10

Recursivitatea în limbajele bazate pe reguli

Paradigma de programare prin reguli nu rejectează *a priori* conceptul de recursivitate. Comportamentul recursiv poate să apară când limbajul permite unei reguli să “apeleze” alte reguli. Atunci, ori de câte ori o regulă invocă aceeași regulă, sau pe una care o invocă pe prima, avem un comportament recursiv. Limbajele care acceptă recursivitate utilizează, implicit, stiva. Un apel recursiv nu diferă, în esență, de un apel oarecare pentru că el este tradus, prin compilare, în aceleași tipuri de operații asupra stivei interne.

Limbajul CLIPS nu încorporează trăsătura de recursivitate. În primul rând, în CLIPS o regulă nu este “apelată” în accepțiunea clasică de apel din programare. Am putea spune că o regulă “se prinde singură” când a sosit momentul în care să-și ofere serviciile. În al doilea rând, limbajul nu permite ca, în secvența de operații ce se desfășoară după aprinderea unei reguli, adică în cursul executării părții ei drepte, alte reguli să poată fi chemate. Din acest motiv stiva nu este un accesoriu firesc al implementării limbajului CLIPS.

Ne punem, firesc, întrebarea: putem “derecurсивa”, prin reguli, algoritmi recursivi? Răspunsul este, desigur, pozitiv și esența soluției constă în realizarea explicită a stivei. Ca exemplificare, ne propunem să realizăm prin reguli doi dintre cei mai comuni algoritmi recursivi: problema turnurilor din Hanoi și factorialul unui număr natural.

10.1. Turnurile din Hanoi

Bine-cunoscuta problemă a turnurilor din Hanoi (*un număr de discuri găurite la mijloc, de diametre descrescătoare – în formularea originală – 64, formează o stivă pe un suport, să-l numim A; se cere să se transfere aceste discuri pe un suport B, folosind ca intermediar un suport C, mișcând un singur disc o dată și în așa fel încât la orice moment în configurațiile celor trei stive să nu apară un disc de diametru mai mare peste unul de diametru mai mic*) are următoarea soluție, într-o formulare liberă: ca să transfer n ($n > 1$) discuri de pe A pe B folosind C drept intermediar:

- transfer, în același mod, $n-1$ discuri de pe A pe C folosind B drept intermediar

- mut al n-lea disc de pe A pe B
 - transfer, în același mod, n-1 discuri de pe C pe B folosind A drept intermediar
 sau, într-o scriere pseudo-cod în care `one-move(a, b)` semnifică mutarea unui disc de pe a pe b (vezi și Figura 32):

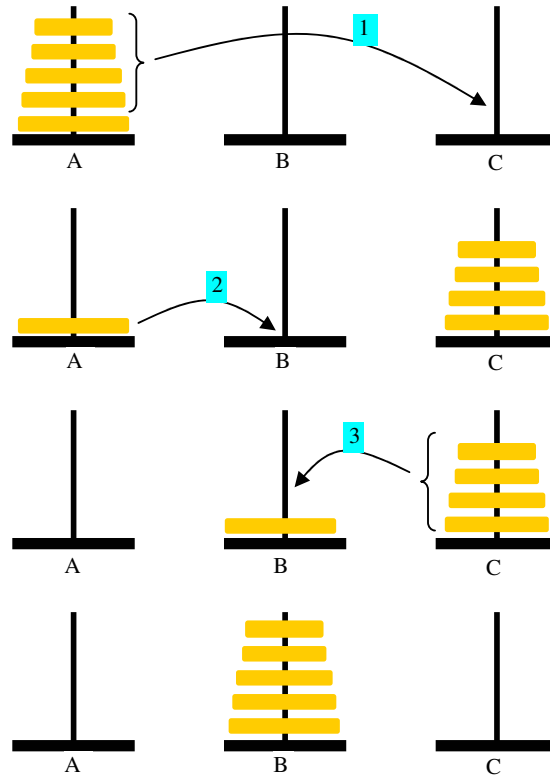


Figura 32: Turnurile din Hanoi

```

1. procedure hanoi(n, a, b, c)
   ; n - numarul de discuri
   ; a - suportul de plecare
   ; b - suportul de destinatie
   ; c - suportul ajutator
2. begin
3.   if n=1 then one-move(a, b) else

```

```

4.   begin
5.       hanoi(n-1, a, c, b)
6.       one-move(a, b)
7.       hanoi(n-1, c, b, a)
8.   end
9. end

```

Pentru transpunerea acestui algoritm în reguli, va trebui, așa cum am mai arătat, să introducem o structură de date care să aibă comportamentul unei stive. La început stiva va conține apelul inițial al procedurii recursive *hanoi*, apoi, la fiecare pas, apelul din vârful stivei va fi interpretat astfel: dacă e vorba de un apel recursiv – atunci el va fi expandat într-o secvență de trei apeluri: un prim apel recursiv, o mutare elementară și un al doilea apel recursiv, iar dacă este vorba de o mutare elementară – ea va fi executată.

Pentru simplificarea operațiilor ce urmează a se executa asupra stivei, preferăm să reprezentăm toate apelurile ca obiecte, argumentele unui apel fiind valori ale unor atribute ale acestor obiecte. Iată o posibilă reprezentare:

```

(deftemplate hanoi
  (slot index)
  (slot nr-disc)
  (slot orig)
  (slot dest)
  (slot rez)
)

(deftemplate muta
  (slot index)
  (slot orig)
  (slot dest)
)

```

Obiectul *hanoi* descrie un apel recursiv; obiectul *muta* descrie un apel-mutare elementară; *index* păstrează un index unic de identificare a apelului; *nr-disc* precizează numărul de discuri; *orig*, *dest* în ambele tipuri de obiecte reprezintă indecșii suporturilor de origine și respectiv destinație; *rez* dă indexul suportului de manevră.

În regulile de mai jos respectăm ordinea operațiilor asupra stivei, astfel încât numai obiectul aflat în vârful stivei să fie prelucrat la oricare intervenție asupra stivei. În funcție de tipul obiectului din vârf, trei operații pot avea loc:

- dacă obiectul este un apel recursiv cu un singur disc (*nr-disc*=1), el este transformat într-un apel de mutare:

```

(defrule transforma-recursiv-in-mutare
  ?stf <- (stiva ?item $?end)
  ?hf <- (hanoi (index ?item) (nr-disc 1) (orig ?fr)
(dest ?to)
          (rez ?re))
  =>
  (retract ?stf ?hf)
  (bind ?idx (gensym))
  (assert (muta (index ?idx) (orig ?fr) (dest ?to))
          (stiva ?idx $?end))
)

```

- dacă obiectul este un apel recursiv cu mai mult decât un singur disc ($\text{nr-disc} > 1$), el este expandat într-o secvență conținând un apel recursiv, un apel de mutare și un alt apel recursiv (vezi Figura 32):

```

(defrule expandeaza-hanoi
  ?stf <- (stiva ?item $?end)
  ?hf <- (hanoi (index ?item)
                (nr-disc ?no&:(> ?no 1))
                (orig ?fr) (dest ?to) (rez ?re))
  =>
  (retract ?stf ?hf)
  (bind ?idx1 (gensym))
  (bind ?idx2 (gensym))
  (bind ?idx3 (gensym))
  (assert
    (hanoi (index ?idx1) (nr-disc =(- ?no 1)) (orig
?fr)
          (dest ?re) (rez ?to))
    (muta (index ?idx2) (orig ?fr) (dest ?to))
    (hanoi (index ?idx3) (nr-disc =(- ?no 1)) (orig
?re)
          (dest ?to) (rez ?fr))
    (stiva ?idx1 ?idx2 ?idx3 $?end))
)

```

- și, în sfârșit, dacă obiectul este un apel de mutare, o acțiune corespunzătoare este efectuată, în cazul de față, tipărirea unui mesaj:

```

(defrule tipareste-mutare
  ?stf <- (stiva ?item $?end)
  ?hf <- (muta (index ?item) (orig ?fr) (dest ?to))
  =>
  (retract ?stf ?hf)
  (printout t "muta disc de pe " ?fr " pe " ?to crlf)
)

```

```

    (assert (stiva $?end))
  )

```

Desigur o regulă inițială trebuie să amorseze procesul prin introducerea primului apel recursiv în stivă. Regula următoare realizează acest lucru după interogarea utilizatorului asupra numărului de discuri:

```

(defrule hanoi-initial
  (not (stiva $?))
  =>
  (printout t "Turnurile din Hanoi. Numarul de discuri?"
  ")
  (bind ?n (read))
  (bind ?idx (gensym))
  (assert (hanoi (index ?idx) (nr-disc ?n)
                 (orig A) (dest B) (rez C))
          (stiva ?idx))
)

```

Să observăm că regulile *transforma-recursiei-in-mutare* și *tipareste-mutare* pot fi concentrate într-una singură, caz în care nu am mai avea nevoie de obiecte de tip *(muta ...)*.

O altă observație, mult mai importantă, se referă la ordinea în care sunt consumate apelurile recursive. Ea este nerelevantă, pentru că ceea ce interesează este doar secvența de mutări elementare cu care am rămâne în stivă după consumarea tuturor apelurilor recursive. Cu alte cuvinte, am putea înlocui stiva cu o **listă** de apeluri, din care să consumăm într-o primă etapă, în maniera arătată, toate apelurile recursive, pentru ca, într-o a doua etapă, să parcurgem în ordine lista pentru execuția mutărilor elementare.

Pentru a transforma ordinea de intervenție asupra obiectelor-apeluri din structura de memorare a apelurilor, dintr-una *ultimul-venit-primul-servit*⁶ (ceea ce face din structura de memorare o stivă) într-una aleatorie, doar următoarele modificări (marcate cu aldine mai jos) ar trebui operate asupra programului (toate regulile primesc în nume o terminație – **v1**):

```

(defrule transforma-recursiei-in-mutare-v1
  ?stf <- (stiva  $?beg ?item $?end)
  ?hf <- (hanoi (index ?item) (nr-disc 1) (orig ?fr)
              (dest ?to) (rez ?re))
  =>
  (retract ?stf ?hf)
  (bind ?idx (gensym))

```

⁶ Engl. *last-in-first-out*.

```

(assert (muta (index ?idx) (orig ?fr) (dest ?to))
        (stiva $?beg ?idx $?end))
)

(defrule expandeaza-hanoi-v1
; un obiect oarecare (cu n>1) din lista de memorare este
; selectat
  ?stf <- (stiva $?beg ?item $?end)
  ?hf <- (hanoi (index ?item) (nr-disc ?no&(> ?no 1))
              (orig ?fr) (dest ?to) (rez ?re))
=>
  (retract ?stf ?hf)
  (bind ?idx1 (gensym))
  (bind ?idx2 (gensym))
  (bind ?idx3 (gensym))
  (assert
    (hanoi (index ?idx1) (nr-disc =(- ?no 1))
            (orig ?fr) (dest ?re) (rez ?to))
    (muta (index ?idx2) (orig ?fr) (dest ?to))
    (hanoi (index ?idx3) (nr-disc =(- ?no 1))
            (orig ?re) (dest ?to) (rez ?fr))
    (stiva $?beg ?idx1 ?idx2 ?idx3 $?end))
)

```

Dacă intervențiile asupra structurii de memorare în vederea transformării apelurilor recursive în mutări elementare se pot face acum în orice ordine, regula de tipărire este singura care impune respectarea unei ordini în descărcarea acestei structuri, de la ultimul element introdus spre primul, ceea ce ne aduce din nou cu gândul la stivă. De asemenea, întrucât descărcarea trebuie să fie ultima fază a algoritmului, vom da regulii care o realizează o prioritate mai mică:

```

(defrule tipareste-mutare-v1
  (declare (salience -10))
  ?stf <- (stiva ?item $?end)
  ?hf <- (muta (index ?item) (orig ?fr) (dest ?to))
=>
  (retract ?stf ?hf)
  (printout t "muta disc de pe " ?fr " pe " ?to crlf)
  (assert (stiva $?end))
)

```


10.2. Calculul factorialului

Funcția recursivă de calcul al factorialului unui număr natural (*produsul numerelor naturale mai mici sau egale cu acesta*) este, într-o notație pseudo-cod, următoarea:

```

1. function fact(n)
2. begin
3.   if n = 1 then 1;
4.   else n * fact(n-1);
5. end

```

Într-un limbaj susținut de mecanismul de stivă, funcția de mai sus se transcrie aproape identic. Spre exemplu, în Lisp, ea arată astfel:

```

(defun fact(n)
  (if (eq n 1) 1 (* n (fact (- n 1)))))

```

Deși aparent mai simplă decât precedenta problemă a turnurilor din Hanoi, soluția calculului factorialului nu are o transcriere elementară într-un limbaj bazat pe reguli. Desigur, și aici, ca și în problema precedentă, stiva de apeluri joacă un rol important, ea trebuind să fie realizată explicit. Diferența față de problema precedentă este că, în cazul factorialului, în stivă apar operații ale căror argumente depind de rezultatele altor apeluri aflate la rândul lor în stivă. În problema turnurilor, apelurile erau de forma (hanoi 2 A B C) (muta A C) (hanoi 2 B C A) etc., adică argumentele erau cunoscute, ele nu depindeau de rezultatele altor apeluri aflate în stivă.

În Figura 33 este schițată evoluția stivei pentru un argument inițial al funcției factorial egal cu 3 (numele funcțiilor apar înaintea parantezelor deschise, între paranteze fiind notate argumentele; săgețile ce pleacă de pe pozițiile argumentelor secunde ale apelurilor funcției de înmulțire semnifică faptul că argumentele secunde trebuie considerate rezultatele evaluărilor apelurilor indicate de săgeți):

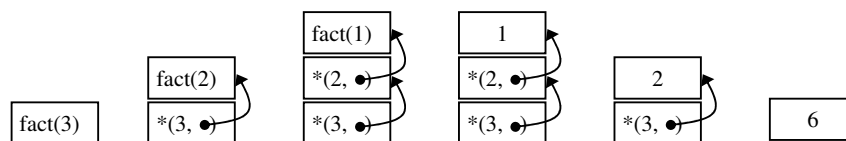


Figura 33: Evoluția stivei în calculul factorialului

După cum sugerează Figura 33, stiva are o dinamică foarte regulată, în sensul că într-o primă etapă ea crește, până la includerea în vârf a apelului factorialului de 1, pentru ca apoi ea să scadă, până ce în stivă rămâne un singur element ce conține rezultatul. Să mai observăm că în etapa descreșterii stivei, elementul din vârf nu conține un apel, ci un rezultat; de aceea vom prefera o reprezentare în care să separăm acest element de stivă, ca în Figura 34:

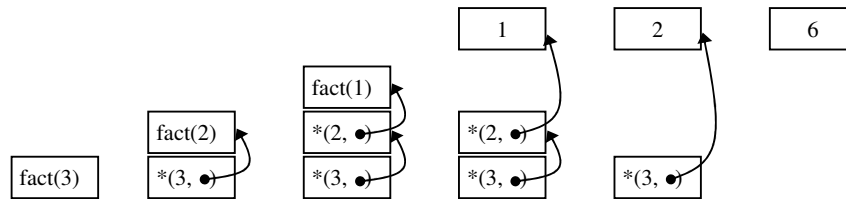


Figura 34: Reprezentare în care rezultatul este separat de stivă

Ca și mai sus, în problema turnurilor, vom reprezenta apelurile funcțiilor factorial și produs prin obiecte CLIPS:

```
(deftemplate factorial
  (slot index)
  (slot arg)
)

(deftemplate produs
  (slot index)
  (slot arg1)
  (slot arg2)
)
```

În aceste reprezentări atributele `index` au rostul de a identifica unic apeluri. Atributul `arg` al obiectelor `factorial` și atributul `arg2` al obiectelor `produs` indică alte obiecte, deci au ca tip valori `index`. Atributul `arg1` al obiectelor `produs` au ca valori numere naturale. Să remarcăm că în stivă obiectele `factorial` și `produs` rămân doar atât timp cât valorile lor nu se cunosc încă. Odată calculate, ele dispar, iar rezultatele apelurilor se vor păstra într-un obiect rezultat:

```
(deftemplate rezultat
  (slot index)
  (slot rez)
)
```

Rândul 3 din definiția pseudo-cod a funcției `factorial` conține condiția de terminare a recursiei. Ea se transcrie astfel:

```
(defrule interpreteaza-factorial-1
  ?stf <- (stiva ?item $?end)
  ?faf <- (factorial (index ?item) (arg 1))
  =>
  (retract ?stf ?faf)
  (assert (rezultat (index ?item) (rez 1))
          (stiva $?end))
)
```

Cu alte cuvinte, când elementul din vârful stivei este apelul unui `factorial` cu argument 1, rezultatul (numărul întreg 1) al apelului este memorat în atributul `rez` al unui obiect nou – `rezultat` – și stiva este decrementată. Aprinderea acestei reguli marchează începutul fazei de decrementare a stivei.

Rândul 4 al definiției conține apelul recursiv. El trebuie realizat prin două reguli, corespunzătoare fazei de incrementare și, respectiv, decrementare a stivei. Prima regulă va modifica stiva înlocuind apelul de `factorial` din vârful stivei cu un apel de produs și adăugând deasupra lui un nou apel de `factorial`. Primul argument al produsului este argumentul `factorialului` eliminat, iar argumentul al doilea al produsului este indexul apelului de `factorial` nou inclus:

```
(defrule interpreteaza-factorial-mai-mare-1
  ?stf <- (stiva ?item $?end)
  ?faf <- (factorial (index ?item) (arg ?n&(> ?n 1)))
  =>
  (retract ?stf ?faf)
  (bind ?idx1 (gensym))
  (assert (produs (index ?item) (arg1 ?n) (arg2 ?idx1))
          (factorial (index ?idx1) (arg =(- ?n 1)))
          (stiva ?idx1 ?item $?end))
)
```

Cea de a doua regulă, `interpreteaza-produs`, decrementează stiva după ce în stivă au rămas numai apeluri de produs. La fiecare aplicare a ei, un apel de produs este eliminat din stivă, după ce obiectul rezultat este actualizat cu produsul dintre valoarea pe care acest obiect o conținea și primul argument al apelului de produs aflat în vârful stivei:

```
(defrule interpreteaza-produs
  ?stf <- (stiva ?item $?end)
  ?prf <- (produs (index ?item) (arg1 ?a1) (arg2 ?idx2))
  ?ref <- (rezultat (index ?idx2) (rez ?r))
```

```

=>
  (retract ?stf ?prf)
  (modify ?ref (index ?item) (rez =(* ?al ?r)))
  (assert (stiva $?end))
)

```

Verificarea naturii numerice a primelor argumente ale apelurilor de produs este inutilă, dată fiind maniera în care aceste apeluri au fost construite.

Ultimele două reguli realizează amorsarea și, respectiv, stingerea procesului. Regula `start` interoghează utilizatorul asupra numărului argument al factorialului, iar regula `stop` afișează rezultatul:

```

(defrule start
  (not (stiva $?))
  =>
  (printout t "Factorial de ...? ")
  (bind ?n (read))
  (bind ?idx (gensym))
  (assert (fact ?n)
           (factorial (index ?idx) (arg ?n))
           (stiva ?idx))
)

(defrule stop
  (fact ?n)
  (stiva)
  (rezultat (rez ?r))
  =>
  (printout t "factorial de " ?n " este " ?r crlf)
)

```

De reflectat

În ultima lor variantă, regulile `transforma-recursiei-in-mutare-v1` și `expandeaza-hanoi-v1` se aplică într-o ordine care e dictată numai de strategia de rezoluție a conflictelor curent utilizată. Regula `tipareste-mutare-v1` va fi ulterior (datorită priorității mai mici) aplicată de un număr de ori egal cu numărul apelurilor elementare rămase în structura de memorare. Ce s-ar întâmpla însă dacă am înlocui regula `tipareste-mutare-v1` cu prima variantă a ei, `tipareste-mutare`, deci dacă ar avea aceeași prioritate cu a celorlalte reguli? Răspunsul este că, probabil, soluția ar fi mai bună. Dar dacă ea ar avea o prioritate mai mare? Răspunsul este că, sigur, soluția ar fi mai bună. Puteți găsi o explicație în favoarea acestor afirmații?

Argumentul `arg2` al obiectelor `produs` din problema factorialului pare superfluu, atât timp cât este folosit numai în faza dinamicii descrescătoare a stivei și indică întotdeauna același obiect `rezultat`. Modificați programul astfel încât obiectele `produs` să nu conțină atributul `arg2`.

Partea a IV-a

Dezvoltarea de aplicații

Operații pe liste, stive și cozi

Sisteme expert în condiții de timp real

Confruntări de șabloane în plan

O problemă de căutare în spațiul stărilor

Calculul circuitelor de curent alternativ

Rezolvarea problemelor de geometrie

Scurt ghid de programare spectaculoasă bazată pe reguli

Capitolul 11

Operații pe liste, stive și cozi

11.1. Inversarea unei liste

Ne propunem să *inversăm elementele unei liste*. La început baza de date va conține un singur fapt de forma:

```
(lista <element>+)
```

pentru ca, la sfârșit, ea să conțină tot un singur fapt, *lista*, dar ale cărei elemente să se afle în ordine inversă.

În soluția pe care o propunem⁷, schimbarea o efectuăm în aceeași structură, prin introducerea unui marcaj, față de care elementele sunt apoi copiate ca în oglindă. Mai întâi introducem în coada listei un marcaj (un simbol care nu mai există nicăieri între elementele listei), de exemplu un asterisc (*):

```
(deffacts lista
  (lista 1 2 3 4 5 6 7)
)

(defrule pregatire
  ?x <- (lista $?n)
  =>
  (retract ?x)
  (assert (lista $?n *))
)
```

Primul element al listei aflat în stânga marcajului (notat ?h în regula *muta-element*) este mutat pe prima poziție de după marcaj. Dacă se repetă această operație, elementele din stânga marcajului se vor muta în dreapta acestuia, în ordine inversă:

⁷ Soluție sugerată de Valentin Irimia.

```
(defrule muta-element
  (declare (salience 10))
  ?x <- (lista ?h&:(neq ?h *) $?t * $?r)
  =>
  (retract ?x)
  (assert (lista $?t * ?h $?r))
)
```

Trebuie să ne asigurăm că regula *pregatire* nu se mai aplică niciodată din momentul în care marcajul a fost introdus în listă. Din această cauză regula *muta-element* este declarată mai prioritară decât regula *pregatire*. Putem face acest lucru bazându-ne pe condiția regulii *pregatire*, care este mai laxă decât cea a regulii *muta-element*. În acest fel, în toate cazurile în care este filtrată regula *muta-element* va fi filtrată și regula *pregatire*. Deci *pregatire* se va aplica numai atunci când *muta-element* nu e filtrată.

În final, marcajul este îndepărtat. Motorul trebuie însă oprit printr-o comandă explicită de oprire datorită cerinței ca, la terminarea execuției, în bază să se afle numai un singur fapt *lista*, exact ca la început. Fără această comandă, motorul ar cicla la nesfârșit:

```
(defrule termina
  (declare (salience 10))
  ?x <- (lista * $?n)
  =>
  (retract ?x)
  (assert (lista $?n))
  (halt)
)
```

Să notăm încă o dată importanța declarației de prioritate. Dacă ea nu ar face din regula *termina* una la fel de prioritară ca și *muta-element*, în momentul în care *muta-element* și-ar înceta activitatea, regula *pregatire* ar putea fi din nou activată.

11.2. Mașini-stivă și mașini-coadă

Ne propunem să realizăm mai întâi o *mașină dedicată exploatarei stivei*. Ea ar putea fi utilizată ca o componentă a oricărei aplicații ce utilizează stiva. Operațiile pe stivă sunt:

- **push** <element>: <element> e introdus în stivă;
- **pop**: elementul din vârful stivei este eliminat într-o structură de memorare temporară **top**.

Vom impune ca structura de stivă să nu poată fi accesată de către proces decât prin intermediul acestor două operații. Deci, orice intervenție directă asupra stivei, văzută ca vector, de exemplu, este interzisă. Procesul care lucrează cu stiva știe că în urma unei operații **pop** va regăsi elementul eliminat din stivă în registrul **top**. Pentru că acesta are o singură poziție, o nouă operație **pop** va înlocui conținutul lui cu ultimul element aflat în vârful stivei.

Cum într-un limbaj bazat pe reguli nu există apeluri, execuția operațiilor **push** și **pop** trebuie comandată prin intermediul unor fapte. Vom adopta convenția ca procesul care utilizează stiva să comande o operație **push** sau una **pop** înscriind în bază fapte de forma: (push <element>) și, respectiv, (pop).

Registrul **top** va fi reprezentat printr-un fapt (top <element>). La început stiva e goală iar **top** acționează ca un registru ce poate memora o singură valoare:

```
(def facts masina-stiva
  (stack)
  (top nil)
)
```

Regula **pop-ok** realizează operația **pop** pentru cazul când în stivă există cel puțin un singur element. Elementul din vârful stivei este transferat în **top** iar stiva este decrementată:

```
(defrule pop-ok
  ?p <- (pop)
  ?s <- (stack ?top $?rest)
  ?t <- (top ?)
  =>
  (retract ?p ?s ?t)
  (assert (top ?top)
          (stack $?rest))
)
```

Următoarea regulă tratează cazul invocării operației **pop** pe o stivă goală. Situația este anunțată procesului prin setarea la **empty** a registrului **top**:

```
(defrule pop-empty
  ?p <- (pop)
  (stack)
  ?t <- (top ?)
  =>
  (retract ?p ?t)
  (assert (top empty))
)
```

Elementul comunicat de proces prin faptul `push` este plasat în vârful stivei:

```
(defrule push
  ?p <- (push ?elem)
  ?s <- (stack $?any)
  =>
  (retract ?p ?s)
  (assert (stack ?elem
                $?any))
)
```

În cele ce urmează descriem *funcționarea unei mașini care realizează o coadă de așteptare*. Faptul `queue` va ține coada iar faptul `temp` – elementul extras din coadă:

```
(deffacts masina-coada
  (queue)
  (temp nil)
)
```

Regula `out1` descrie operația **out**, de eliminare a unui element dintr-o coadă ce conține minimum un element:

```
(defrule out1
  ?o <- (out)
  ?q <- (queue ?first $?rest)
  ?t <- (temp ?)
  =>
  (retract ?o ?q ?t)
  (assert (temp ?first)
          (queue $?rest))
)
```

Comanda de eliminare a unui element dintr-o coadă goală provoacă includerea în `temp` a valorii `empty`:

```
(defrule out2
  ?o <- (out)
  (queue)
  ?t <- (temp ?)
  =>
  (retract ?o ?t)
  (assert (temp empty))
)
```

Comanda de includere a unui element în coadă se dă prin intermediul unui fapt `in` ce conține elementul de inclus:

```
(defrule in
  ?i <- (in ?elem)
  ?q <- (queue $?any)
  =>
  (retract ?i ?q)
  (assert (queue $?any ?elem))
)
```

11.3. Un proces care lucrează cu stiva

Să ne imaginăm că dorim să realizăm *inversarea elementelor listei utilizând o stivă*. Putem proceda astfel: se parcurg în secvență de la stânga la dreapta elementele listei și se încarcă, rând pe rând, fiecare element în stivă. După această secvență, în vârful stivei va sta ultimul element al listei. Într-o a doua etapă se descarcă stiva în listă, în ordinea de la stânga la dreapta. În acest fel ultimul element introdus, cel ce era la baza stivei, adică primul din lista inițială, va apărea acum pe ultima poziție a listei. Regulile de lucru pe stivă sunt cele din problema precedentă și, ca urmare, nu le mai reproducem aici.

```
(def facts procesul
  (lista 1 2 3 4 5 6 7)
)

(def facts control
  (faza incarca)
)

(defrule incarca-stiva
  (declare (salience -10))
  (faza incarca)
  ?l <- (lista ?first
              $?rest)
  =>
  (retract ?l)
  (assert (push ?first)
          (lista $?rest))
)
```

În faza `incarca stiva` se încarcă printr-un șir de operații **push**. Această regulă cheamă mașina de stivă prin plasarea în bază a unui fapt `push`. Dorim ca,

imediat ce un astfel de fapt apare, mașina să-l și execute. Execuția imediată ar trebui să aibă loc inclusiv în cazul în care un fapt `pop` s-ar introduce în bază. Ca urmare, va trebui să facem ca prioritățile regulilor mașinii stivă să fie mai mari decât cele ale regulilor procesului. Pentru a păstra intacte regulile stivei, vom coborî prioritățile regulilor procesului la -10.

Când lista s-a golit, procesul trece în faza de descarcare a stivei în listă:

```
(defrule comuta-faza
  (declare (salience -10))
  ?f <- (faza incarca)
  (lista)
  =>
  (retract ?f)
  (assert (faza descarca))
)
```

Următoarea regulă comandă mașinii de stivă să execute o operație **pop**. Ea este amorsată de existența unui registru `top` resetat la `nil`. O dată faptul `pop` depus în bază, datorită priorității mai mari a regulilor mașinii stivă, această operație este executată imediat:

```
(defrule descarca-stiva1
  (declare (salience -10))
  (faza descarca)
  (top nil)
  =>
  (assert (pop))
)
```

Rezultatul execuției unui **pop** este un element plasat în registrul `top`. Regula `descarca-stiva2` preia acest element și îl duce în coada listei. O precauție este luată ca elementul din `top` să fie diferit atât de `nil` cât și de `empty`:

```
(defrule descarca-stiva2
  (declare (salience -10))
  (faza descarca)
  ?l <- (lista $?any)
  ?t <- (top ?elem&:(and
    (neq ?elem nil)
    (neq ?elem
      empty)))
  =>
  (retract ?l ?t)
  (assert (lista $?any))
)
```

```

        ?elem)
      (top nil))
    )

```

Terminarea este forțată, în momentul în care `top` anunță stiva goală, prin retragerea faptului care precizează faza. Cum toate regulile procesului utilizează acest fapt, motorul se oprește:

```

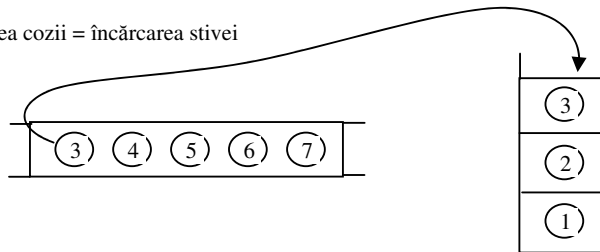
(defrule termina
  (declare (salience -10))
  ?f <- (faza descarca)
  (top empty)
  =>
  (retract ?f)
)

```

11.4. Un proces care lucrează simultan cu o stivă și o coadă

Următoarea problemă pe care o propunem complică lucrurile și mai mult, plecând de la observația că operațiile asupra listei, respectiv golirea ei, în prima fază, urmată de umplerea ei în ordine inversă, în faza a doua, sunt în fapt operații caracteristice cozii de așteptare. Vrem așadar să descriem *inversarea listei utilizând simultan mașinile stivă și coadă*. Figura 35 lămurește:

a. descărcarea cozii = încărcarea stivei



b. descărcarea stivei = încărcarea cozii

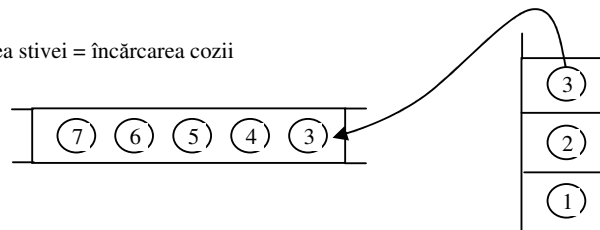


Figura 35: Lista (privită ca o coadă) inversată prin intermediul unei stive

Cum mașinile stivă și coadă sunt cele definite deja mai sus, dăm mai jos numai procesul care comandă operațiile asupra acestor două structuri. Elementele listei de inversat le considerăm inițial existente în coadă:

```
(deffacts masina-coada
  (queue 1 2 3 4 5 6 7)
  (temp nil)
)
```

Procesul se va desfășura în două faze: prima – de descărcare a cozii, respectiv încărcare a stivei – și cea de a doua – de încărcare a cozii, respectiv, descărcare a stivei.

```
(deffacts control
  (faza descarca-coada)
)

(defrule comanda-out
  (faza descarca-coada)
  (temp nil)
  =>
  (assert (out))
)
```

După ce mașina de coadă procesează o comandă **out**, în registrul `temp` apare un element diferit de `nil` și, atâta timp cât coada încă nu e vidă, diferit de `empty`:

```
(defrule comanda-push
  (declare (salience -10))
  (faza descarca-coada)
  ?t <- (temp ?elem&:(and (neq ?elem nil)
                           (neq ?elem empty)))
  =>
  (retract ?t)
  (assert (push ?elem)
          (temp nil))
)
```

O comandă **out** este urmată de o comandă **push**. Derularea acestor două operații, întotdeauna în această ordine, e dictată de conținutul registrului `temp` care trebuie să fie `nil` pentru execuția comenzii **out** și diferit de `nil` sau `empty` la execuția comenzii **push**. Pe de altă parte, după un **push** `temp` e făcut `nil`, iar după execuția unui **out** `temp` e umplut de mașina de coadă. Sincronizarea se realizează, așadar, automat.

La golirea cozii se trece în faza a doua – încarca-coada:

```
(defrule comuta-faza
  (declare (salience -10))
  ?f <- (faza descarca-coada)
  (temp empty)
  =>
  (retract ?f)
  (assert (faza incarca-coada))
)
```

În faza a doua, cele două comenzi în pereche sunt **pop** și **in**: **pop** amorsată de un registru **top** cu conținutul **nil**:

```
(defrule comanda-pop
  (declare (salience -10))
  (faza incarca-coada)
  (top nil)
  =>
  (assert (pop))
)
```

... iar **in** – de un registru **top** diferit de **nil** sau de **empty**. Rezultatul unui **pop** e umplerea lui **top**, care este vărsat în coadă prin operația **in**. În urma acestei operații **top** este resetat:

```
(defrule comanda-in
  (declare (salience -10))
  (faza incarca-coada)
  ?t <- (top ?elem&:(and (neq ?elem nil)
                        (neq ?elem empty)))
  =>
  (retract ?t)
  (assert (in ?elem)
          (top nil))
)
```

Regula următoare exprimă condiția de terminare, golirea lui **top**:

```
(defrule termina
  (declare (salience -10))
  ?f <- (faza incarca-coada)
  (top empty)
  =>
  (retract ?f))
```

Rezultatul se regăsește în configurația de fapte:

```
(queue 7 6 5 4 3 2 1)
(top empty)
(temp empty)
```

11.5. Evaluarea expresiilor

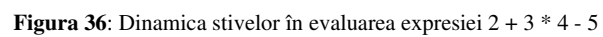
Ultima aplicație urmărește *evaluarea expresiilor aritmetice*. Pentru a păstra caracterul elementar al implementărilor, vom considera că expresiile nu conțin paranteze. Invităm cititorul să dezvolte aplicația pentru prelucrarea expresiilor cu paranteze.

Un binecunoscut algoritm de evaluare a expresiilor aritmetice utilizează două stive, una a operatorilor și cealaltă a operanzilor. E limpede că pot fi scrise reguli care să efectueze direct operațiile asupra stivelor, dar în cele ce urmează ne vom amuza să manipulăm două mașini-stivă simultan. Devine clar că ne trebuie o mașină de stivă generală (și nu avem motive să credem că lucrurile nu ar sta la fel și în cazul cozii) care să poată fi “clonată” în oricâte copii, astfel încât manipularea acestora să fie independentă și evoluția lor să poată fi sincronizată prin intermediul comenzilor push și pop, ce s-ar adresa fiecăreia în parte, și a registrelor ce păstrează elementele top. Este evident că stivele nu mai pot fi anonime, așa încât vom presupune că numele stivei este întotdeauna al doilea simbol al faptelor care le memorează, adică vom avea: (stiva opr <element>*) pentru operatori, respectiv (stiva opd <element>*) pentru operanzi. La fel, faptele ce comandă operațiile push vor fi etichetate (push opr <element>), respectiv (push opd <element>), operațiile pop: (pop opr) respectiv (pop opd) și elementele din vârf: (top opr <element>), respectiv (top opd <element>). În plus, pentru că algoritmul necesită la un moment dat o descărcare dublă a stivei operanzilor, vom permite o operație pop2 al cărei efect va fi extragerea a două elemente în registrul top corespunzător. Figura 36 este, în acest sens, ilustrativă.

Programul următor dă soluția. Există trei niveluri de priorități:

- al mașinii stivă (ridicată aici la 100),
- al regulilor curente ale domeniului problemei (considerate de prioritate 0)
- a două grupuri de reguli de condiții disjuncte (ambele de prioritate 10):

unul format din regula de terminare cu succes și cea de terminare pe eroare și altul al regulilor de calcul al operațiilor aritmetice:



```
;; Definitiile masinii-stiva

(defrule pop-ok
  (declare (salience 100))
  ?p <- (pop ?name)
  ?s <- (stiva ?name ?top $?rest)
```

```

    ?t <- (top ?name ?)
    =>
    (retract ?p ?s ?t)
    (assert (top ?name ?top)
             (stiva ?name $?rest))
  )

  (defrule pop2-ok
    (declare (salience 100))
    ?p <- (pop2 ?name)
    ?s <- (stiva ?name ?top1 ?top2 $?rest)
    ?t <- (top ?name ?)
    =>
    (retract ?p ?s ?t)
    (assert (top ?name ?top1 ?top2)
             (stiva ?name $?rest))
  )

  (defrule pop-empty
    (declare (salience 100))
    ?p <- (pop ?name)
    (stack ?name)
    ?t <- (top ?name ?)
    =>
    (retract ?p ?t)
    (assert (top ?name empty))
  )

  (defrule pop2-empty
    (declare (salience 100))
    ?p <- (pop2 ?name)
    (stack ?name ?)
    ?t <- (top ?name ?)
    =>
    (retract ?p ?t)
    (assert (top ?name empty))
  )

  (defrule push
    (declare (salience 100))
    ?p <- (push ?name ?elem)
    ?s <- (stiva ?name $?any)
    =>
    (retract ?p ?s)
    (assert (stiva ?name ?elem $?any))
  )

```

```

;;
;; Definitii din domeniul problemei
;;

(deffacts masina-stiva-opr
  (stiva opr)
  (top opr nil)
)

(deffacts masina-stiva-opd
  (stiva opd)
  (top opd nil)
)

(deffacts expresie
  (expr 2 + 3 * 4 - 5)
)

(deffacts prioritati
  (prio * 2)
  (prio / 2)
  (prio + 1)
  (prio - 1)
)

(defrule opd
; Primul element al expresiei este un operand: se comanda
; o operatie ; push asupra stivei opd cu acest operand.
  ?ex <- (expr ?x&:(numberp ?x) $?rest)
  =>
  (retract ?ex)
  (assert (expr $?rest)
    (push opd ?x))
)

(defrule opr1
; Primul element al expresiei este un operator: se
; comanda o operatie ; push asupra stivei opr.
  ?ex <- (expr ?x&:(not (numberp ?x)) $?rest)
  (stiva opr)
  =>
  (retract ?ex)
  (assert (expr $?rest)
    (push opr ?x))
)

```

```

(defrule opr2
; Primul element al expresiei este un operator de
; prioritate mai mare decit a celui din virful stivei
; opr: se comanda o operatie push in stiva opr.
  ?ex <- (expr ?x&:(not (numberp ?x)) $?rest)
  (stiva opr ?top $?oprs)
  (prio ?x ?px)
  (prio ?top ?ptop&:(> ?px ?ptop))
=>
  (retract ?ex)
  (assert (expr $?rest)
          (push opr ?x))
)

(defrule opr3
; Primul element al expresiei este un operator de
; prioritate mai mica sau egala cu cea a operatorului din
; virful stivel opr: se comanda descarcarea unui element
; al stivei opr si a doua elemente ale stivei opd.
  (expr ?x&:(not (numberp ?x)) $?rest)
  (stiva opr ?top $?oprs)
  (prio ?x ?px)
  (prio ?top ?ptop&:(<= ?px ?ptop))
=>
  (assert (pop opr)
          (pop2 opd))
)

(defrule opr4
; La terminarea expresiei se comanda descarcarea unui
; element al stivei opr si a doua elemente ale stivei
; opd. Expresia goala este stearsa si adaugata pentru a
; face regula aplicabila repetat.
  ?e <- (expr)
=>
  (retract ?e)
  (assert (expr)
          (pop opr)
          (pop2 opd))
)

(defrule terminare-rezultat
; Opre cind expresia si stiva opr sint goale iar stiva
; opd are un singur element
  (declare (salience 10))
  ?e <- (expr)
  (stiva opd ?x)

```

```

    (stiva opr)
    =>
    (retract ?e)
    (printout t "rezultat: " ?x crlf)
)

(defrule eroare
; Oprește pe eroare în faza de descarcare în care nu sunt
; minimum 2 elemente în stiva-opd
  (declare (salience 10))
  (top opd empty)
  =>
  (printout t "eroare!" crlf)
)

(defrule desc-mult
; Operatorul * disponibil în registrul top opr și
; doi operanți disponibili în registrul top opd:
; se realizează înmulțirea și se comandă push rezultatul
; în stiva opd. Prioritatea trebuie să fie mai mare decât
; a restului regulilor domeniului pentru a se executa
; întotdeauna când există registre top pline în această
; configurație.
  (declare (salience 10))
  ?sr <- (top opr *)
  ?sd <- (top opd ?opd1 ?opd2)
  =>
  (retract ?sr ?sd)
  (assert (top opr nil)
           (top opd nil)
           (push opd =(* ?opd2 ?opd1)))
)

(defrule desc-imp
; Analog pentru operatorul /
  (declare (salience 10))
  ?sr <- (top opr /)
  ?sd <- (top opd ?opd1 ?opd2)
  =>
  (retract ?sr ?sd)
  (assert (top opr nil)
           (top opd nil)
           (push opd =( / ?opd2 ?opd1)))
)

(defrule desc-adu
; Analog pentru operatorul +

```

```
(declare (salience 10))
?sr <- (top opr +)
?sd <- (top opd ?opd1 ?opd2)
=>
(retract ?sr ?sd)
(assert (top opr nil)
        (top opd nil)
        (push opd =(+ ?opd2 ?opd1)))
)

(defrule desc-sca
; Analog pentru operatorul -
  (declare (salience 10))
  ?sr <- (top opr -)
  ?sd <- (top opd ?opd1 ?opd2)
  =>
  (retract ?sr ?sd)
  (assert (top opr nil)
          (top opd nil)
          (push opd =(- ?opd2 ?opd1)))
)
```


Capitolul 12

Sisteme expert în condiții de timp real

Problemele în care evenimente externe trebuie tratate de un sistem expert sunt frecvente. Ceea ce interesează într-o astfel de realizare sunt cel puțin următoarele aspecte:

- apar evenimente externe la momente de timp oarecare;
- un eveniment extern trebuie procesat în general imediat, dacă el nu este la concurență cu alte resurse ale sistemului;
- timpul este un element important.

Pentru ca un sistem expert care conduce un proces ce funcționează în condiții de timp real să poată fi testat trebuie construit pentru el un **simulator**. Situația este explicată în Figura 37:

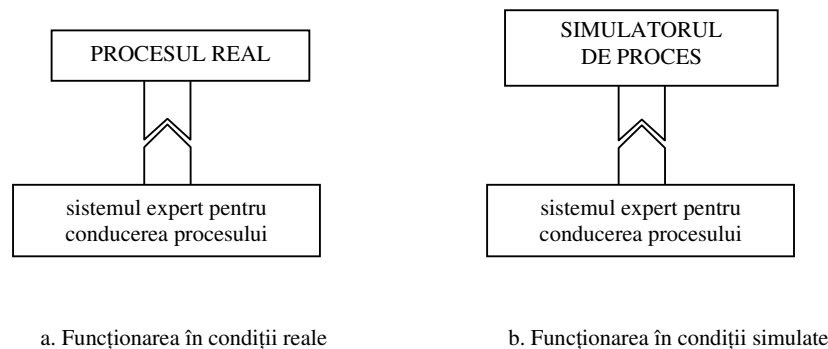


Figura 37: Un proces în timp real și simularea lui

Simulatorul de proces trebuie să reproducă cât mai fidel posibil evenimentele externe, în timp ce **sistemul expert pentru conducerea procesului** trebuie să facă sistemul să funcționeze.

În proiectarea aplicației este important să punem în capitole separate părțile care țin de conducerea procesului și cele care țin de controlul simulării. Dacă procedăm în acest mod, în momentul în care simularea a reușit, vom putea

îndepărta fără nici o problemă codul simulatorului, pentru a rămâne cu partea care realizează funcționarea sistemului, ce urmează să fie apoi cuplată la procesul real.

Există, de asemenea, cazuri în care doar simularea contează, sistemul expert este construit numai pentru a studia pe el comportamentul unui sistem real, ceea ce se urmărește fiind în acest caz cunoașterea mai bună a procesului în sine.

Pentru realizarea unui sistem expert care să lucreze în condiții de timp real, ne preocupă deci să găsim răspuns la următoarele întrebări:

- cum pot fi simulate evenimente externe?
 - ce caracteristici trebuie să aibă sistemul pentru a procesa evenimentele externe?
 - cum procedăm atunci când una din componentele sistemului este timpul?
- Cum simulăm ceasul?

Exemplul următor oferă câteva răspunsuri la aceste întrebări.

12.1. Servirea clienților la o coadă

Enunțul problemei este următorul: *se dorește simularea unei cozi de așteptare la un ghișeu. Coada e formată din clienți iar la ghișeu există un funcționar care servește clienții. Pentru a servi un client, funcționarul consumă o cantă de timp, întotdeauna aceeași. Apariția clienților în coadă se face la momente aleatorii de timp. Cel servit este primul din coadă, iar ultimul venit intră la urma cozii.*

Există două tipuri de evenimente externe în această problemă: intrarea clienților în coadă și tactul ceasului de timp real. Să numim evenimentele din prima categorie – evenimente **client**, și pe cele din a doua – evenimente **ceas**. Apariția evenimentelor client e aleatorie în timp, pe când apariția evenimentelor ceas se face întotdeauna după același interval de timp – cuanta de timp pe care o considerăm propice problemei noastre. Pentru simularea activității la o coadă secunda e o cantă de timp prea fină iar ora prea grosieră. Cea care convine e minutul.

Într-o primă variantă, vom simula apariția evenimentelor client de o manieră statică, adică memorând în fapte apariția acestor evenimente. Dimpotrivă vom lăsa pe seama unei reguli simularea evenimentelor ceas.

Grupul de fapte evenimente memorează faptele ce descriu evenimentele externe: ceasul – în formatul (ceas <minut>) și apariția clienților în coadă – în formatul (client <nume> <momentul apariției>). Ceasul este inițializat la momentul 0:

```
(def facts evenimente
  (ceas 0)
  (client Ion 0)
  (client Matei 1)
  (client Mircea 4))
```

```
(client Maria 11)
)
```

Grupul `parametri` memorează parametrii simulării – în cazul de față numai timpul de servire al unui client (presupus întotdeauna același):

```
(def facts parametri
  (time-servire 3)
)
```

Grupul `date` conține alte fapte ce ajută la simulare: coada – inițial goală, un indicator care ține starea funcționarului (`liber` ori `ocupat`) și un contor al timpului rămas pentru servirea unui client:

```
(def facts date
  (coada)
  (functionar liber)
  (trsc 0)
)
```

Regula de apariție în coadă a unui client se activează dacă ceasul ajunge la momentul în care trebuie luat în considerare un client pentru că acesta intră în coadă. Acțiunile efectuate sunt: retragerea faptului ce memorează apariția clientului – consumat – și introducerea numelui clientului în extremitatea dreaptă a cozii:

```
(defrule vine-client
  (ceas ?t)
  ?cl <- (client ?nume ?t)
  ?co <- (coada $?sir-clienti)
  =>
  (retract ?cl ?co)
  (assert (coada $?sir-clienti
    ?nume))
  (printout t "vine clientul " ?nume " la momentul "
    ?t crlf)
)
```

Regula ce marchează începerea servirii unui client este `inc-serv-client`: dacă funcționarul este liber și în coadă se găsește cel puțin un client, atunci funcționarul devine ocupat cu servirea primului client aflat la rând. Să notăm că dintre cele 5 condiții ale părții stângi a regulii, doar primele două sunt restrictive (ele testând respectiv funcționarul liber și existența cel puțin a unui nume în coadă). Faptul (`trsc ...`) este inițializat la valoarea cuantei de timp alocate clientului:

```
(defrule inc-serv-client
  ?func <- (functionar liber)
  (coada ?primul $?rest)
  (timp-servire ?ts)
  (ceas ?t)
  ?tr <- (trsc ?)
  =>
  (retract ?func ?tr)
  (assert (functionar ocupat
            ?primul)
          (trsc ?ts))
  (printout t "incepe servirea
    clientului " ?primul " la
    momentul " ?t crlf)
)
```

Regula de terminare a servirii unui client: dacă funcționarul este ocupat cu servirea clientului aflat în față la rând și timpul de servire al acestuia a expirat, atunci funcționarul devine liber și clientul iese din coadă:

```
(defrule termin-serv-client
  ?func <- (functionar ocupat ?nume)
  ?co <- (coada ?nume $?rest)
  ?tr <- (trsc 0)
  (ceas ?t)
  =>
  (retract ?func ?co)
  (assert (functionar liber)
          (coada $?rest))
  (printout t "termin servirea
    clientului " ?nume " la
    momentul " ?t crlf)
)
```

Următoarea regulă simulează ceasul: faptele care păstrează timpul și numărul de minute rămase pentru servirea clientului aflat în față sunt actualizate – primul incrementat, al doilea decrementat. Să observăm că prioritatea acestei reguli este cea mai mică dintre toate regulile simulării. Cum condițiile ei de aplicare sunt satisfăcute la orice pas al simulării (existența faptelor `ceas` și `trsc`), doar declarația de prioritate minimă face ca regula să se activeze numai în cazul în care nici o altă regulă nu mai poate fi aplicată.

```
(defrule tact-ceas
  (declare (salience -100))
```

```

?ce <- (ceas ?t)
?tr <- (trsc ?v)
=>
(retract ?ce ?tr)
(bind ?t (+ ?t 1))
(assert (ceas ?t)
        (trsc =(- ?v 1)))
(printout t "minutul: " ?t crlf)
)

```

Oprirea simulării se face când coada este vidă și nici un fapt client nu a mai rămas în bază. Maniera de oprire aleasă aici a fost prin retragerea unui fapt care este folosit în absolut toate regulile, cel conținând ceasul:

```

(defrule oprire
  (declare (salience 10))
  (coada)
  (not (client $?))
  ?ce <- (ceas ?)
  =>
  (retract ?ce)
)

```

Iată rezultatele rulării acestui program:

```

CLIPS> (load "Coada.clp")
TRUE
CLIPS> (reset)
CLIPS> (run)
vine clientul Ion la momentul 0
incepe servirea clientului Ion la momentul 0
minutul: 1
vine clientul Matei la momentul 1
minutul: 2
minutul: 3
termin servirea clientului Ion la momentul 3
incepe servirea clientului Matei la momentul 3
minutul: 4
vine clientul Mircea la momentul 4
minutul: 5
minutul: 6
termin servirea clientului Matei la momentul 6
incepe servirea clientului Mircea la momentul 6
minutul: 7
minutul: 8
minutul: 9

```

```

termin servirea clientului Mircea la momentul 9
minutul: 10
minutul: 11
vine clientul Maria la momentul 11
incepe servirea clientului Maria la momentul 11
minutul: 12
minutul: 13
minutul: 14
termin servirea clientului Maria la momentul 14
CLIPS>

```

12.2. Evenimente externe pseudo-aleatorii

Într-o a doua variantă a cozii simulate mai sus *vrem să generăm aleator evenimentele externe de tip **client**. Să presupunem că mai cerem ca simularea să se petreacă în timp real și dorim să putem controla numărul maxim de clienți ce pot să apară în unitatea de timp.*

Succesul satisfacerii acestor cerințe stă, evident, în abilitatea de a utiliza funcții care măsoară timpul și care generează numere pseudo-aleatorii. În CLIPS ele sunt (time) – care raportează timpul sistemului și (random) – care, la fiecare apel, generează un număr aleator în plaja $1 \div 2^{15}$.

Pentru exersarea utilizării generatorului de numere aleatoare, următorul program realizează o distribuție controlată de probabilitate: dacă în faptul prob se comunică un număr natural n, la întreruperea rulării faptul da va conține un număr de aproximativ n ori mai mare decât cel raportat de faptul nu.

```

(deffacts fapte
  (da 0)
  (nu 0)
  (prob 4)
)

(defrule test-prob
; Regula aserteaza de prob ori mai multe da-uri decit
; nu-uri.
  ?d <- (da ?x)
  ?n <- (nu ?y)
  ?v <- (prob ?p)
  =>
  (bind ?r (random))
  (if (> ?r (/ 32768 (+ ?p 1)))
    then (retract ?d) (assert (da =(+ ?y 1)))
    else (retract ?n) (assert (nu =(+ ?x 1)))
  )
)

```

Să pregătim simularea cozii în timp real realizând, mai întâi, un cod care generează simboluri noi la intervale aleatorii de timp, într-o anumită marjă de timp.

Marja de timp în care au loc aparițiile de simboluri este dată ca parametru într-un fapt `interval-max`, în secunde. Inițial, regula `aparitie` se aprinde pentru că momentul 0, ce inițializează un fapt `moment-urmator`, este mai mic decât timpul curent al sistemului. Ca urmare, momentul anunțat în faptul `moment-urmator` este actualizat la o valoare mai mare decât timpul curent cu o durată aleatorie calculată într-un interval cuprins între 1 și valoarea marjei de timp, printr-o funcție:

```
<moment-urmator> = (time) + <interval-max> * (random) /
215
```

De fiecare dată când acest moment în timp este atins, regula `aparitie` actualizează valoarea momentului următor. Măsurarea scurgerii acestui interval este datorată regulii `asteptare`. Condiția ei, fiind complementară celei a regulii `aparitie`, face inutilă o declarație de prioritate.

```
(deffacts de-timp
  (moment-urmator 0)
  (interval-max 60)
)

(defrule aparitie
; Intervalul s-a scurs: un simbol e generat si afisat
  ?a <- (moment-urmator ?ta&:(<= ?ta (time)))
  (interval-max ?im)
  =>
  (retract ?a)
  (printout t "apare " (gensym) crlf)
  (assert (moment-urmator
            (+ (time) (/ (* ?im (random)) 32766))))
)

(defrule asteptare
; Astept scurgerea intervalului.
  ?a <- (moment-urmator ?ta&:(> ?ta (time)))
  =>
  (retract ?a)
  (assert (moment-urmator ?ta))
)
```

Pregătirea simulării cozii de așteptare în timp real este acum terminată. Următorul program realizează această simulare.

Faptul (`timp-curent`) ține timpul curent al sistemului, deși el poate fi obținut oricând. Regulile `aparitie` și `asteptare` sunt cele construite mai sus. Fiind reguli care exploatează timpul prin condiții extrem de generoase (faptele `moment-urmator`, `interval-maxim` și `timp-curent` sunt permanent în bază), prioritatea lor este declarată mai mică decât a oricărei reguli a procesului. Numele unui client este generat de regula `aparitie` prin funcția (`gensym`), clientul în sine fiind anunțat prin includerea în bază a unui fapt `client`. Cele trei reguli ale domeniului rămân, așadar: regula care tratează includerea unui client în coadă – `vine-client`, regula care tratează începerea servirii unui client – `inc-serv-client` și regula care tratează terminarea servirii unui client – `termin-serv-client`:

```
(deffacts date-simulare
  (coada)
  (functionar liber)
  (mom-term-serv 0)
  (timp-servire 10)
)

(deffacts date-de-timp
  (moment-urmator 0)
  (interval-max 30)
  (timp-curent 0)
)

(defrule aparitie
; Intervalul s-a scurs: un simbol e generat si afisat
  (declare (salience -100))
  ?a <- (moment-urmator ?ta&:(<= ?ta (time)))
  (interval-max ?im)
  =>
  (retract ?a)
  (bind ?cl (gensym))
  (printout t "apare " ?cl crlf)
  (assert (moment-urmator
            =(+ (time) (/ (* ?im (random)) 32766)))
          (client ?cl))
)

(defrule asteptare
; Astept scurgerea intervalului.
  (declare (salience -100))
  ?a <- (moment-urmator ?ta&:(> ?ta (time)))
```



```

    ?atc <- (timp-curent ?tc)
    =>
    (retract ?a ?atc)
    (assert (moment-urmator ?ta)
             (timp-curent (time)))
  )

;;
;; Regulile domeniului
;;

(defrule vine-client
  ?cl <- (client ?nume)
  ?co <- (coada $?sirclienti)
  =>
  (retract ?cl ?co)
  (assert (coada $?sirclienti ?nume))
  (printout t "vine clientul " ?nume " la momentul "
             (time) crlf)
)

(defrule inc-serv-client
  (coada ?primul $?rest)
  ?func <- (functionar liber)
  (timp-servire ?ts)
  ?amts <- (mom-term-serv ?)
  =>
  (retract ?func ?amts)
  (bind ?t (time))
  (assert (functionar ocupat ?primul)
           ; aici ordinea are importanta
           (mom-term-serv =(+ ?t ?ts)))
  (printout t "incepe servirea clientului " ?primul
             " la momentul " ?t crlf)
)

(defrule termin-serv-client
  ?func <- (functionar ocupat ?nume)
  ?co <- (coada ?nume $?rest)
  (timp-curent ?tc)
  (mom-term-serv ?mts&:(<= ?mts ?tc))
  =>
  (retract ?func ?co)
  (assert (functionar liber) (coada $?rest))
  (printout t "termin servirea clientului " ?nume
             " la momentul " (time) crlf)
)

```


Capitolul 13

Confruntări de șabloane în plan

Ne propunem să investigăm în acest capitol probleme ce se caracterizează prin condiții asupra unor amplasamente de obiecte în plan. Atunci când acest lucru e posibil, este de dorit ca șabloanele regulilor însele să fie cât mai apropiate ca formă de structurile particulare căutate.

Limbajul CLIPS nu oferă opțiuni speciale pentru o confruntare de șabloane care să exploreze așezări ale obiectelor în spațiu. În [9], [10] se descrie un limbaj bazat pe reguli, numit **L-exp**, care încorporează trăsături speciale de definire a relațiilor de vecinătate între obiecte. Utilizând astfel de relații, se pot descrie apoi condiții complexe pentru găsirea lanțurilor de obiecte diferit configurate în spațiu. Regulile limbajului L-exp permit combinarea descrierilor de șabloane cu descrieri ale relațiilor de vecinătate. Alăturarea liniară în partea stângă a unei reguli L-exp, sub incidența unei relații de vecinătate, a mai multor șabloane, fiecare responsabil cu identificarea unor obiecte de un anumit tip, semnifică tentativa de depistare a lanțurilor de obiecte ce se supun, pe de o parte, restricțiilor elementare descrise de șabloane și, pe de altă parte, condițiilor de vecinătate descrise de relațiile invocate.

Prin faptele sale multi-câmp, CLIPS poate fi considerat ca oferind facilități rudimentare de definire spațială a obiectelor și de căutare corespunzătoare prin șabloane. Așezarea câmpurilor într-un fapt fiind una liniară, singura facilitate de descriere spațială oferită de CLIPS este una care exploatează ordinea liniară. De aceea configurațiile de obiecte care părăsesc linia pentru a se regăsi în plan trebuie descrise explicit prin șabloane succesive. Acest lucru poate fi făcut, de exemplu, prin tăierea planului în carioaje și gruparea obiectelor aflate pe aceste carioaje în faptele declarate. În Figura 38 se arată maniera (notorie) de reprezentare a unei figuri plane prin pătrate de dimensiunea unui carioaj. Un pătrat este considerat ca aparținând figurii dacă el conține semnificativ de multe puncte ale figurii.

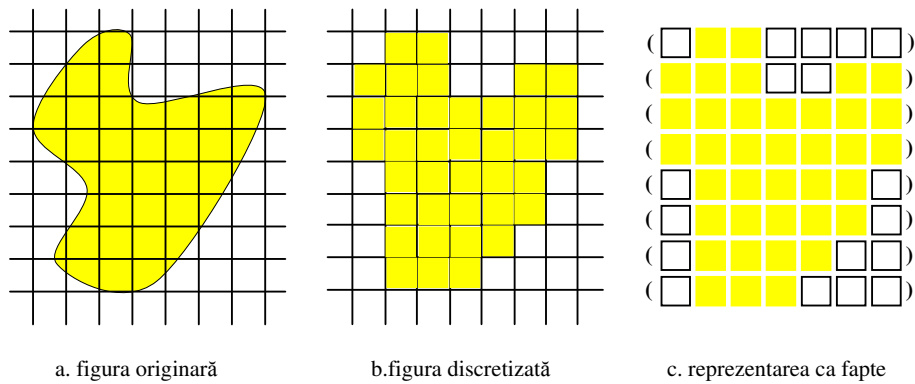


Figura 38: Reprezentări planare

Lipsa unei ordonări apriorice a faptelor în baza de date obligă la numerotarea liniilor pentru a permite specificarea explicită a coordonatei verticale, de exemplu ca în Figura 39.

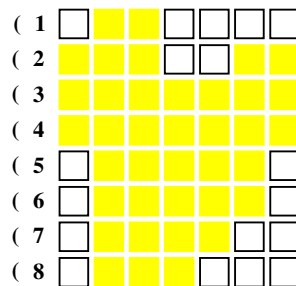


Figura 39: Specificarea explicită a ordonatei în fapte

Configurațiile liniare de obiecte sunt ușor de recunoscut prin reguli. De exemplu, o configurație ca aceasta:

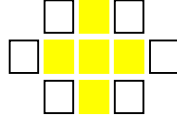


ce este plasată oriunde într-un rând, poate fi recunoscută cu un șablon:












(?lin \$?beg      \$?end)

Dacă trecem de la rând la plan, trebuie să facem ca formelor planare să le corespundă aranjări succesive de șabloane liniare. Alinierea orizontală a

șabloanelor, care urmează a se aplica asupra faptelor ce corespund rândurilor succesive, nu rezultă însă automat. Pentru a lămurii acest aspect, să considerăm situația în care am dori să recunoaștem aranjamente de obiecte de forma:














Această figură nu poate fi însă recunoscută cu o listă de șabloane de genul:

```
(?I1 $b1    $e1)
(?I2&:(= ?I2 (+ 1 ?I1)) $b2      $e2)
(?I3&:(= ?I3 (+ 2 ?I1)) $b3    $e3)
```

datorită libertății variabilelor șir \$b1, \$b2, \$b3 de a se lega la oricâte elemente și, prin aceasta, de a produce dezalinieră obiectelor de interes. Ca urmare, trebuie să controlăm pozițiile pe orizontală ale obiectelor ce trebuie recunoscute, în fiecare rând în parte. Există două modalități prin care putem face acest lucru:












- prin controlarea lungimii șirurilor de obiecte legate de variabilele șir ce colectează obiectele aflate în fața obiectelor de interes sau
- prin notarea poziției pe abscisă a obiectelor din figură.

Mai jos, continuăm exemplul de recunoaștere a cruciulițelor, considerând prima soluție:

```
(?I1 $b1    $e1)
(?I2&:(= ?I2 (+ ?I1 1)) $b2&:(= (length$ $b2) (- (length$ $b1) 1))      $e2)
(?I3&:(= ?I3 (+ ?I1 2)) $b3&:(= (length$ $b3) (length$ $b1))    $e3)
```

Prin aceasta, impunem restricția ca șirul de obiecte ce precede porțiunea din figură aparținând rândului al doilea să fie mai scurt cu o poziție decât cel ce precede secțiunea figurii de pe primul rând, iar acesta să fie egal în lungime celui de pe ultimul rând. Evident, o astfel de definiție a șabloanelor exploatează faptul că există o linie verticală care delimitează în stânga figura de căutat.

O scriere a șabloanelor care ar exploata simetria figurii de recunoscut plecând de la linia a doua înspre în sus și în jos ar trebui să plaseze întâi în regulă șablonul pentru linia a doua:

```
(?I2 $b2      $e2)
(?I1&:(= ?I1 (- ?I2 1)) $b1&:(= (length$ $b1) (+ (length$ $b2) 1))    $e1)
(?I3&:(= ?I3 (+ ?I2 1)) $b3&:(= (length$ $b3) (+ (length$ $b2) 1))    $e3)
```

O astfel de aranjare în pagină a șabloanelor este, însă, neintuitivă pentru că acestea nu mai reflectă fidel figura din plan.

În continuare, vom analiza două probleme inspirate din jocuri, care presupun recunoașteri de obiecte aflate în plan: jocul *8-puzzle* și jocul *vaporașe*.

13.1. Jocul 8-puzzle

În jocul *8-puzzle*, 8 piese pătrate care au desenate cifrele de la 1 la 8 trebuie mutate prin mișcări de glisare orizontală sau verticală dintr-o configurație inițială într-una finală, într-un spațiu de formă pătrată, 3 pe 3, din care lipsește o piesă (blancul).

În soluția pe care o propunem, obiectele au notate în fapte, în dreapta lor, pozițiile pe abscisă. Practic, un fapt care descrie o linie este de forma:

```
(linie <număr_linie> * <piesa_1> 1
                        * <piesa_2> 2
                        * <piesa_3> 3 *)
```

Marcajele `*` au rostul de a despărți coloanele formate din două elemente (un număr care indică piesa și un număr de coloană) pentru ca expresia șablon care identifică numărul coloanei să nu confunde o coloană cu o piesă. În această aranjare, numărul coloanei se află întotdeauna în a doua poziție după asterisc. Piesele sunt notate cu cifre între 1 și 8 iar blancul cu B. Lista celor trei fapte ce urmează descrie o configurație inițială de forma:

1	3	4
8	2	
7	6	5

```
(deffacts tabla
  (line 1 * 1 1 * 3 2 * 4 3 *)
  (line 2 * 8 1 * 2 2 * B 3 *)
  (line 3 * 7 1 * 6 2 * 5 3 *)
)
```

În continuare, regula `move-B-up` descrie o mutare a blancului în sus:

```
(defrule move-B-up
  ?l0 <- (line ?lin0&:(>= ?lin0 2)
           $?beg0 * B ?col * $?end0)
  ?l1 <- (line ?lin1&:(= ?lin1 (- ?lin0 1))
           $?beg1 * ?n ?col * $?end1)
  =>
  (retract ?l0 ?l1)
  (assert (line ?lin0 $?beg0 * ?n ?col * $?end0))
```

```

        (line ?lin1 $?beg1 * B ?col * $?end1))
    )

```

Primul șablon identifică linia cu blanc iar cel de al doilea șablon – linia aflată deasupra. În acest caz o aranjare a șabloanelor care să reflecte fidel aranjarea liniilor nu este recomandată, pentru că identificarea liniei cu blanc poate fi făcută imediat și, deci, economisim timp de calcul dacă plasăm primul șablonul care îi corespunde. Variabila `?col` identifică coloana blancului, pentru ca mutarea blancului în sus să se poată realiza în aceeași coloană.

Regula care descrie mutarea în jos a blancului este asemănătoare:

```

(defrule move-B-down
  ?l0 <- (line ?lin0&:(<= ?lin0 2)
           $?beg0 * B ?col * $?end0)
  ?l1 <- (line ?lin1&:(= ?lin1 (+ ?lin0 1))
           $?beg1 * ?n ?col * $?end1)
  =>
  (retract ?l0 ?l1)
  (assert (line ?lin0 $?beg0 * ?n ?col * $?end0)
          (line ?lin1 $?beg1 * B ?col * $?end1))
)

```

Pentru efectuarea mutărilor pe linie este suficient câte un singur șablon în fiecare regulă:

```

(defrule move-B-left
  ?l <- (line ?lin $?beg0
            * ?n ?col0
            * B ?col1&:(>= ?col1 2) * $?end0)
  =>
  (retract ?l)
  (assert (line ?lin $?beg0
            * B ?col0 * ?n ?col1 * $?end0))
)

(defrule move-B-right
  ?l <- (line ?lin $?beg0
            * B ?col0&:(<= ?col0 2)
            * ?n ?col1 * $?end0)
  =>
  (retract ?l)
  (assert (line ?lin $?beg0
            * ?n ?col0 * B ?col1 * $?end0)))

```

13.2. Jocul cu vapoare

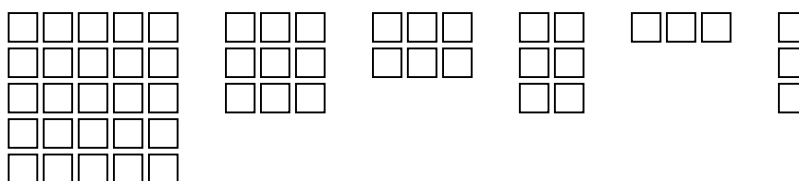
În jocul cu vapoare doi “combatanți maritimi” se confruntă pe un “câmp de luptă”, fiecare încercând să “scufunde” vasele celuilalt. Fiecare partener are în față două suprafețe de joc, una pe care sunt marcate vasele lui și cealaltă pe care marchează cunoașterea lui asupra zonei în care își ascunde adversarul vasele sale. Fiecare suprafață de joc este o tablă de 10x10 pătrățele, vapoarele pot avea mărimi variind între 2 și 4 pătrățele și forme date de orice configurație de pătrățele adiacente. Pentru scufundarea unui vaporăș de mărime n sunt necesare tot atâtea lovituri în exact pătrățelele pe care acesta le ocupă.

În general, într-o partidă de vapoare, un jucător întâi “bombardează” după un plan oarecare (care poate fi și aleator) caroiul adversarului, până ce o lovitură nimerește un vas al acestuia. Urmează un set de lovituri aplicate în vecinătatea celei norocoase, până la scufundarea țintei.

Sunt multe moduri prin care un program care joacă vapoare cu un partener uman poate fi făcut mai performant: maniera de scanare a terenului până la descoperirea poziției unei ținte, maniera de atac dintre prima lovitură aplicată unei ținte și scufundarea completă a acesteia, învățarea preferințelor adversarului în privința plasării vapoarelor și a formelor acestora în vederea măririi șanselor de câștig în partide viitoare (lovirea cu precădere în poziții care amintesc de forme ce par astfel a fi cele mai vulnerabile) etc.

În cele ce urmează ne vom preocupa de realizarea strategiilor pentru aruncarea bombelor în cele două situații de luptă: găsirea țintelor și scufundarea lor după o lovitură inițială. Fiecare regulă va descrie operațiile de efectuat când o anumită situație este întâlnită pe teren.

Într-un joc de vapoare, în fazele de localizare a țintelor preferăm să plasăm loviturile cât mai uniform în spațiul de joc, astfel încât densitatea loviturilor să crească relativ constant pe întreaga suprafață. Pentru recunoașterea suprafețelor goale putem aplica familii de șabloane planare cu care să descoperim oricare dintre configurațiile următoare de zone libere (număr linii * număr coloane): 5x5, 3x3, 2x3, 3x2, 1x3 și 3x1:



Primul șablon, de exemplu, pune în evidență o zonă liberă de 5x5 celule, care, pe o tablă inițială 10x10, poate fi instanțiată în 36 de moduri. Pentru

exemplificare, partea stângă a unei reguli care tentează descoperirea unei zone 2x3, poate fi scrisă astfel:

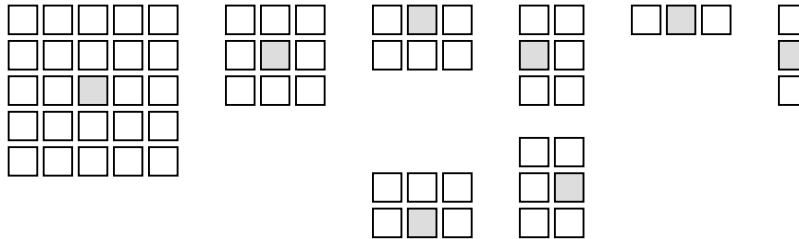
```
(defrule EU-la-joc-alege-zona-libera-2x3-1
  (declare (salience 8))
  ...
  (TU-T ?lin1 $?beg1 0 0 0 $?end1)
  (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1))
    $?beg2&:(= (length$ $?beg2) (length$ $?beg1))
      0 0 0 $?end2)
  =>
  ...
)
```

Într-o astfel de descriere zona de luptă a adversarului este reprezentată printr-o mulțime de 10 fapte liniare de forma:

```
(TU-T <nr-linie> <p1> <p2> <p3> <p4> <p5> <p6> <p7> <p8>
<p9> <p10>)
```

în care <p1> ... <p10> reprezintă cele 10 poziții ale liniei <nr-linie>, fiecare putând fi notată cu 0, dacă poziția nu este atinsă, sau cu numărul vaporului în cauză, atunci când se confirmă că lovitura a atins un vapor.

O dată descoperită o astfel de zonă, o regulă decide aplicarea unei lovituri într-o poziție care se află fie în mijlocul zonei goale, fie foarte aproape de mijloc (pătrățelele gri de mai jos):



Ca urmare, opt reguli, iar nu șase, descriu acțiunile ce trebuie efectuate la descoperirea zonelor goale. Pentru exemplificare, printre alte lucruri, regula care mai sus a fost prezentată pentru partea stângă ar putea conține în partea dreaptă și următoarele:

```
(defrule EU-la-joc-alege-zona-libera-3x2-1
  ...
  (TU-T ?lin1 $?beg1 0 0 0 $?end1)
  (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1))
```

```

    $?beg2&:(= (length$ $?beg2) (length$ $?beg1))
              0 0 0 $?end2)
=>
...
(printout t "Lovesc in patratul: (" ?lin2 ","
  (+ 1 (length$ $?beg2))
  ")!. Comunica rezultatul!" crlf)
)

```

Aşa cum ştim, prioritatea declarată a regulilor este importantă în stabilirea ordinii în care zonele goale vor fi lovite. Este uşor de văzut că numărul de instanţieri pe tabla de joc ale unui şablon este cu atât mai mare cu cât el încearcă descoperirea unor zone mai mici. Lăsarea ordinii de aplicare a regulilor în voia strategiei de rezoluţie a conflictelor cu care rulează sistemul poate rezulta într-o aglomerare a loviturilor în anumite zone ce sunt puse în evidenţă de şabloane care cercetează suprafeţe mici. Cum suprafaţa “acoperită” de un şablon reflectă în proporţie inversă densitatea de aplicare a loviturilor ($1/25$ – pentru şablon-ul 5×5 , $1/9$ pentru şablon-ul 3×3 ş.a.m.d. $1/3$ pentru cel 1×3 sau 3×1) şi cum ne preocupă să realizăm o acoperire a zonei de luptă, uniformă în densitatea loviturilor, care să varieze dinspre densităţi mici spre cele mari, ordinea de aplicare a regulilor trebuie să fie: mai întâi cele cu şabloane corespunzând suprafeţelor mari şi apoi cele cu şabloane corespunzând suprafeţelor mici. Cum nu avem nici un motiv pentru care să preferăm configuraţia 2×3 înaintea celei 3×2 , sau a configuraţiei 1×3 înaintea celei 3×1 , declaraţiile de prioritate vor trebui să repartizeze regulile pe şase niveluri.

```

(defrule EU-la-joc-alege-zona-libera-5x5
  (declare (salience 10))
  ...
  (TU-T ?lin1 $?beg1 0 0 0 0 0 $?end1)
  (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1)) $?beg2&:(= (length$
$?beg2) (length$ $?beg1)) 0 0 0 0 0 $?end2)
  (TU-T ?lin3&:(= ?lin3 (+ 2 ?lin1)) $?beg3&:(= (length$
$?beg3) (length$ $?beg1)) 0 0 0 0 0 $?end3)
  (TU-T ?lin4&:(= ?lin4 (+ 3 ?lin1)) $?beg4&:(= (length$
$?beg4) (length$ $?beg1)) 0 0 0 0 0 $?end4)
  (TU-T ?lin5&:(= ?lin5 (+ 4 ?lin1)) $?beg5&:(= (length$
$?beg5) (length$ $?beg1)) 0 0 0 0 0 $?end5)
  =>
  ...
  (printout t "Lovesc in patratul: (" ?lin3 "," (+ 3
(length$ $?beg3)) ")!. Comunica rezultatul!" crlf)
  )

(defrule EU-la-joc-alege-zona-libera-3x3

```

```

        (declare (salience 9))
        ...
        (TU-T ?lin1 $?beg1 0 0 0 $?end1)
        (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1)) $?beg2&:(= (length$
$?beg2) (length$ $?beg1)) 0 0 0 $?end2)
        (TU-T ?lin3&:(= ?lin3 (+ 2 ?lin1)) $?beg3&:(= (length$
$?beg3) (length$ $?beg1)) 0 0 0 $?end3)
        =>
        ...
        (printout t "Lovesc in patratul: (" ?lin2 "," (+ 2
(length$ $?beg2)) ")!. Comunica rezultatul!" crlf)
        )

    (defrule EU-la-joc-alege-zona-libera-2x3-1
      (declare (salience 8))
      ...
      (TU-T ?lin1 $?beg1 0 0 0 $?end1)
      (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1)) $?beg2&:(= (length$
$?beg2) (length$ $?beg1)) 0 0 0 $?end2)
      =>
      ...
      (printout t "Lovesc in patratul: (" ?lin2 "," (+ 2
(length$ $?beg2)) ")!. Comunica rezultatul!" crlf)
      )

    (defrule EU-la-joc-alege-zona-libera-2x3-2
      (declare (salience 8))
      ...
      (TU-T ?lin1 $?beg1 0 0 0 $?end1)
      (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1)) $?beg2&:(= (length$
$?beg2) (length$ $?beg1)) 0 0 0 $?end2)
      =>
      ...
      (printout t "Lovesc in patratul: (" ?lin1 "," (+ 2
(length$ $?beg1)) ")!. Comunica rezultatul!" crlf)
      )

    (defrule EU-la-joc-alege-zona-libera-3x2-1
      (declare (salience 8))
      ...
      (TU-T ?lin1 $?beg1 0 0 0 $?end1)
      (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1)) $?beg2&:(= (length$
$?beg2) (length$ $?beg1)) 0 0 0 $?end2)
      (TU-T ?lin3&:(= ?lin3 (+ 2 ?lin1)) $?beg3&:(= (length$
$?beg3) (length$ $?beg1)) 0 0 0 $?end3)
      =>
      ...

```

```

(printout t "Lovesc in patratul: (" ?lin2 "," (+ 1
(length$ $?beg2)) ")!. Comunica rezultatul!" crlf)
)

(defrule EU-la-joc-alege-zona-libera-3x2-2
  (declare (salience 8))
  ...
  (TU-T ?lin1 $?beg1 0 0 $?end1)
  (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1)) $?beg2&:(= (length$
$?beg2) (length$ $?beg1)) 0 0 $?end2)
  (TU-T ?lin3&:(= ?lin3 (+ 2 ?lin1)) $?beg3&:(= (length$
$?beg3) (length$ $?beg1)) 0 0 $?end3)
  =>
  ...
  (printout t "Lovesc in patratul: (" ?lin2 "," (+ 2
(length$ $?beg2)) ")!. Comunica rezultatul!" crlf)
)

(defrule EU-la-joc-alege-zona-libera-1x3
  (declare (salience 7))
  ...
  (TU-T ?lin $?beg 0 0 0 $?end)
  =>
  ...
  (printout t "Lovesc in patratul: (" ?lin "," (+ 2
(length$ $?beg)) ")!. Comunica rezultatul!" crlf)
)

(defrule EU-la-joc-alege-zona-libera-3x1
  (declare (salience 7))
  ...
  (TU-T ?lin1 $?beg1 0 $?end1)
  (TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1)) $?beg2&:(= (length$
$?beg2) (length$ $?beg1)) 0 $?end2)
  (TU-T ?lin3&:(= ?lin3 (+ 2 ?lin1)) $?beg3&:(= (length$
$?beg3) (length$ $?beg1)) 0 $?end3)
  =>
  ...
  (printout t "Lovesc in patratul: (" ?lin2 "," (+ 1
(length$ $?beg1)) ")!. Comunica rezultatul!" crlf)
)

```

Cel de al doilea motiv pentru care, în jocul de vaporeșe, am fi interesați de dispunerea elementelor în plan este legat de strategia de joc pe care o adoptăm pentru scufundarea unui vapor o dată realizată o primă lovitură. Vom implementa o

“strategie de luptă” prin care se lovesc cu prioritate pătrățelele aflate în imediata apropiere a unor lovituri deja aplicate unui vapor ce este încă nescufundat.

Astfel, presupunând că un fapt de forma (TU-V-lovite <nr-vapor> <nr-lovituri-primite>) memorează pentru fiecare vapor al adversarului ce a primit cel puțin o lovitură numărul de lovituri primite, iar (tip-V <nr-vapor> <gabarit>) reține pentru fiecare vapor, indiferent de jucător, din câte pătrățele este format, atunci decizia de lovire în poziția p a tablei adversarului este dată de conjuncția de condiții:

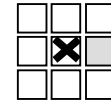
- vaporul n are lovite un număr de poziții mai mare decât zero dar mai mic decât numărul de poziții al tipului respectiv de vapor;
- poziția p se află în vecinătatea unei poziții lovite în vaporul n .

Prima dintre cele două condiții poate fi realizată de secvența următoarelor două șabloane:

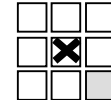
```
(TU-V-lovite ?nr-V ?lov)
(tip-V ?nr-V ?tip-V&:(< ?lov ?tip-V))
```

iar cea de a doua condiție, de diferite combinații de șabloane ce cercetează dispuneri planare în jurul unei poziții atinse în vaporul astfel depistat. În reprezentările de mai jos, cu **✕** s-a notat poziția veche și cu **■** cea nouă. În șabloane, se cercetează ca în poziția nouă să existe un 0.

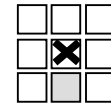
```
(TU-T ?lin $?beg ?nr-V 0 $?end)
```



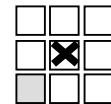
```
(TU-T ?lin1 $?beg1 ?nr-V $?end1)
(TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1))
 $?beg2&:(= (length$ $?beg2) (+ 1 (length$ $?beg1)))
 0 $?end)
```



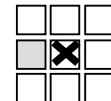
```
(TU-T ?lin1 $?beg1 ?nr-V $?end1)
(TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1))
 $?beg2&:(= (length$ $?beg2) (length$ $?beg1))
 0 $?end)
```



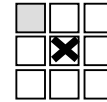
```
(TU-T ?lin1 $?beg1 ?nr-V $?end1)
(TU-T ?lin2&:(= ?lin2 (+ 1 ?lin1))
 $?beg2&:(= (length$ $?beg2) (- (length$ $?beg1) 1))
 0 $?end)
```



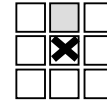
```
(TU-T ?lin $?beg 0 ?nr-V $?end)
```



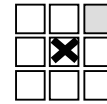
```
(TU-T ?lin1 $?beg1 ?nr-V $?end1)
(TU-T ?lin2&:(= ?lin2 (- ?lin1 1))
$?beg2&:(= (length$ $?beg2) (- (length$ $?beg1) 1))
0 $?end)
```



```
(TU-T ?lin1 $?beg1 ?nr-V $?end1)
(TU-T ?lin2&:(= ?lin2 (- ?lin1 1))
$?beg2&:(= (length$ $?beg2) (length$ $?beg1)) 0
$?end)
```



```
(TU-T ?lin1 $?beg1 ?nr-V $?end1)
(TU-T ?lin2&:(= ?lin2 (- ?lin1 1))
$?beg2&:(= (length$ $?beg2) (+ 1 (length$ $?beg1)))
0 $?end)
```



Capitolul 14

O problemă de căutare în spațiul stărilor

14.1. Maimuța și banana – o primă tentativă de rezolvare

O maimuță se află într-o cameră în care se află o banană, atârnată de tavan la o înălțime la care maimuța nu poate ajunge, și, într-un colț, o cutie. După un număr de încercări nereușite de a apuca banana, maimuța merge la cutie, o deplasează sub banană, se urcă pe cutie și apucă banana. Se cere să se formalizeze maniera de raționament a maimuței ca un sistem de reguli.

Problema este una clasică de inteligență artificială și, ca urmare, proiectarea unei soluții trebuie să urmărească descrierea stărilor, a tranzițiilor dintre stări (un set de operatori sau reguli) și o decizie în ceea ce privește controlul, adică maniera în care trebuie aplicate regulile.

În proiectarea stărilor putem pune în evidență trei tipuri de relații între entitățile participante la scenariul descris în problemă: dintre maimuță și cutie, dintre cutie și banană și dintre maimuță și banană, după cum urmează.

Relația maimuță – cutie:

MC-departe = maimuța se află departe de cutie

MC-lângă = maimuța se află lângă cutie

MC-pe = maimuța se afla pe cutie

MC-sub = maimuța de află sub cutie

Relația cutie – banană:

CB-lateral = cutia este așezată lateral față de banană

CB-sub = cutia este așezată sub banană

Relația maimuță – banană:

MB-departe = maimuța se află departe de banană

MB-ține = maimuța ține banana

Cu aceste predicate putem descrie orice stare a problemei, inclusiv stările inițială și finală (vezi și Figura 40):

Starea inițială: **MC-departe, CB-lateral, MB-departe.**

Starea finală: **MC-pe, CB-sub, MB-ține.**

Putem apoi inventaria următoarele tranziții între stări (setul de operatori, sau reguli):

aflată departe de cutie, maimuța se apropie de cutie: **apropie-MC**;

aflată lângă cutie, maimuța se depărtează de cutie: **depărtează-MC**;

aflată lângă cutie și lateral față de banană, maimuța trage cutia sub banană:
trage-sub-MCB;

aflată lângă cutie și sub banană, maimuța trage lateral cutia de sub banană:
trage-lateral-MCB;

aflată lângă cutie, maimuța se urcă pe ea: **urcă-MC**;

aflată pe cutie, maimuța coboară de pe ea: **coboară-MC**;

aflată lângă cutie, maimuța își urcă cutia deasupra capului:
urcă-pe-cap-MC;

din postura în care maimuța ține cutia deasupra capului, maimuța își dă jos cutia de pe cap: **coboară-de-pe-cap-MC**;

aflată pe cutie și sub banană maimuța apucă banana: **apucă-MB**;

Utilizând setul de predicate pe care l-am inventariat pentru descrierea stărilor, tranzițiile pot fi formulate ca reguli astfel:

```
apropie-MC:
dacă {MC-departe}
atunci șterge {MC-departe}, adaugă {MC-lângă}
```

```
depărtează-MC:
dacă {MC-lângă}
atunci șterge {MC-lângă}, adaugă {MC-departe}
```

```
trage-sub-MCB:
dacă {MC-lângă, CB-lateral}
atunci șterge {CB-lateral}, adaugă {CB-sub}
```

```
trage-lateral-MCB:
dacă {MC-lângă, CB-sub}
atunci șterge {CB-sub}, adaugă {CB-lateral}
```

```
urcă-MC:
dacă {MC-lângă}
atunci șterge {MC-lângă}, adaugă {MC-pe}
```

```
coboară-MC:
dacă {MC-pe}
atunci șterge {MC-pe}, adaugă {MC-lângă}
```



```

urcă-pe-cap-MC:
dacă {MC-lângă}
atunci șterge {MC-lângă}, adaugă {MC-sub}

coboară-de-pe-cap-MC:
dacă {MC-sub}
atunci șterge {MC-sub}, adaugă {MC-lângă}

apucă-MCB:
dacă {MC-pe, MB-departe, CB-sub}
atunci șterge {MB-departe}, adaugă {MB-ține}

```

O soluție a problemei ar fi, atunci, cea sugerată în Figura 40:

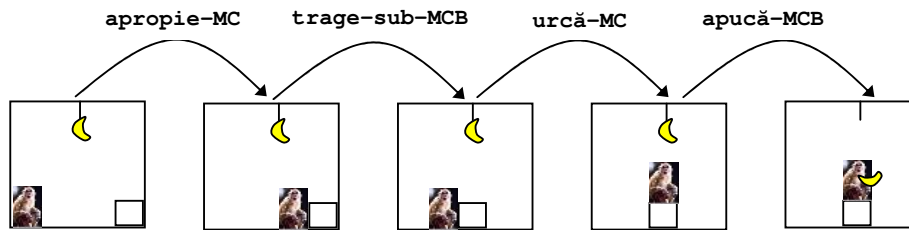


Figura 40: O secvență de stări și tranziții care rezolvă problema

Primul impuls este să “transcriem” aproape direct în CLIPS starea inițială și regulile, așa cum le-am inventariat mai sus, și să lăsăm motorul limbajului să meargă. Starea inițială se transpune în declarația următoare:

```

(deffacts initial-state
  (MC-departe)
  (CB-lateral)
  (MB-departe)
)

```

în timp ce regulile au o transcriere aproape *mot-à-mot*. Iată cum ar putea arăta una din ele:

```

(defrule apropiu-MC
; daca maimuta e departe de cutie atunci muimuta se
; apropiu de cutie
?mc <- (MC-departe)
=>
  (retract ?mc)
  (assert (MC-langa)))

```

Procedând în acest mod, ne vom da seama foarte repede că nu avem nici o șansă să rezolvăm problema, din două motive: lipsește o definiție a scopului și lipsește o strategie de apropiere de scop.

În cele ce urmează vom utiliza două strategii: una irevocabilă (*hill-climbing*) și una tentativă (cu revenire), prima fără succes, a doua cu succes.

14.2. Hill-climbing

Reamintim algoritmul *hill-climbing* (v., de exemplu, [33]):

```

procedure hill-climbing(initial-state)
{ current-state = initial-state;
  if (current-state e stare finală)
    return current-state;
  while (mai există operatori posibil de aplicat lui
current-state) {
    selectează un operator care nu a fost aplicat lui
current-state și aplică-l => new-state;
    if (new-state e stare finală) return new-state;
    if (new-state e mai bună decât current-state)
      current-state = new-state;
  }
  return fail;
}

```

Pentru aplicarea acestei strategii avem nevoie de o funcție de cost care să ne ajute să răspundem la întrebarea: “în ce condiții este o stare mai bună decât alta?”. În cazul de față, am putea să notăm cu diverse scoruri relațiile din familiile **MC**, **CB** și **MB** în așa fel încât starea finală să fie caracterizată de un scor maxim:

```

MC-departe = 0
MC-lângă = 1
MC-pe = 2
MC-sub = -1
CB-lateral = 0
CB-sub = 1
MB-departe = 0
MB-ține = 2

```

Cu aceste note, starea inițială e calificată: $0+0+0=0$, iar starea finală: $2+1+2=5$. Cum ne așteptăm însă, această funcție nu e univocă. Iată două stări care au același scor: starea {**MC-pe**, **CB-lateral**, **MB-departe**} are scorul:

$2 + 0 + 0 = 2$, iar starea {MC-lângă, CB-sub, MB-departe} are scorul: $1 + 1 + 0 = 2$. Ne așteptăm deci la situații de ezitări în evoluția spre soluție. De asemenea, nu e imposibil ca în acest parcurs să atingem stări intermediare care sunt caracterizate de maxime locale ale funcției de scor (orice stare în care s-ar putea tranzita are un scor mai mic). După cum se știe *hill-climbing* nu poate depăși stările de maxim local. Dar, să încercăm...

Algoritmul de mai sus sugerează o aplicare “virtuală” a unui operator, urmată de o evaluare și de consolidarea stării obținute în urma aplicării operatorului, sau, dimpotrivă, de invalidarea ei, în funcție de obținerea ori nu a unui scor mai bun. Pentru a realiza tentativa de aplicare a operatorului vom folosi, în afara unui fapt care păstrează starea curentă, un altul care ține starea virtuală, pasibilă de a lua locul stării curente. În plus, pare firesc să păstrăm scorul stării actuale chiar în reprezentarea stării. Faptele corespunzătoare stării curente și virtuale, ar putea fi deci:

```
(stare-curenta <scor> <predicat>*)
(stare-virtuala <scor> <predicat>*)
```

adică:

```
(def facts stari
  (stare-curenta 0 MC-departe CB-lateral MB-departe)
  (stare-finala 5 MC-langa CB-sub MB-tine)
)
```

În aceste convenții de reprezentare a stărilor să încercăm să descriem meta-regulile de control. Le numim meta-reguli pentru că, la fel ca în capitolul 5, ele manipulează fapte printre care se află și regulile de tranziție între stările problemei. Avem în vedere mai multe activități: aflarea unei stări virtuale plecând de la starea curentă, calculul scorului stării virtuale, modificarea eventuală a stării virtuale în stare actuală (dacă scorul stării virtuale îl surclasează pe cel al stării actuale).

Următoarele faze sunt semnificative și se repetă ciclic:

- selecția operatorilor,
- modificarea stării virtuale,
- calculul scorului stării virtuale,
- actualizarea stării curente din starea virtuală.

Putem reprezenta tranzițiile dintre stări ca obiecte regula caracterizate de atributele: nume, daca, sterge, adauga. Ca să evităm anumite complicații, dar și pentru că instanța problemei noastre ne permite, vom considera că atributele sterge și adauga țin, fiecare, exact câte un predicat:

```
(def template regula
  (slot nume)
```

```
(multislot daca)
(slot sterge)
(slot adauga)
)
```

```
(deffacts operatori
  (regula (nume apropiere-MC) (daca MC-departe) (sterge MC-
departe) (adauga MC-langa))
  (regula (nume departeaza-MC) (daca MC-langa) (sterge
MC-langa) (adauga MC-departe))
  (regula (nume trage-sub-MCB) (daca MC-langa CB-lateral)
(sterge CB-lateral) (adauga CB-sub))
  (regula (nume trage-lateral-MCB) (daca MC-langa CB-sub)
(sterge CB-sub) (adauga CB-lateral))
  (regula (nume urca-MC) (daca MC-langa) (sterge MC-
langa) (adauga MC-pe))
  (regula (nume coboara-MC) (daca MC-pe MB-departe)
(sterge MC-pe) (adauga MC-langa))
  (regula (nume urca-pe-cap-MC) (daca MC-langa) (sterge
MC-langa) (adauga MC-sub))
  (regula (nume lasa-de-pe-cap-MC) (daca MC-sub) (sterge
MC-sub) (adauga MC-langa))
  (regula (nume apuca-MB) (daca MC-pe MB-departe CB-sub)
(sterge MB-departe) (adauga MB-tine))
)
```

Putem ține scorurile individuale ale operatorilor într-o mulțime de fapte, astfel:

```
(deffacts scoruri
  (scor MC-departe 0)
  (scor MC-langa 1)
  (scor MC-pe 2)
  (scor MC-sub -1)
  (scor CB-lateral 0)
  (scor CB-sub 1)
  (scor MB-aproape 1)
  (scor MB-tine 2)
)
```

Într-un fapt faza vom memora faza în care se află automatul. Ne va fi comod ca în acest fapt să depozităm, ulterior, toți operatorii ce se pot aplica în starea curentă. Faza inițială este cea de selecție a operatorilor:

```
(deffacts initial-phase
  (faza selectie-operatori))
```

Urmează descrierea meta-regulilor procesului. Pentru a urmări derularea procesului vom prefera să afișăm starea curentă de fiecare dată când sistemul se află în faza *selectie-operatori*:

```
(defrule afiseaza
  (declare (salience 20))
  ?faz <- (faza selectie-operatori)
  (stare-curenta ?scor $?predsSC)
  =>
  (printout t "Stare: " ?scor $?predsSC crlf)
  (retract ?faz)
)
```

Dacă setul de predicate care descriu starea finală este o submulțime a setului de predicate din starea curentă, înseamnă că starea finală e satisfăcută și execuția se termină cu succes. Testarea terminării cu succes se face după afișarea stării curente și înainte de orice altă activitate a fazei:

```
(defrule succes
  (declare (salience 10))
  ?faz <- (faza selectie-operatori)
  (stare-curenta ?scor $?predsSC)
  (stare-finala ?scor $?predsSF&:(subsetp $?predsSF
    $?predsSC))
  =>
  (printout t "Succes!" crlf)
  (retract ?faz)
)
```

Regula *selectie-operatori* adună în coada faptului faza toți operatorii ce-și satisfac condițiile:

```
(defrule selectie-operatori
  ?faz <- (faza selectie-operatori $?selected-ops)
  (stare-curenta ?scor $?preds)
  (regula (nume ?r&:
    (not (member$ ?r $?selected-ops)))
    (daca $?conds))
  (test (subsetp $?conds $?preds))
  =>
  (retract ?faz)
  (assert (faza selectie-operatori $?selected-ops ?r))
)
```

Când nici un operator nu mai poate fi selectat, se tranzitează în faza de calcul a scorurilor stărilor posibil de atins din starea curentă:

```
(defrule tranzitie-in-calcul-scoruri
  (declare (salience -10))
  ?faz <- (faza selectie-operatori $?selected-ops)
  =>
  (retract ?faz)
  (assert (faza calcul-scoruri $?selected-ops)
          (scoruri-ops)))
)
```

Scorurile stărilor în care se poate face tranziția din starea curentă se calculează prin ștergerea din scorul stării curente a scorului predicatului de șters și adăugarea la rezultat a scorului predicatului de adăugat. Regula `calcul-scoruri-operatori` se aplică de atâtea ori câți operatori au fost selectați. Rezultatele, noile scoruri ale stărilor în care se poate face tranziție din starea curentă, sunt păstrate în faptul (`scoruri-ops...`), în pereche cu numele operatorilor care realizează tranzițiile:

```
(defrule calcul-scoruri-operatori
  ?faz <- (faza calcul-scoruri ?oper $?rest-ops)
  (regula (nume ?oper) (daca $?conds) (sterge ?del)
          (adauga ?add))
  (stare-curenta ?scor-sc ??)
  ?sc-ops <- (scoruri-ops $?all-scores)
  (scor ?del ?sc-del)
  (scor ?add ?sc-add)
  =>
  (retract ?faz ?sc-ops)
  (assert (faza calcul-scoruri $?rest-ops)
          (scoruri-ops $?all-scores
                      ?oper =(+ ?sc-add (- ?scor-sc ?sc-del))))
)
```

Când toate scorurile stărilor destinație au fost calculate, se trece în faza următoare – de ordonare a lor:

```
(defrule tranzitie-in-faza-ordonare
  (declare (salience -10))
  ?faz <- (faza calcul-scoruri)
  =>
  (retract ?faz)
  (assert (faza ordonare))
)
```

Pentru ordonarea scorurilor se caută în faptul `scoruri-ops` o secvență de două perechi `<operator, scor>` în care al doilea scor e mai mare decât primul și se comută între ele:

```
(defrule ordonare-scoruri-stari-noi
  (faza ordonare)
  ?sc-ops <- (scoruri-ops $?primii
                                     ?opr1 ?sc1&:(numberp ?sc1)
                                     ?opr2 ?sc2&:(> ?sc2 ?sc1)
                                     $?ultimii)

  =>
  (retract ?sc-ops)
  (assert (scoruri-ops $?primii ?opr2 ?sc2 ?opr1 ?sc1
                      $?ultimii))
)
```

Dacă nici o comutare nu mai poate fi operată, se trece în faza următoare – de tranziție efectivă a mașinii din starea curentă în starea de scor maxim:

```
(defrule tranzitie-in-faza-schimbarii-de-stare
  (declare (salience -10))
  ?faz <- (faza ordonare)
  =>
  (retract ?faz)
  (assert (faza schimbare-stare))
)
```

Regula `actualizeaza-stare` descrie operațiile ce se aplică stării curente prin tranzitarea cu operatorul cel mai eficient găsit, dacă acesta duce într-o stare de scor mai bun. Operatorul din fruntea listei `scoruri-ops`, care ține perechi `<operator, scor>` ordonate descrescător după scoruri, este aplicat stării curente. Pentru trasarea rulării se afișează operatorul aplicat. Din starea curentă se șterge un predicat și se include altul. Mașina tranzitează apoi din nou în faza selecție-operatori:

```
(defrule actualizeaza-stare
  ?faz <- (faza schimbare-stare)
  ?sc-ops <- (scoruri-ops ?opr ?scor-nou $?)
  (regula (nume ?opr) (sterge ?del) (adauga ?add))
  ?st-crt <- (stare-curenta
              ?scor-vechi&:(> ?scor-nou ?scor-vechi)
              $?prim-preds ?del $?rest-preds)

  =>
```

```
(printout t "Operator: " ?opr " >> ")
(retract ?faz ?sc-ops ?st-crt)
(assert (stare-curenta
        ?scor-nou $?prim-preds ?add $?rest-preds)
        (faza selectie-operatori))
)
```

Dacă în faza de schimbare de stare, regula anterioară nu se poate aplica, înseamnă că scorul stării curente, care nu este starea finală, este mai mare, sau cel puțin egal, cu al celei mai bune stări în care s-ar putea tranzita, și deci avem un eșec:

```
(defrule esec
  (declare (salience -10))
  ?faz <- (faza schimbare-stare)
  (stare-curenta ?scor $?predsSC)
  =>
  (printout t "Esec!! Stare: " ?scor " " $?predsSC crlf)
  (retract ?faz)
)
```

Rularea programului este următoarea:

```
CLIPS> (run)
Stare: 0 (MC-departe CB-lateral MB-departe)
Operator: apropiie-MC >> Stare: 1 (MC-langa CB-lateral MB-departe)
Operator: urca-MC >> Stare: 2 (MC-pe CB-lateral MB-departe)
Esec!! Stare: 2 (MC-pe CB-lateral MB-departe)
CLIPS>
```

Ea relevă oprirea în starea (stare-curenta 2 MC-pe CB-lateral MB-departe) care reprezintă într-adevăr un punct de maxim local pentru problemă. Blocarea în starea raportată, fără puțină de ieșire din ea, se produce datorită așezării întâmplătoare și “nefericite” a două elemente de scoruri maxime egale în lista scorurilor după terminarea ordonării acestora de către regula ordonare-scoruri-stari-noi:

```
(scoruri-ops urca-MC 2 trage-sub-MCB 2 departeaza-MC
0 urca-pe-cap-MC -1).
```


Pe primele două poziții s-au plasat aici operatorii *urca-MC* și *trage-sub-MCB*, ambii urmând a urca scorul stării curente la 2, însă în următoarele două configurații diferite:

- după o eventuală aplicare a operatorului *urca-MC*: (*stare-curenta* 2 MC-pe CB-lateral MB-departe);
- după o eventuală aplicare a operatorului *trage-sub-MCB*: (*stare-curenta* 2 MC-langa CB-sub MB-departe).

Singura mișcare posibilă în starea (*stare-curenta* 2 MC-pe CB-lateral MB-departe) fiind *coboara-MC*, care aduce scorul stării curente din nou la 1, această stare este una de maxim local.

14.3. O maimuță ezitantă: metoda tentativă exhaustivă

Pentru ieșirea din impas trebuie folosită o metodă cu revenire. Vom rula întâi problema maimuței și a bananei pe mașina de căutare exhaustivă cu revenire proiectată în capitolul 5.

Pentru aceasta nu avem decât să declarăm problema, starea inițială, cea finală și regulile de tranziție și să le integrăm motorului cu revenire proiectat atunci. Singura diferență între declarațiile utilizate mai sus și cele trebuincioase pentru rularea cu motorul tentativ constă în asignarea de priorități regulilor – o trăsătură pe care am impus-o instanțelor problemelor motorului tentativ. Dacă nu avem nici un motiv pentru care să atribuim priorități diferențiate regulilor problemei, le vom da la toate aceeași prioritate – zero.

Rularea dovedește un comportament extrem de ezitant, dar care, în cele din urmă, găsește soluția:

```
CLIPS> (run)
Stare: (MC-departe CB-lateral MB-departe)
Operator: apropie-MC >> Stare: (MC-langa CB-lateral MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral MB-departe)
backtracking in starea: (MC-langa CB-lateral MB-departe)
Operator: trage-sub-MCB >> Stare: (CB-sub MC-langa MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-sub MB-departe)
backtracking in starea: (CB-sub MC-langa MB-departe)
Operator: trage-lateral-MCB >> Stare: (CB-lateral MC-langa MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral MB-departe)
backtracking in starea: (CB-lateral MC-langa MB-departe)
```

```
Operator: urca-MC >> Stare: (MC-pe CB-lateral MB-departe)
Operator: coboara-MC >> Stare: (MC-langa CB-lateral MB-
departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral
MB-departe)
backtracking in starea: (MC-langa CB-lateral MB-departe)
Operator: urca-pe-cap-MC >> Stare: (MC-sub CB-lateral MB-
departe)
Operator: lasa-de-pe-cap-MC >> Stare: (MC-langa CB-
lateral MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral
MB-departe)
backtracking in starea: (CB-lateral MC-langa MB-departe)
Operator: urca-pe-cap-MC >> Stare: (MC-sub CB-lateral MB-
departe)
Operator: lasa-de-pe-cap-MC >> Stare: (MC-langa CB-
lateral MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral
MB-departe)
backtracking in starea: (MC-langa CB-lateral MB-departe)
Operator: urca-MC >> Stare: (MC-pe CB-lateral MB-departe)
Operator: coboara-MC >> Stare: (MC-langa CB-lateral MB-
departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral
MB-departe)
backtracking in starea: (CB-sub MC-langa MB-departe)
Operator: urca-MC >> Stare: (MC-pe CB-sub MB-departe)
Operator: coboara-MC >> Stare: (MC-langa CB-sub MB-
departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-sub MB-
departe)
backtracking in starea: (MC-langa CB-sub MB-departe)
Operator: trage-lateral-MCB >> Stare: (CB-lateral MC-
langa MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral
MB-departe)
backtracking in starea: (CB-lateral MC-langa MB-departe)
Operator: urca-pe-cap-MC >> Stare: (MC-sub CB-lateral MB-
departe)
Operator: lasa-de-pe-cap-MC >> Stare: (MC-langa CB-
lateral MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral
MB-departe)
backtracking in starea: (MC-langa CB-sub MB-departe)
Operator: urca-pe-cap-MC >> Stare: (MC-sub CB-sub MB-
departe)
```

```

Operator: lasa-de-pe-cap-MC >> Stare: (MC-langa CB-sub
MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-sub MB-
departe)
backtracking in starea: (MC-langa CB-sub MB-departe)
Operator: trage-lateral-MCB >> Stare: (CB-lateral MC-
langa MB-departe)
Operator: departeaza-MC >> Stare: (MC-departe CB-lateral
MB-departe)
backtracking in starea: (MC-pe CB-sub MB-departe)
Operator: apuca-MB >> Stare: (MB-tine MC-pe CB-sub)
Succes! Solutia: apropiere-MC trage-sub-MCB urca-MC apuca-
MB

```

14.4. O maimuță decisă: metoda *best-first*

Metoda *best-first* (v. de exemplu [33]) are avantajul că duce la soluție, în principiu, mai repede decât o metodă de căutare exhaustivă în spațiul stărilor. Motivul este căutarea ghidată de scor și precauția de a evita “agățarea” în maximele locale.

În continuare prezentăm câteva indicații pentru realizarea unei astfel de implementări. Se va utiliza o listă de stări candidate a fi vizitate – tradițional numită OPEN. Includerea unei stări în această listă va fi făcută dacă sunt satisfăcute două condiții: starea nu este deja acolo și starea nu a fost deja vizitată. Lista OPEN este continuu sortată, în așa fel încât cele mai reprezentative stări (adică cele caracterizate de scorurile cele mai mari) să poată fi vizitate primele. Chiar dacă toate stările care pot fi generate din starea actuală sunt de scoruri mai mici decât ale acesteia (deci când s-a atins un maxim – local ori absolut), ele sunt introduse totuși în listă. În acest fel se lasă posibilitatea de ieșire dintr-un punct de maxim local prin efectuarea unui salt direct într-o stare ce, cândva, a fost lăsată în suspans, pentru a încerca un alt urcuș.

Pentru verificarea condiției ca o stare deja vizitată să nu mai fie introdusă în lista OPEN este păstrată o înregistrare a stărilor deja vizitate. Vom numi această listă CLOSED. Așadar o stare este introdusă în lista CLOSED îndată ce a fost vizitată.

Să presupunem un moment în derularea algoritmului în care:

- mașina se află în starea curentă;
- există o listă OPEN a stărilor care urmează să fie vizitate, listă ordonată descrescător după scoruri;

- există o listă CLOSED a stărilor deja vizitate.

Secvența de operații în această situație pare a fi următoarea:

- dacă lista OPEN este vidă, atunci termină cu eșec, pentru că nu mai sunt stări de vizitat;

- dacă nu, alege ca stare curentă primul element al listei OPEN, elimină starea curentă din lista OPEN și include-o în starea CLOSED;
 - dacă starea curentă este starea finală, atunci termină cu succes;
 - dacă nu, generează toate stările ce pot fi atinse din starea curentă. Pentru aceasta:
 - selectează toți operatorii posibil de aplicat din starea curentă;
 - compune stările următoare de vizitat prin modificarea corespunzătoare a stării curente, calculând totodată scorurile acestor stări;
 - pentru fiecare dintre aceste stări generate, apoi:
 - verifică dacă starea nu a mai fost deja generată;
 - verifică dacă starea nu a fost deja traversată, prin comparare cu lista CLOSED;
 - verifică dacă starea nu este deja inclusă în lista OPEN;
 - include starea în lista OPEN în poziția corespunzătoare scorului.
 - starea curentă devine stare intermediară și reia ciclul.
- Rularea, mult mai scurtă, este următoarea:

```
CLIPS> (run)
Operator: >> Stare: 0 (MC-departe CB-lateral MB-departe)
Operator: >> Stare: 1 (MC-langa CB-lateral MB-departe)
Operator: >> Stare: 2 (MC-pe CB-lateral MB-departe)
Operator: >> Stare: 2 (MC-langa CB-sub MB-departe)
Operator: >> Stare: 3 (MC-pe CB-sub MB-departe)
Operator: >> Stare: 5 (MC-pe CB-sub MB-tine)
Succes!
CLIPS>
```

Capitolul 15

Calculul circuitelor de curent alternativ

Problema circuitelor de curent alternativ constă în calculul curenților care străbat diferitele ramuri ale unui circuit în care se află legate rezistoare, condensatoare și bobine. În schema electrică reală a unui rezistor apare întotdeauna o rezistență, în cea a unui condensator – o capacitate, iar în cea a unei bobine – o inductanță. În realitate însă, pe lângă aceste impedanțe dominante, mai pot coexista impedanțe parazite de valori mai mici. O bobină inclusă într-un circuit, de exemplu, contribuie și cu o mică rezistență, după cum, adesea, un condensator adaugă o mică inductanță. Calculul curenților se face prin aplicarea legilor lui Ohm, lucru asupra căruia nu vom insista în acest capitol. Ceea ce ne interesează este maniera în care putem utiliza un limbaj de programare bazat pe reguli pentru calculul impedanțelor circuitelor serie, paralel sau ale combinațiilor serie-paralel. După cum vom vedea însă, învățămintele trase în urma acestei experiențe vor fi chiar mai bogate decât strict găsirea unei maniere de rezolvare a unei probleme de fizică de liceu.

În notarea elementelor de circuit în curent alternativ este curentă utilizarea numerelor complexe, o impedanță fiind caracterizată de o componentă reală și una imaginară:

$$z = a + ib$$

Impedanțele se calculează după binecunoscutele formule:

- serie:

$$z_s = \sum_{i=1}^n z_i$$

- sau paralel:

$$1/z_p = \sum_{i=1}^n \frac{1}{z_i}$$

iar, din punct de vedere al impedanței, rezistoarele (notate R), capacitățile (notate C) și inductanțele (notate L) au formulele:

$$z_R = R$$

$$z_C = iL\omega$$

$$z_C = i\omega/C$$

unde ω reprezintă pulsația curentului alternativ.

Pentru modelarea problemei noastre ca una de calcul bazat pe reguli, să observăm că putem descompune calculul impedanței echivalente a oricărui circuit RLC serie-paralel într-o secvență de operații serie ori paralel, în care să intervină numai două componente de circuit. Astfel, de exemplu, impedanța echivalentă a unui circuit ca acesta:

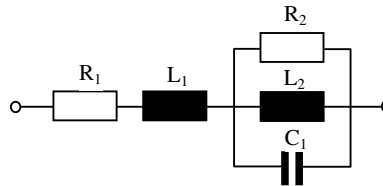


Figura 41: Un exemplu de grupare serie-paralel

poate fi calculată ca o secvență de operații de genul:

$$z_{S1} = z_{R1} \text{ serie } z_{L1}$$

$$z_{P1} = z_{R2} \text{ paralel } z_{L2}$$

$$z_{P2} = z_{P1} \text{ paralel } z_{C1}$$

$$z_{S2} = z_{S1} \text{ serie } z_{P2}$$

Datorită asociativității operației de adunare, ordinea în care sunt grupate impedanțele într-o dispunere serie sau într-una paralel este nerelevantă. Organizarea calculului schițată mai sus sugerează construirea a două reguli, una capabilă să calculeze impedanța echivalentă a un circuit serie format din două elemente de circuit, cealaltă – a unui paralel de aceeași natură. Apoi, nedeterminismul inerent al aplicării regulilor va constitui matca naturală de combinare a impedanțelor într-o ordine aleatorie. Pentru că nu avem de ce să controlăm acest proces, paradigma de programare bazată pe reguli pare să fie cea mai convenabilă.

Putem avea ca intrări direct valorile rezistorilor, capacităților și inductanțelor, caz în care trebuie organizat un calcul care să le aducă la formatul de numere complexe. Spre exemplu, putem memora aceste valori printr-o serie de fapte de

forma (rezistor <nume> <valoare>), (capacitate <nume> <valoare>), (inductanta <nume> <valoare>), unde, în fiecare caz în parte, valorile sunt date în unitățile de măsură corespunzătoare, respectiv ohmi, farazi, henri și un singur fapt (pulsatie <valoare>) care precizează pulsația curentului, în Hz. Dacă reprezentăm numerele complexe prin fapte de forma (complex <nume> <parte-reala> <parte-imaginara>), atunci transformarea elementelor electrice de circuit în impedanțe poate fi realizată printr-un set de trei reguli de forma:

```
(defrule calcul-rezistor
  ?r <- (rezistor ?num ?val)
  =>
  (retract ?r)
  (assert (complex ?num ?val 0))
)

(defrule calcul-capacitate
  ?c <- (capacitate ?num ?val)
  (pulsatie ?p)
  =>
  (retract ?c)
  (assert (complex ?num 0 =(/ ?p ?val)))
)

(defrule calcul-inductanta
  ?l <- (inductanta ?num ?val)
  (pulsatie ?p)
  =>
  (retract ?l)
  (assert (complex ?num 0 =(* ?p ?val)))
)
```

În continuare, definirea circuitului o facem printr-o serie de fapte de forma (serie <nume> <nume-1> <nume-2>) și (paralel <nume> <nume-1> <nume-2>), unde prin <nume> notăm combinația serie ori paralel realizată, iar prin <nume-1> și <nume-2> – circuitele parțiale ce intervin în grupare. Astfel, pentru grupul rezistiv-capacitiv-inductiv din Figura 41, intervine următoarea descriere:

```
(def facts circuit
  (serie S1 R1 L1)
  (paralel P1 R2 L2)
  (paralel P2 P1 C1)
  (serie F P2 S1))
```

```
(deffacts comenzi
  (calcul F)
)
```

Din păcate această descriere face mai mult decât am dori: ea precizează într-o anumită măsură și ordinea de efectuare a calculelor. Într-adevăr, dacă notăm cu S1 gruparea serie realizate de rezistorul R1 și inductanța L1, cu P1 – gruparea paralelă dintre rezistorul R2 și inductanța L2, cu P2 – gruparea paralelă realizată de gruparea anterioară și capacitatea C1 și, în final, cu F – gruparea serie dintre P2 și S1, atunci nu mai rămâne prea mult nedeterminism pentru că, de exemplu, nu pot să realizez întâi gruparea paralelă dintre C1 și L2, pentru ca rezultatul acesteia să-l combin apoi cu R2 ș.a.m.d. Să numim acest necaz “ordine serie-paralel precizată” și să-l uităm deocamdată, promițând a reveni asupra lui. Pentru moment ne vom concentra asupra modului în care putem controla propagarea calculelor în acest circuit, în care, așa cum am văzut, maniera de descriere impune o ordonare parțială a efectuării calculelor, ordonare sugerată de laticia din Figura 42:

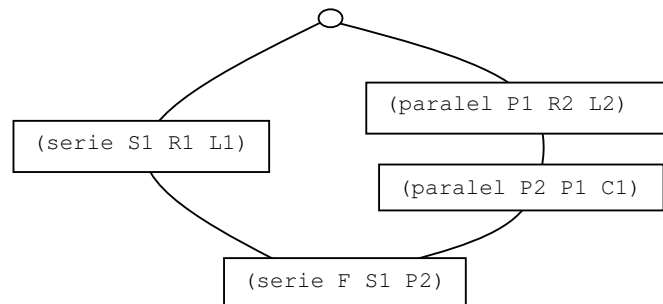


Figura 42: Ordinea operațiilor serie-paralel văzută ca o latică

Poza de mai sus exprimă următoarele: “ca să pot calcula serie F, trebuie să am simultan rezultatele de la serie S1 și paralel P2, iar ca să pot calcula paralel P2, trebuie să cunosc paralel P1. Care dintre serie S1 și paralel P1 se calculează întâi – nu este important, atâta timp cât valorile R1, R2, L1, L2 și C1 pot fi accesate în orice moment”.

Ideea este de a propaga și multiplica un set de “fanioane”, dinspre expresiile finale către cele inițiale, care să anunțe valorile necesare efectuării calculelor. Dacă un fanion ajunge pe o expresie care poate fi calculată, aceasta este calculată prioritar, pe când dacă un fanion ajunge pe o expresie care, pentru a fi calculată, depinde de alte expresii, fanionul se multiplică în tot atâtea alte fanioane câte expresii necalculate referă expresia în cauză.

Astfel, o regulă care “calculează” o expresie “calculabilă” serie poate fi următoarea:

```
(defrule serie-CC
  (declare (salience 10))
  ?c <- (calcul ?nume-s)
  (serie ?nume-s ?nume-1 ?nume-2)
  (complex ?nume-1 ?a1 ?b1)
  (complex ?nume-2 ?a2 ?b2)
  =>
  (retract ?c)
  (assert (complex ?nume-s =(+ ?a1 ?a2) =(+ ?b1 ?b2)))
)
```

Expresia de forma (serie <nume-s> <nume-1> <nume-2>) este recunoscută ca fiind calculabilă prin existența în bază a faptelor (complex <nume-1> ...) și (complex <nume-2> ...) ce dau expresiile ca numere complexe ale elementelor de circuit <nume-1> și <nume-2>.

Analoga ei pentru expresii paralel va arăta mult mai complicat, datorită expresiilor inversate ce apar în echivalarea paralelă:

```
(defrule paralel-CC
  (declare (salience 10))
  ?c <- (calcul ?nume-s)
  (paralel ?nume-s ?nume-1 ?nume-2)
  (complex ?nume-1 ?a1 ?b1)
  (complex ?nume-2 ?a2 ?b2)
  =>
  (retract ?c)
  (assert (complex ?nume-s =(...) =(...)))
)
```

Dimpotrivă, următoarele trei reguli “propagă fanioane” din expresii “necalculabile” încă, adică expresii ce conțin sub-expresii încă necalculate:

```
(defrule serie-NC-C
; Un circuit serie in care prima expresie e necalculata
; si a doua calculata propaga fanionul pentru prima
; expresie
  (calcul ?nume-s)
  (serie ?nume-s ?nume-1 ?nume-2)
  (complex ?nume-2 ? ?)
  =>
  (assert (calcul ?nume-1))
)
```

```

(defrule serie-C-NC
; Un circuit serie in care prima expresie e calculata si
; a doua necalculata propaga fanionul pentru a doua
; expresie
(calcul ?nume-s)
(serie ?nume-s ?nume-1 ?nume-2)
(complex ?nume-1 ? ?)
=>
(assert (calcul ?nume-2))
)

(defrule serie-NC-NC
; Un circuit serie in care ambele expresii sint
; necalculate propaga fanionul pentru ambele expresii
(calcul ?nume-s)
(serie ?nume-s ?nume-1 ?nume-2)
=>
(assert (calcul ?nume-1)
(calcul ?nume-2))
)

```

Un set analog de trei reguli vor propaga fanioane de amorsare a calculelor pentru circuite paralel. O abordare de acest fel face ca fanioanele să se multiplice plecând de jos în sus, până în nodurile expresiilor calculabile. Apoi, prin efectul priorității declarate mai mari a regulilor ce descriu operațiile serie și paralel, procesul se va inversa realizându-se o propagare a calculelor de sus în jos. Efectul general este o parcurgere *bottom-up depth-first* a grafului calculelor. Astfel, pe graful din Figura 42 se va derula următoarea secvență: fanion pe serie F (din declarație), fanion pe serie S1, fanion pe paralel P2, calcul serie S1, fanion pe paralel P1, calcul paralel P1, calcul paralel P2, calcul serie F.

Capitolul 16

Rezolvarea problemelor de geometrie

Vom explica în acest capitol cum poate fi utilizat un limbaj bazat pe reguli pentru *rezolvarea problemelor de geometrie de nivel mediu*. Ne vom limita la probleme relativ simple, în care elementele geometrice sunt doar puncte, segmente, unghiuri și triunghiuri, în care nu apar construcții ajutătoare și în care nu se cer locuri geometrice.

Punctele, segmentele, unghiurile și triunghiurile sunt definite ca obiecte cu proprietăți. Ideea de rezolvare a unei astfel de probleme are în vedere derularea următorilor pași: inițializarea spațiului faptelor cu obiecte generate din ipotezele enunțate ale problemei, urmată de o populare recursivă a spațiului cu obiecte geometrice și proprietăți ale lor, derivate din cele existente prin efectul aplicării teoremelor. În felul acesta, fie se ajunge la concluzie, caz în care programul raportează pașii utili de raționament a căror secvență constituie însuși demonstrația, fie raționamentul se stinge fără atingerea concluziei, caz în care programul anunță un eșec.

Sunt cunoscute hibeale unui raționament monoton, cu înlănțuire înainte: explozia combinatorială a obiectelor și proprietăților geometrice, cele mai multe dintre ele nefiind utile pentru raționament. Se cunoaște, de asemenea, celălalt pericol al unei explorări exhaustive a spațiului stărilor: nesiguranța că soluția găsită este cea semnificativă, în sensul de **cea mai scurtă**.

Un motiv de îngrijorare, desigur, îl poate constitui explozia exponențială a obiectelor generate, dintre care multe vor fi redundante (de exemplu, apariția punctele A și B duce la generarea atât a segmentului AB cât și a segmentului BA). Apoi, multe obiecte vor fi nerelevante pentru mersul demonstrației.

Vom începe prin a propune o modalitate de reprezentare a obiectelor geometrice, apoi vom prezenta un limbaj foarte simplu de definire a problemelor de geometrie, vom elabora maniera în care putem reprezenta teoremele de geometrie, vom găsi modalități de verbalizare a pașilor de raționament, pentru ca, în final, să ne ocupăm de procesul inferențial în sine și de posibilitatea de a controla lungimea rulării și a demonstrației⁸.

⁸ Soluția îi aparține lui Iulian Văideanu [35].

16.1. Reprezentarea obiectelor geometrice și a relațiilor dintre ele

Vom defini un punct printr-un nume – o literă:

```
(deftemplate point
  (slot name (type STRING))
)
```

Un segment este definit prin nume – o secvență de două litere – și prin lungime (aici, ca și în cazul altor valori, necunoașterea lungimii e marcată printr-o valoare negativă):

```
(deftemplate segment
  (slot name (type STRING))
  (slot size (type NUMBER) (default -1))
)
```

Un unghi se definește prin nume – o secvență de trei litere – și măsură:

```
(deftemplate angle
  (slot name (type STRING))
  (slot size (type NUMBER) (default -1))
)
```

Un triunghi este definit prin nume – o secvență de trei litere –, prin perimetru și arie:

```
(deftemplate triangle
  (slot name (type STRING))
  (slot perimeter (type NUMBER) (default -1))
  (slot area (type NUMBER) (default -1)))
```

Un punct poate fi mijlocul unui segment. Aceasta înseamnă că obiectul din clasa punct se află, pe de o parte, pe dreapta care unește capetele segmentului și, pe de altă parte, la distanță egală de capete.

Vom reprezenta relațiile dintre obiectele geometrice prin fapte de genul:

```
(fact {p|c} <relație> <obiect>*)
```

în care: *p* înseamnă că relația este dată (premisă), *c* – că ea trebuie demonstrată (concluzie), iar <relație> este un nume de relație ce există sau, respectiv, se cere a fi demonstrată între obiectele <obiect>*, date prin numele lor.

De exemplu, premisa că punctul O se află între punctele A și C pe segmentul AC se reprezintă prin:

```
(fact p between O A C)
```

iar concluzia că segmentele AB și AD sunt congruente, prin:

```
(fact c congr-segm AB AD)
```

16.2 Limbajul de definire a problemei

Există trei tipuri de notații ale limbajului de definire a problemei: declarațiile de entități geometrice, declarațiile de relații cunoscute între aceste entități, declarațiile de concluzii (ceea ce trebuie demonstrat).

Declarațiile de entități au formatul:

```
<nume-entitate> <tip-figură> [<proprietate> [...]]
```

unde:

<nume-entitate> este un șir de litere ce dă numele entității;

<tip-figură> poate fi (în versiunea restrânsă la care am convenit): punct, unghi, segment, triunghi;

<proprietate> poate fi: drept pentru unghi și dreptunghic, isoscel sau echilateral pentru triunghi.

Exemple:

```
A punct
AB segment
ABC triunghi dreptunghic isoscel
```

Declarațiile de relații pot fi clasificate în:

- congruențe:

```
AB =s MN          ; congruenta de segmente
ABC =a MNP        ; congruenta de unghiuri
ABC =t MNP        ; congruenta de triunghiuri
```

- relații între puncte și segmente:

```
O mijloc AB
O pe AB
O între A B
O intersectie AB MN
```

- relații între segmente:

```
AB || MN ; AB este paralel cu MN
```

- declarații de segmente ca elemente în unghiuri și triunghiuri

```
AD bisectoare BAC
AD inaltime ABC
AD mediana ABC
```

Pentru că nu sunt prevăzute notații separate pentru dreapta determinată de două puncte – AB, pentru segmentul închis – [AB] și pentru segmentul deschis – (AB), o aserțiune de tipul: *segmentele [AB] și [MN] se intersectează în punctul O* va trebui exprimată prin următoarea secvență:

```
O pe AB
O între A B
O pe MN
O între M N
```

Forma declarațiilor de concluzii nu diferă de cea a declarațiilor de premise, dar vom conveni să separăm concluziile de premise printr-un rând gol.

Parserul de problemă trebuie să fie capabil să transforme secvența de declarații a problemei într-o mulțime de fapte premisă și concluzie. Pentru exemplificare, să considerăm următoarea problemă: *Fie O mijlocul segmentului AC. Se consideră punctele B și D, diferite, astfel încât $\triangle OCB$ să fie egal cu $\triangle OCD$. Să se arate că segmentele AB și AD sunt congruente.* Definirea acestei probleme în mini-limbajul descris mai sus este:

```
AC segment
O punct
OCB triunghi
OCD triunghi
O mijloc AC
OCB =t OCD

AB =s AD
```

Să presupunem că o funcție de parsare (pe care nu o vom detalia aici) e capabilă să transforme acest șir de expresii în mulțimea de fapte:

```
(segment (name "AC"))
(point (name "O"))
(triangle (name "OCB"))
(triangle (name "OCD"))
(fact p between O A C)
(fact p congr-segm AO OC)
(fact p congr-tr OCB OCD)
(fact c congr-segm AB AD)
```

care exprimă într-o manieră uniformă premisele și concluziile problemei. După cum observăm, am ignorat aici declararea explicită a anumitor elemente, precum punctele A, C, B și D.

16.3 Propagarea inferențelor

Ideea unei demonstrații este ca din premise, prin aplicarea unor adevăruri cunoscute (teoreme), să regăsim concluziile. Un proces de raționament automat poate face acest lucru, dar, pentru ca demonstrația să fie urmărită, trebuie ca pașii ei să fie relevați de sistemul de raționament. Va trebui să facem deci ca orice fapt care este adăugat pe parcursul derulării demonstrației să conțină și o înregistrare a manierei în care a fost el dedus de sistem. Acești pași vor fi memorați în capătul din dreapta al faptelor (`fact ...`).

În cele ce urmează enumerăm și exemplificăm câteva categorii de reguli.

Reguli de generare de noi obiecte geometrice din cele existente precum și obiecte din relații:

- trei puncte generează un triunghi:

```
(defrule points-to-triangle
  (declare (salience 20))
  (point (name ?A))
  (point (name ?B))
  (point (name ?C))
  (test (and (not (eq ?A ?B))
              (not (eq ?B ?C))
              (not (eq ?C ?A))))
  =>
  (assert (triangle (name (str-cat ?A ?B ?C))))
)
```

- un triunghi generează trei puncte:

```
(defrule triangle-to-points
  (declare (salience 20))
  (triangle (name ?ABC))
  =>
  (assert (point (name (sub-string 1 1 ?ABC)))
          (point (name (sub-string 2 2 ?ABC)))
          (point (name (sub-string 3 3 ?ABC))))
  )
```

- două puncte generează un segment:

```
(defrule points-to-segment
  (declare (salience 20))
  (point (name ?A))
  (point (name ?B))
  (test (not (eq ?A ?B)))
  =>
  (assert (segment (name (str-cat ?A ?B))))
  )
```

- un segment generează două puncte, capetele segmentului:

```
(defrule segment-to-points
  (declare (salience 20))
  (segment (name ?AB))
  =>
  (assert (point (name (sub-string 1 1 ?AB)))
          (point (name (sub-string 2 2 ?AB))))
  )
```

- trei puncte generează un unghi:

```
(defrule points-to-angle
  (declare (salience 20))
  (point (name ?A))
  (point (name ?B))
  (point (name ?C))
  (test (and (not (eq ?A ?B))
              (not (eq ?B ?C))
              (not (eq ?C ?A)))))
  =>
  (assert (angle (name (str-cat ?A ?B ?C))))
  )
```


- un unghi generează trei puncte:

```
(defrule angle-to-points
  (declare (salience 20))
  (angle (name ?ABC))
  =>
  (assert (point (name (sub-string 1 1 ?ABC)))
          (point (name (sub-string 2 2 ?ABC)))
          (point (name (sub-string 3 3 ?ABC))))
)
```

Reguli de simetrie, reflexivitate și tranzitivitate. Aceste reguli propagă proprietățile obiectelor prin efectul simetriei, reflexivității și tranzitivității relațiilor de congruență între triunghiuri sau unghiuri sau de egalitate a mărimilor (unghiuri, lungimi). De exemplu, următoarea regulă stabilește explicit congruența a două unghiuri ce au aceeași mărime:

```
(defrule ca-equal-size
  (declare (salience 10))
  (angle (name ?ABC) (size ?s))
  (angle (name ?MNP) (size ?s))
  (test (and (not (< ?s 0)) (not (eq ?ABC ?MNP))))
  (not (fact p congr-angle ?ABC ?MNP $?any))
  =>
  (assert (fact p congr-angle ?ABC ?MNP))
)
```

Următoarea regulă dublează, prin efectul relației de simetrie, declarația de congruență a două unghiuri:

```
(defrule ca-simetry
  (declare (salience 10))
  (fact p congr-angle ?ABC ?MNP $?pdem)
  (not (fact p congr-angle ?MNP ?ABC $?))
  =>
  (assert (fact p congr-angle ?MNP ?ABC ?pdem))
)
```

Să notăm că o astfel de declarație redundantă este necesară, pentru că nu putem ști ordinea în care se propagă anumite proprietăți. Dacă segmentul AB este congruent cu segmentul MN, atunci va trebui să definim și congruența dintre segmentele MN și AB, dacă vrem să nu pierdem anumite inferențe care ar putea să fie generate de MN iar nu de AB. Din același motiv, segmentele vor fi tratate ca vectori iar unghiurile și triunghiurile vor avea un sens de citire.

În aceeași idee a definițiilor redundante dar necesare, următoarea regulă introduce în baza de fapte congruența unui unghi cu el însuși, citit invers.

```
(defrule ca-reverse
  (declare (salience 10))
  (angle (name ?ABC))
  (angle (name ?CBA))
  (test (eq ?CBA
            (str-cat
              (sub-string 3 3 ?ABC)
              (sub-string 2 2 ?ABC)
              (sub-string 1 1 ?ABC)
            )
        ))
  (not (fact p congr-angle ?ABC ?CBA $?any))
  =>
  (assert (fact p congr-angle ?ABC ?CBA))
)
```

Următoarea regulă aplică tranzitivitatea relației de congruență a unghiurilor:

```
(defrule ca-transitivity
  (declare (salience 10))
  (fact p congr-angle ?ABC ?MNP $?pdem1)
  (fact p congr-angle ?MNP ?XYZ $?pdem2)
  (test (not (eq ?ABC ?XYZ)))
  (not (fact p congr-angle ?ABC ?XYZ $?any))
  =>
  (assert
    (fact p congr-angle ?ABC ?XYZ
      (create$ ?pdem1 ?pdem2
        (str-cat ?ABC "=" ?MNP ", " ?MNP "=" ?XYZ " => "
?ABC "=" ?XYZ)
      )
    )
  )
)
```

Pentru prima oară, în această regulă apare o completare a declarației unei relații (în cazul de față, cea de congruență) cu un șir ce intenționează să verbalizeze pașii din demonstrarea acestei relații. Cu alte cuvinte, dacă unghiurile ABC și XYZ au fost dovedite a fi congruente, prin tranzitivitate, din relațiile “ABC congruent cu MNP” și “MNP congruent cu XYZ”, atunci în bază va apare un fapt:

```
(fact p congr-angle ABC XYZ <sir1> <sir2> ABC=MNP,
MNP=XYZ => ABC=XYZ)
```

unde `<sir1>` și `<sir2>` sunt șiruri de verbalizări ale demonstrației faptului că $ABC=MNP$ și, respectiv, $MNP=XYZ$.

În aceeași manieră, o seamă de reguli generează declarații de congruență a segmentelor pe baza egalității lungimilor acestora și aplică proprietățile de simetrie și tranzitivitate relației de congruență a segmentelor:

```
(defrule cs-equal-length
  (declare (salience 10))
  (segment (name ?AB) (size ?s))
  (segment (name ?MN) (size ?s))
  (test (and (not (< ?s 0)) (not (eq ?AB ?MN))))
  (not (fact p congr-segm ?AB ?MN $?any))
  =>
  (assert (fact p congr-segm ?AB ?MN))
)

(defrule cs-simetry
  (declare (salience 10))
  (fact p congr-segm ?AB ?MN $?pdem)
  (not (fact p congr-segm ?MN ?AB $?any))
  =>
  (assert (fact p congr-segm ?MN ?AB ?pdem))
)

(defrule cs-reflexivity
  (declare (salience 10))
  (segment (name ?AB))
  (not (fact p congr-segm ?AB ?AB $?any))
  =>
  (assert (fact p congr-segm ?AB ?AB))
)

(defrule cs-reverse
  (declare (salience 10))
  (segment (name ?AB))
  (segment (name ?BA))
  (test (eq ?BA (str-cat (sub-string 2 2 ?AB)
                        (sub-string 1 1 ?AB))))
  (not (fact p congr-segm ?AB ?BA $?any))
  =>
  (assert (fact p congr-segm ?AB ?BA))
)

(defrule cs-tranzitivity
  (declare (salience 10))
```

```

(fact p congr-segm ?AB ?MN $?pdem1)
(fact p congr-segm ?MN ?XY $?pdem2)
(test (not (eq ?AB ?XY)))
(not (fact p congr-segm ?AB ?XY $?any))
=>
(assert
  (fact p congr-segm ?AB ?XY
    (create$ ?pdem1 ?pdem2
      (str-cat ?AB "=" ?MN "," ?MN "=" ?XY "=>" ?AB
"=" ?XY)
    )
  )
)
)

```

Nici măcar obiectele geometrice degenerate, ca triunghiurile applatizate, de exemplu, nu pot fi eliminate. Ar fi foarte ușor să scriem o regulă ca următoarea:

```

(defrule elim-degen-tr
  (declare (salience 80))
  ?t <- (triangle (name ?ABC))
  (fact p on ?A ?BC)
  (test (and
    (eq ?A (sub-string 1 1 ?ABC))
    (eq ?BC (sub-string 2 3 ?ABC))
  ))
=>
  (retract ?t)
)

```

dar, o dată un astfel de triunghi eliminat, condițiile de regenerare a lui vor fi din nou satisfăcute, ceea ce va duce la intrarea în bucle infinite.

Reguli de geometrie. Regulile din această clasă sunt cele mai interesante. Ele propagă relații dintre obiecte și proprietăți ale acestora pe baza teoremelor din geometrie. De exemplu, un număr de reguli tratează cazurile de congruență ale triunghiurilor.

Cazul latură-unghi-latură:

```

(defrule ct-LUL
  (fact p congr-angle ?ABC ?MNP $?pdem2)
  (fact p congr-segm ?AB ?MN $?pdem1)
  (test (and
    (eq ?AB (sub-string 1 2 ?ABC))
    (eq ?MN (sub-string 1 2 ?MNP))
  ))
)

```

```

))
(fact p congr-segm ?BC ?NP $?pdem3)
(test (and
      (eq (sub-string 2 3 ?ABC) ?BC)
      (eq (sub-string 2 3 ?MNP) ?NP)
    ))
(not (fact p congr-tr ?ABC ?MNP $?any))
=>
(assert
  (fact p congr-tr ?ABC ?MNP
    (create$ ?pdem1 ?pdem2 ?pdem3
      (str-cat ?AB "=" ?MN "," ?ABC "="
?MNP "," ?BC "=" ?NP ">" ?ABC "#" ?MNP "(LUL)"))
    )
  )
)
)

```

Ca și mai sus, regula completează declarația de congruență adăugată în bază, cu pașii care au dus la demonstrarea ei. În cazul de față, la pașii care au dus la demonstrarea egalității segmentelor AB și MN, a congruenței unghiurilor ABC și MNP și a egalității segmentelor BC și NP, se daugă acest ultim pas care, din cele trei ipoteze, trage concluzia că triunghiurile ABC și MNP sunt congruente.

Următoarea regulă exprimă o proprietate a triunghiurilor isoscele: faptul că din egalitatea a două unghiuri se poate trage concluzia egalității laturilor adiacente:

```

(defrule congr-angles-to-sides
  (fact p congr-angle ?ABC ?ACB $?pdem)
  (test (and
        (eq (sub-string 1 1 ?ABC) (sub-string 1 1 ?ACB))
        (eq (sub-string 2 2 ?ABC) (sub-string 3 3 ?ACB))
        (eq (sub-string 3 3 ?ABC) (sub-string 2 2 ?ACB))
      ))
  (segment (name ?AB))
  (test (eq ?AB (sub-string 1 2 ?ABC)))
  (segment (name ?AC))
  (test (eq ?AC (sub-string 1 2 ?ACB)))
  (not (fact p congr-segm ?AB ?AC $?any))
=>
  (assert
    (fact p congr-segm ?AB ?AC
      (create$ ?pdem (str-cat ?ABC "=" ?ACB ">"
?AB "=" ?AC "(isoscel)"))
    )
  )
)
)

```

16.4. Lungimea rulării și a demonstrației

Un fapt concluzie, o dată demonstrat, este eliminat din bază. Procesul de inferență se oprește când toate concluziile au fost demonstrate:

```
(defrule stop-inference
  (declare (salience 50))
  (not (fact c $?any))
=>
  (printout t "Toate concluziile au fost demonstrate."
crlf)
  (halt)
)
```

Într-o abordare cum este cea de față, care urmărește o parcurgere cvasi-exhaustivă a spațiului stărilor în vederea producerii unei demonstrații, este firesc să ne preocupe lungimea rulării și a soluției generate. Nu este evident că acestea sunt direct proporționale.

Este neîndoios că dacă o soluție există, pentru sistemul de proprietăți geometrice stabilite, ea va fi găsită, pentru că rularea se face cu un motor monoton. Strategia de rezoluție a conflictelor poate juca însă un rol important în viteza de găsim a soluției, cât și în lungimea demonstrației. O aplicare a regulilor în maniera **întâi-în-adâncime** poate duce la explorări foarte detaliate în cotloane neinteresante ale spațiului stărilor, deci poate genera soluții lungi. Adoptarea strategiei **întâi-în-lărgime** echivalează cu o căutare cu rază constantă în toate direcțiile, deci ar trebui, în principiu, să găsească cele mai scurte soluții. Nu e exclus însă ca acestea să fie găsite mai greu.

Rularea exemplului definit mai sus în strategia **întâi-în-lărgime** provoacă aprinderea a 584 de reguli care generează nu mai puțin de 445 de obiecte geometrice și relații între ele, plecând de la cele declarate inițial. Marea majoritate a acestor obiecte sunt însă aberații de genul unghiurilor și triunghiurilor aplatizate (de exemplu AOC) sau multiplicări inutile de obiecte obținute prin simple permutări de litere (triunghiurile OCB, OBC, CBO, COB etc.). În final însă se produce următoarea demonstrație:

```
CLIPS> (run)
Concluzia (congr-segm "AB" "AD") rezulta astfel:

O@AC=>ACB=OCB
OCB#OCD=>OCB=OCD
ACB=OCB, OCB=OCD=>ACB=OCD
O@AC=>OCD=ACD
ACB=OCD, OCD=ACD=>ACB=ACD
CBO#CDO=>CB=CD
```

```

AC=AC, ACB=ACD, CB=CD=>ACB#ACD (LUL)
BAC#DAC=>BA=DA
AB=BA, BA=DA=>AB=DA
AB=DA, DA=AD=>AB=AD
---

Toate concluziile au fost demonstrate.
CLIPS>

```

În acest raport, semnul @ semnifică apartenența, # – congruența de triunghiuri, iar = egalitatea de unghiuri.

Aceeași intrare, dar rulată cu strategia **întâi-în-adâncime**, deși produce un număr nesemnificativ diferit de obiecte și relații, prin aproximativ tot atâtea aprinderi de reguli, rezultă într-o demonstrație mai lungă:

```

CLIPS> (run)
Concluzia (congr-segm "AB" "AD") rezulta astfel:

BCO#DCO=>BCO=DCO
O@AC=>OCB=ACB
OCB=ACB, ACB=BCA=>OCB=BCA
BCO=OCB, OCB=BCA=>BCO=BCA
ACB=BCA, BCA=BCO=>ACB=BCO
DCO=BCO, BCO=ACB=>DCO=ACB
O@AC=>ACD=OCD
DCA=ACD, ACD=OCD=>DCA=OCD
DCO=OCD, OCD=DCA=>DCO=DCA
ACD=DCA, DCA=DCO=>ACD=DCO
ACB=DCO, DCO=ACD=>ACB=ACD
BCO#DCO=>BC=DC
DC=BC, BC=CB=>DC=CB
CB=DC, DC=CD=>CB=CD
AC=AC, ACD=ACB, CD=CB=>ACD#ACB (LUL)
ADC#ABC=>AD=AB
---

Toate concluziile au fost demonstrate.
CLIPS>

```


Capitolul 17

Sfaturi de programare bazată pe reguli

17.1. Recomandări de stil în programare

Mai multe reguli mici, în care fiecare se ocupă de un aspect elementar, dar bine definit, sunt mai bune decât mai puține reguli complicate, fiecare rezolvând o porțiune amorf definită a problemei.

Evitați regulile care încorporează în părțile drepte decizii ce s-ar putea implementa prin comparații ale șabloanelor asupra faptelor, deci ca mai multe reguli elementare. O rezolvare elegantă, exploatând principiile programării bazate pe reguli, adesea aduce mai multe beneficii decât o rezolvare aparent mai eficientă care impurifică regulile cu părți mari de cod aparținând paradigmei imperative.

Evitați o exploatare exagerată a nivelurilor de prioritate. Deși limbajele bazate pe reguli oferă o plajă imensă de valori de prioritate, personal n-am întâlnit probleme care să fi necesitat mai mult de șapte niveluri, iar cazurile cele mai frecvente sunt acelea în care două, maximum trei niveluri sunt suficiente. Apoi, regulile plasate pe același nivel de prioritate trebuie să fie caracterizate de o anumită structură comună a șabloanelor care să se circumscrie indicației enunțate în capitolul 7 relativ la acoperirea parțială a condițiilor.

În general, structura unui program bazat pe reguli este aceea a unui automat cu stări (faze), în care tranziția între faze poate să însemne comutarea conținutului unui fapt ce păstrează numele fazei. Execuția caracteristică unei faze este formată fie din ciclări fie din tranziții în alte faze. Uneori o fază poate fi caracterizată de subfaze, caz în care o structură de mai multe fapte poate memora plasarea exactă a execuției într-o anumită subfază a unei subfaze a unei faze... Indiferent de nivelul de imbricare a unei astfel de organizări, vom considera că nivelul execuției caracterizat de satisfacerea unui anumit subscop al problemei se cheamă tot fază, menirea declarației de prioritate fiind realizarea unui control al execuției doar în mulțimea regulilor ce sunt caracteristice unei faze.

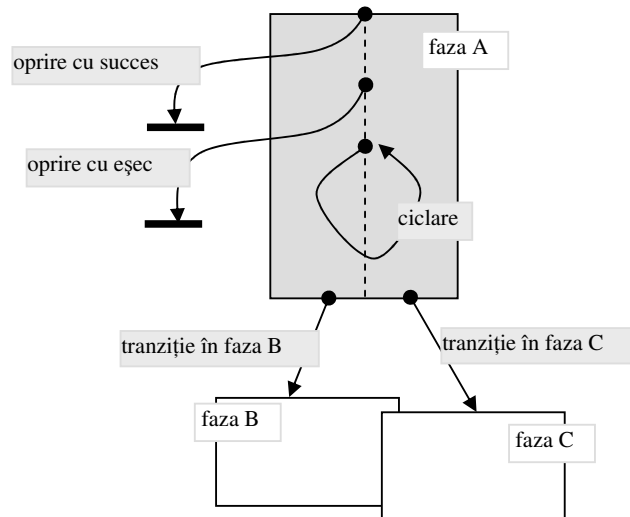


Figura 43: O configurație posibilă de reguli caracteristice unei anumite faze A din evoluția unui sistem

Astfel, Figura 43 sugerează o posibilă organizare a unui sistem, la nivelul unei faze A, în care există următoarele posibilități de evoluție a calculului:

- oprire pe succes;
- oprire cu eșec;
- ciclare în interiorul fazei;
- tranziție într-o stare B și
- tranziție într-o stare C.

Diferite înălțimi considerate pe linia întreruptă de simetrie verticală ce ilustrează faza A, sugerează niveluri de prioritate grupate în jurul priorității de nivel 0, atașată regulii de ciclare (considerată centrală pentru semantica fazei), după cum urmează:

```
(defrule terminare_cu_succes
  (declare (salience 20))
  ... condiții de terminare cu succes
  =>
  ...
)

(defrule terminare_cu_eșec
  (declare (salience 10))
  ... condiții de terminare cu eșec, potențial mai slabe
    decât ale regulii de terminare cu succes
```

```

=>
...
)

(defrule ciclare
  ... condiții de ciclare, potențial mai slabe decât ale
    regulii de terminare cu eșec
  =>
  ...
)

(defrule tranzitie_in_faza_B
  (declare (salience -10))
  ... condiții de tranziție în faza B, potențial mai slabe
    decât ale regulii de ciclare și complementare față
    de cele ale regulii de tranziție în faza C
  =>
  ...
)

(defrule tranzitie_in_faza_C
  (declare (salience -10))
  ... condiții de tranziție în faza C, potențial mai slabe
    decât ale regulii de ciclare și complementare față
    de cele ale regulii de tranziție în faza B
  =>
  ...
)

```

Execuția sugerată de cazul prezentat este una în care, la intrarea în faza A, se testează întâi terminarea rulării. Dacă aceasta nu se confirmă, regula de terminare cu succes este următoarea care ar putea fi luată în considerare. Abia dacă nici condițiile ei nu sunt verificate, se execută regula de ciclare, atâta timp cât își mai satisface încă condițiile de aplicare. Când aceasta nu se mai aplică, una dintre regulile de tranziție în faza B sau C se poate aplica. Condițiile lor fiind presupuse complementare, numai una dintre acestea poate fi satisfăcută la un anumit moment dat. De remarcat că pe parcursul rulării regulii de ciclare nu se mai poate efectua o terminare cu succes sau eșec doar atâta timp cât modificările produse la fiecare pas de regula de ciclare nu vor satisface condițiile regulilor de terminare, presupuse mai tari.

17.2. Erori în execuția programelor CLIPS

De câte ori, compilând ori executând programele scrise de noi, nu am fost tentați să credem că mașina cu care lucrăm “a luat-o prin bălării”, pentru că ceea ce rezulta nu semăna cu ceea ce gândeam că ar trebui să rezulte. Și de câte ori nu am recunoscut, înfrânți dar luminați, că mașina avea dreptate (ori că *eroarea era corectă*, așa cum, hâtru, se exprima un ilustru student al meu⁹).

Ceea ce urmează este un scurt compendiu de erori comentate, întâlnite în rularea programelor CLIPS.

Eroare de aliniere a unei variabile din șablon cu câmpurile unui fapt

```
Function < expected argument #2 to be of type integer or
float
[DRIVE1] This error occurred in the join network
      Problem resides in join #2 in rule(s):
      sortare-mijloc
[PRCCODE4] Execution halted during the actions of defrule
sortare-extrema-dreapta.
```

Eroarea a apărut într-o regulă ce conținea următoarele șabloane:

```
?lst <- (lista-finala $?prime ?un-nume ?o-val ?alt-nume
?alt-val $?ultime)
?pct <- (punctaj (nume ?num)
            (valoare ?val&:(and (< ?val ?o-val)
                                (> ?val ?alt-val))))
```

Intenția era de a căuta în lista `lista-finala`, în care alternau perechi de valori simbolice cu valori numerice, două astfel de perechi între ale căror valori numerice să poată fi plasată o altă valoare luată din lista `punctaj`.

Necazul este că perechea de variabile `?un-nume ?o-val` din primul șablon se poate potrivi pe o pereche care începe cu o valoare numerică și continuă cu o valoare simbolică, astfel încât valoarea comparată nu poate fi argument într-un test numeric.

Corecția constă în plasarea unui filtru suplimentar care să “centreză” variabilele pe șir în așa fel încât valorile numerice să fie în urma celor simbolice:

```
?lst <- (lista-finala $?prime
            ?un-nume ?o-val&:(numberp ?o-val)
            ?alt-nume ?alt-val
            $?ultime)
```

⁹ Dan Gurău, pe unde vei mai fi hălăduind?

```
?pct <- (punctaj (nume ?num)
              (valoare ?val&:(and (< ?val ?o-val)
                                   (> ?val ?alt-val))))
```

Eroare de folosire a variabilelor index

Să considerăm o situație în care se dorește memorarea într-o listă a indexului unui fapt iar nu a faptului ca atare, urmând ca apoi să se încerce a se identifica, în faptul a căruia adresă am salvat-o în acest mod, anumite câmpuri. Următoarea tentativă eșuează:

```
(defrule start
  ?sc <- (stare (tip curenta))
  =>
  (assert (OPEN ?sc)
)

(defrule selectie-stare-curenta
  (OPEN ?prim $?rest)
  ?prim <- (stare (predicate $?preds))
  =>
  ...
)
```

Eroarea raportată este una de compilare și este generată de imposibilitatea de utilizare a unei variabile index ce a fost deja legată într-o condiție-șablon:

```
[ANALYSIS2] Pattern-address ?prim used in CE #2 was
previously bound within a pattern CE.
```

```
ERROR:
(defrule MAIN::selectie-stare-curenta
  (OPEN ?prim $?rest)
  ?prim <- (stare (predicate $?preds))
  =>
  (printout "predicate in starea curenta " $?preds
  crlf))
```

O soluție, care ocolește problema pentru că evită folosirea indecșilor faptelor, constă în a marca în locul indexului un simbol unic, de exemplu generat cu (gensym), ca aici (unde nume este slotul care păstrează acest simbol-adresă):

```
(defrule start
  ?sc <- (stare (tip curenta))
  =>
```

```

(bind ?sym (gensym))
(modify ?sc (nume ?sym))
(assert (OPEN ?sym)
)

(defrule selectie-stare-curenta
  (OPEN ?prim $?rest)
  (stare (nume ?prim) (predicate $?preds))
=>
  (printout t "predicate in starea curenta " $?preds
crlf)
)

```

17.3. Așteptări neîndeplinite

Lăsat să ruleze un anumit timp, intenționam, cu următorul program, să găsească *limitele între care generează numere funcția random*:

```

(deffacts min-max
  (max 25000)
  (min 25000)
)

(defrule act
  ?max <- (max ?x)
  ?min <- (min ?y)
=>
  (bind ?r (random))
  (if (> ?r ?x) then (retract ?max) (assert (max ?r)))
  (if (< ?r ?y) then (retract ?min) (assert (min ?r)))
)

```

Urmărirea rulării arată însă că regula se aplică de un număr foarte mic de ori. În fapt, ea se aplică până se întâmplă prima oară ca apelul funcției `random` să dea un număr în interiorul intervalului `min-max`. Pentru că în acel moment nici unul din fapte nu se mai modifică și principiul refractabilității împiedică o nouă aplicare a ei.

Repararea constă în modificarea de fiecare dată a faptelor `min` și `max`, chiar și atunci când valorile pe care le memorează nu se schimbă:

```

(deffacts min-max
  (max 25000)
  (min 25000)
)

```

```
(defrule act-v1
  ?max <- (max ?x)
  ?min <- (min ?y)
  =>
  (retract ?max ?min)
  (bind ?r (random))
  (if (> ?r ?x) then (assert (max ?r) (min ?y))
      else (if (< ?r ?y) then (assert (min ?r) (max ?x))
                  else (assert (min ?y) (max ?x))))
)
```

Pentru peste 40000 de aplicări ale regulii, extremele astfel găsite au fost 2 și 32766, dar limitele reale sunt între 1 și 2^{15} .

Bibliografie

1. AIKINS, J. S., KUNZ, J. C. , SHORTLIFFE, E. H. FALLAT, R. J. *PUFF: an expert system for interpretation of pulmonary function data*. Computers and Biomedical Research 16:199-208, 1983.
2. BARR, A., E., FEIGENBAUM, A., COHEN, P. (eds.) *The Handbook of Artificial Intelligence*, Volumes 1-3. Los Altos, CA: Kaufmann, 2 vol., 1981, 1982.
3. BENNETT, J. S. *ROGET: a knowledge-based system for acquiring the conceptual structure of an expert system*. Journal of Automated Reasoning 1(1):49-74, 1985.
4. BUCHANAN, B. G.; FEIGENBAUM, E. A. *DENDRAL and META-DENDRAL: their applications dimensions*. Artificial Intelligence 11:5-24, 1978.
5. BUCHANAN, B. G.; MITCHELL, T. *Model directed learning of production rules*. In D. A. Waterman and F. Hayes-Roth (eds.), *Pattern-Directed Inference System*. New York: Academic Press, 1978.
6. BUCHANAN, B. G., SHORTLIFFE, E. H. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley, 1984.
7. BUCHANAN, B.; SMITH, R.. *Fundamentals of Expert Systems*, Annual Review of Computer Science 3, 23-58, 1988.
8. COOPER, T., WOGGIN, N. *Rule-based Programming with OPS5*, Morgan Kaufmann Publishers, 1988.
9. CRISTEA, D. *Probleme de analiza limbajului natural*. În curs de apariție la Editura Universității "Alexandru Ioan Cuza" Iași, 2002.
10. CRISTEA, D., GALESCU, L., BACALU, C. *L-Exp: A Language for building NLP applications*, Proceedings of 14th International Avignon Conference Natural Language Processing, AI'94, Paris, 1994, pp. 97-107.
11. DAVIS, R., LENAT, D. *Knowledge-Based Systems in Artificial Intelligence: AM and TEIRESIAS*. New York: McGraw-Hill, 1982.
12. FAGAN, L. M. *VM: representing time-dependent relations in a medical setting*. Memo HPP 831 (Knowledge Systems Laboratory), June 1980.
13. FORGY, C. L. *OPS5 User's Manual*, Technical Report, CMU-CS-81-135, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1981.
14. FORGY, C. L. *RETE: A fast algorithm for the many pattern/many object pattern match problem*, Artificial Intelligence 19(1):17-37, September 1982.

15. GÂLEA, D., COSTIN, M., ZBANCIOC, M. *Programarea în CLIPS prin exemple*. Editura Tehnopress, 2001.
16. GIARRATANO, J.; RILEY, G. *Expert Systems Principles and Practice*", ediția a doua, PWS Publishing, Boston, MA., 1994, 644 pp.
17. GRAY, N. A. B.; SMITH, D. H.; VARKONY, T. H.; CARHART, R. E.; BUCHANAN, B. G. *Use of a computer to identify unknown compounds: the automation of scientific inference*. Chapter 7 in G. R. Waller and O. C. Dermer, eds., *Biomedical Application of Mass Spectrometry*. New York: Wiley, 1980.
18. GRIES, D. *The Science of Programming*. Springer-Verlag, 1981.
19. HASLING, D., CLANCEY, W. J., RENNELS, G. *Strategic explanations for a diagnostic consultation system*. International Journal of Man-Machine Studies 20(1):3-19, 1984.
20. JACKSON, P. *Introduction to Expert Systems, 3rd Edn*. Harlow, England: Addison Wesley Longman, 1999.
21. KATZ, S.S. *Emulating the Prospector expert system with a raster GIS*. Computer and Geosciences, 18., 1991.
22. KRIVINE, J.P., DAVID, J.M. *L'Acquisition des Connaissances vue comme une Processus de Modélisation: Méthodes et Outils*. Intellectica (12), 1991.
23. LENAT, D. *The nature of heuristics*. Artificial Intelligence 19(2): pp. 189-249, 1981.
24. LENAT, D. *EURISKO: a program that learns new heuristics and domain concepts*. Artificial Intelligence 21(2):61-98 (1983).
25. LINDSAY, R. K., BUCHANAN, B. G., FEIGENBAUM, E. A., LEDERBERG, J. *Application of Artificial Intelligence for Chemistry: The DENDRAL Project*. New York: McGraw-Hill, 1980.
26. LUCANU, D. *Proiectarea algoritmilor. Tehnici elementare*, Ed. Universității "Al. I. Cuza" Iași, 1993.
27. MERRITT, D. *Building Expert Systems in Prolog*, Springer-Verlag, 1989.
28. MIRANKER, DANIEL P. *TREAT: A better match algorithm for AI production systems*. Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), pp. 42-47, 1987.
29. MIRANKER, DANIEL P. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, Morgan Kaufmann Publishers, 1990.
30. MOORE, T., MORRIS, K., BLACKWELL, G. *COAMES - Towards a Coastal Management Expert System*, Proceedings of the International Conference on GeoComputation, University of Leeds United Kingdom, 17 - 19 September 1996.
31. PERLIN, M. *The match box algorithm for parallel production system match*, Technical Report CMU-CS-89-163, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1989.

32. PUPPE, F. *Systematic Introduction to Expert Systems: Knowledge Representation and Problem Solving Methods*. Berlin: Springer, 1993.
33. RICH, E., KNIGHT, K. *Artificial Intelligence*, McGraw Hill, 1991.
34. SLAGE, J., WICK, M. *A method for evaluating expert system applications*, AI Magazine 9, 1988.
35. VĂIDEANU, I. *GEOM-2 – Sistem expert pentru rezolvarea problemelor de geometrie în plan*. Teză de licență, Facultatea de Informatică, Universitatea "A.I.Cuza" Iași, 2001.
36. YAO, SUK I.; KIM, II KON. *DIAS1: An Expert System for Diagnosing Automobiles with Electronic Control Units*, Expert Systems with Applications, 4(1), pp. 69-78, 1992.