Cap3. Optimizarea programelor

3.1. Strategii de ordonare a regulilor în agendă

Agenda este o listă de reguli ce au condițiile deja satisfăcute, dar care nu au fost încă executate. În cazul în care se practică o programare modulară e bine de reținut faptul că fiecare modul are propria sa agendă. Putem asemăna agenda cu o stivă în care regula cea mai din vârf este prima executată. Modificările în cadrul agendei sunt dinamice, astfel apariția sau dispariția unui fapt din lista de fapte poate atrage după sine apariția sau dispariția unor reguli în agendă.

Când o regulă este recent activată plasarea ei în agendă se bazează, în ordine, pe următorii factori:

- O regulă nou activată este plasată înaintea tuturor regulilor cu prioritate mai mică şi după toate regulile cu prioritate mai mare (salience).
- În cazul în care avem reguli cu aceeaşi prioritate este folosită strategia actuală a motorului de inferență pentru a stabili locul regulii actuale între celelalte reguli cu salience egal.
- Dacă o regulă este activată de o aceeaşi aserțiune (ca şi alte reguli) sau de ştergerea unui fapt şi primele două criterii nu sunt capabile să stabilească o ordine, atunci se va realiza arbitrar (nu aleator) o ordonare a regulii în raport cu celelalte reguli echivalente. Este de recomandat a nu se ajunge în situația în care funcționarea corectă a unui program să fie dependentă de această ordine arbitrară.

CLIPS-ul oferă şapte strategii de căutare: **depth**, **breadth**, **simplicity**, **complexity**, **lex**, **mea**, şi **random**. În cazul în care nu se specifică o anumită strategie cea folosită în mod implicit va fi *depth*. Strategia curentă poate fi setată folosindu-se comanda **set-strategy**. Cu ajutorul comenzii **get-strategy** se poate afla care este strategia curentă.

Strategia **DEPTH** – *o regulă nou activată este plasată înaintea regulilor care au aceeaşi prioritate*. De exemplu, dacă în agendă avem regula-1 activată de faptul-a şi regula-2 activată de faptul-b, atunci dacă faptul-a este asertat înainte de faptul-b (are adresa de fapt mai mică) atunci *regula-2 va fi amplasată înainte de regula-1 în agendă.*

Strategia **BREADTH** – *o regulă nou activată este plasată sub regulile de aceeaşi prioritate*. De exemplu dacă în agendă avem regula-1 activată de faptul-a şi regula-2 activată de faptul-b, atunci dacă faptul-a este asertat înainte de faptul-b (are adresa de fapt mai mică) *atunci regula-2 va fi amplasată după regula-1 în agenda*.

Strategia **SIMPLICITY** – *o regulă nou activată este amplasată între regulile cu aceeaşi prioritate înaintea regulilor care au un număr mai mare sau egal de comparații de efectuat în LHS. Cu alte cuvinte criteriul de selecție este*

simplitatea. Comparațiile cu o constantă, legările între variabile, apelurile de funcție dintr-un element condițional de test, câmp condițional predicativ, sau câmp condițional valoare returnată =, sporesc complexitatea unei reguli. Folosirea funcțiilor booleane *and*, *or* şi *not* nu sporeşte complexitatea, însă argumentele lor pot să o mărească. Funcțiile apelate din interiorul altei funcții nu sunt luate în calcul.

De exemplu regula următoareare gradul de complexitate egal cu 5. Comparația cu o constantă, comparația lui ?x cu variabila ?x anterior legată, apelul funcților *numberp*, < şi > aduc un total de 5. Apelul funcției + realizat din interiorul altei funcții, precum şi funcția *and* nu se adaugă la complexitatea regulii după cum s-a afirmat şi mai sus.

```
(defrule example
        (item ?x ?y ?x)
        (test  (and  (numberp ?x)  (> ?x (+ 10 ?y))  (< ?x 100) ) )
         =>)
```

Strategia **COMPLEXITY** – *o regulă nou activată este amplasată între regulile cu aceeaşi prioritate, înaintea regulilor care au un număr mai mic sau egal de comparații de efectuat în LHS..Deci criteriul de selecție este complexitatea.*

Strategia **LEX** – *o regulă nou activată este amplasată între regulile cu aceeaşi prioritate folosindu-se strategia OPS5*. Mai întâi se evaluează cât de recente sunt entitățile din pattern-ul unei reguli. Fiecare fapt şi instanță este marcată intern cu o etichetă de timp "tag time" pentru a indica cât de recent este acesta relativ la celelalte fapte sau instanțe din sistem. Entitățile din pattern-ul regulii sunt sortate în ordine descrescătoare. O activare cu entitățile din pattern mai recente este plasată înaintea celor mai puțin recente. Pentru a determina ordinea de amplasare a două activări se compară etichetele de timp sortate descrescător una câte una. Comparația continuă până când o etichetă de timp este mai mare decât perechea sa. Activarea cu cea mai mare etichetă de timp este amplasată înaintea celeilalte activări în agendă. Dacă toate etichetele comparate sunt identice, dar o activare are mai multe entități, atunci aceasta va fi amplasată înaintea celeilalte. Dacă şi numărul de entități este egal, atunci este luată în calcul complexitatea regulii, astfel încât cea cu un grad de complexitate mai mare este amplasată în fața celeilalte. În cazul folosirii elementului conditional **not** (fapt semnalat de apariția unei virgule la sfârşitul unei activări) eticheta de timp este întotdeauna mai mică decât cea a unei entități din pattern.

Fie următorul exemplu cu 6 activări ordonate de strategia LEX . In partea dreaptă se observă o ordonare a indicilor, folosită în comparația etichetelor de timp, transparentă pentru utilizator:

rule-6: f-1,f-4	rule-6: f-4,f-1
rule-5: f-1,f-2,f-3,	rule-5: f-3,f-2,f-1,
rule-1: f-1,f-2,f-3	rule-1: f-3,f-2,f-1
rule-2: f-3,f-1	rule-2: f-3,f-1
rule-4: f-1,f-2,	rule-4: f-2,f-1,
rule-3: f-2,f-1	rule-3: f-2,f-1

Strategia **MEA** – *o regulă nou activată este amplasată între regulile cu aceiaşi prioritate folosindu-se strategia OPS5. Strategia MEA este negata strategiei LEX, deci toate afirmaţiile făcute anterior sunt valabile cu modificările de rigoare*. Se dă un exemplu cu 6 activări ordonate de acestă strategie.

rule-2: f-3,f-1	rule-2: f-1,f-3
rule-3: f-2,f-1	rule-3: f-1,f-2
rule-6: f-1,f-4	rule-6: f-1,f-4
rule-5: f-1,f-2,f-3,	rule-5: f-1,f-2,f-3,
rule-1: f-1,f-2,f-3	rule-1: f-1,f-2,f-3
rule-4: f-1,f-2,	rule-4: f-1,f-2,

Strategia **RANDOM** – *fiecărei activări îi este asociat un număr aleator utilizat pentru a stabili ordinea între regulile cu aceiaşi prioritate*. Acest număr este păstrat în momentul în care este schimbată strategia, astfel încât revenirea la această strategie va conduce la aceeaşi ordine a activărilor în agendă.

3.2. Problema turnurilor din HANOI – continuare

Au fost enumerate anterior toate cele 7 tipuri de strategii, deoarece în unele probleme este vital pentru o funcționare corectă a programului alegerea unei strategii adecvate. Vom da un mic exemplu pentru a observa cum modul de activare a regulilor în agenda poate modifica total execuția unui program.

```
(defrule regula_1
        ?f <- (exemplu)
        (not (nu_sterge))
        =>
        (retract ?f) )

(defrule regula_2
        (exemplu)
        =>
        (printout  t  "Acesta este un exemplu...") )

CLIPS> (assert (exemplu))                    //strategia implicită depth
CLIPS> (run)
CLIPS> (set-strategy breadth)
CLIPS> (assert (exemplu))
CLIPS> (run)
Acesta este un exemplu...
```

Când strategia folosită a fost cea implicită *depth*, ordinea activărilor din agenda a fost regula-1 şi apoi regula-2, şi după ce a fost executată prima regulă şi a fost retractat faptul *(exemplu)*, regula-2 a fost şi ea ştearsă din agendă (nu mai există în lista de fapte un fapt care să se potrivească cu pattern-ul regulii). Pentru cea de-a doua strategie, ordinea activărilor în agendă este inversată: mai întâi va fi lansată în execuție regula-2 şi apoi regula-1. În acest fel se explică afişarea mesajului pentru cel de-al doilea caz.

În următorul exemplu cu ajutorul unei reguli vom aserta cele 10 cifre arabe şi apoi aceste vor fi afişate una câte una de cea de-a doua regulă.

```
(defrule create_digit
        =>
        (loop-for-count (?i 0 9)
                (assert (digit ?i)) ) )

(defrule print_digit
        (digit ?d)
        =>
        (printout  t  ?d) )

CLIPS> (run)                          //strategia implicită depth
9876543210
CLIPS> (set-strategy breadth)
CLIPS> (run)
0123456789
```

Explicația diferențelor obținute în cazul celor două rulări pentru strategii diferite este de asemenea legată de modul de ordonare a activărilor în agenda. După cum se poate observa ordinea activărilor regulilor în agenda este inversă pentru cele două tipuri de strategii *depth* şi *breadth*. Pentru varianta următoare a problemei turnurilor din Hanoi pentru o execuție corectă este absolut necesară setarea strategiei **breadth**. Pentru varianta de CLIPS sub Windows schimbarea strategiei motorului de inferență se poate realiza direct prin selecția lui **Options...** din meniul **Execution**. În această variantă se va încerca implementarea unor stive cu ajutorul unor fapte a căror construcție deftemplate ar putea fi următoarea:

```
(deftemplate stiva
        (slot numar)
        (slot nume)
        (multislot valori) )
```

Dacă se doreşte un număr variabil de discuri, care să fie introdus de utilizator de la tastatură, atunci trebuiesc modificate variabila globală ?*N* (numărul de discuri) şi stiva inițială din construcția deffacts.

```
(defglobal ?*N* = 3)                  ;numarul de discuri

(deffacts date-initiale               ; in stiva initiala avem 3 discuri
        (stiva (numar 1)  (nume A) (valori 3 2 1) )
        (stiva (numar 2) (nume B) )
        (stiva (numar 3) (nume C) )  )
```

Dacă dorim un număr variabil de discuri vom modifica valoarea variabilei globale ?*N* şi valorile din stiva inițială (putem utiliza comanda *modify* sau putem şterge de tot stiva inițială şi să asertăm noua stivă când numărul de discuri este cunoscut). Pentru o funcționare corectă nu trebuie să avem două stive inițiale. Regula ar putea eventual verifica şi dacă numărul introdus este un întreg cuprins într-un interval dat.

```
(defrule interogare_utilizator
        (declare (salience 10))
        =>
        (printout t "Doriti un alt numar de discuri (N=3) Y/N ? ")
        (bind ?flag (readline))
        (if (neq ?flag "y" "Y" "yes" "Yes" "YES")
                then else (assert (schimba_numar_discuri) ) )
)

(defrule introducere_numar_discuri
        (declare (salience 10))
        ?f <- (stiva (numar 1)  (valori 3 2 1))
        ?new <- (schimba_numar_discuri)
        =>
        (retract ?new)
        (printout t "Introduceti numarul de discuri N (1..9) : ")
        (bind ?*N* (read))
        (while (or (not (integerp ?*N*)) (< ?*N* 1) (> ?*N* 9))
                (printout t "Introduceti numarul de discuri N (1..9) : ")
                (bind ?*N* (read)))
        (bind $?val (create$))
        (loop-for-count (?i 1 ?*N*)
                (bind $?val (create$ ?i $?val)) )
        (modify ?f (valori $?val))
)
```

Regulile anterioare în care se introduce de la tastatură numărul de discuri din stiva inițială poate lipsi, utilizatorul realizând toate modificările manual în faptele inițiale din construcția *deffacts*. Stiva inițială nu trebuie să fie neapărat prima stiva, în cazul în care se doreşte modificarea acesteia, trebuiesc modificați şi parametrii funcției recursive, ce furnizează programului mutările ce vor fi realizate.

```
(deffunction letter (?no)
     (nth$ ?no (create$ A B C D E F G H I J K L M N O P Q R S T U V X Z)) )

(deffunction  hanoi (?n  ?tija_start  ?tija_end)
        (if (> ?n 1)
         then (hanoi  (- ?n 1)  ?tija_start   (- 6 ?tija_start ?tija_end)) )
        (assert  (move  ?n  ?tija_start  ?tija_end) )
        (if (> ?n 1)
         then (hanoi  (- ?n 1)  (- 6 ?tija_start ?tija_end)  ?tija_end))
)
```

De această dată funcția Hanoi va aserta un fapt în care se oferă informații despre mutarea care trebuie realizată. După apelul funcției recursive vom avea asertate toate mutările ce trebuiesc realizate. Aplicându-se o strategie *depth*, primul fapt *move* care va fi analizat va fi de fapt ultima mutare. Este necesară aplicarea unei strategii care să permită folosirea faptelor *move* în ordinea în care au fost

asertate, de către regula care va modifica cele trei stive. O astfel de strategie este **breadth**, fără o alegere convenabilă a strategiei programul funcționând incorect.

Foarte important de reținut este faptul că în CLIPS în mod implicit nu se pot introduce două fapte identice. Pentru a rezolva problema turnurilor din Hanoi pentru un număr de discuri N mai mare, în mod sigur va trebui să efectuăm mutări identice în momente diferite de timp (ex: pentru N=3 mutarea discului 1 de la A la C se efectuează şi la început şi la sfârşit. CLIPS-ul nu poate aserta două fapte identice şi execuția programului în aceste condiții este incorectă. Se poate modifica dualitatea faptelor selectând **Options...** în meniul **Execution** sau cu ajutorul funcției **set-fact-duplication**.

```
(defrule start_program
        =>
        (set-strategy breadth)
        (set-fact-duplication TRUE)
        (hanoi ?*N* 1 3)
)
```

În regula următoare se va face o verificare suplimentară a corectitudinii mutării. Astfel regula se va aprinde numai dacă discul ce se doreşte a fi mutat se află în vârful stivei de start. Odată efectuate modificările în stive, putem şterge din lista de fapte mutarea care a fost efectuată.

```
(defrule afisare_mutare
        ?f <- (move ?n ?start ?end)
        =>
        (retract ?f)
        (assert (curent_move ?n ?start ?end) )
        (printout t  "Discul " ?n "   " (letter ?start) " -> " (letter ?end) crlf)
)

(defrule mutare_disc
        (declare (salience 5))
        ?f <- (curent_move ?n ?start ?end)
        ?f1 <- (stiva (numar ?start) (valori $?v1 ?n) )
        ?f2 <- (stiva (numar ?end) (valori $?v2) )
        =>
        (retract ?f)
        (modify ?f1 (valori $?v1))
        (modify ?f2 (valori $?v2 ?n))
)
```

În acest moment programul poate fi rulat pas cu pas **(run 1)** sau se poate introduce un **break-point** la fiecare lansare a regulii *mutare_disc* şi astfel se pot urmări în lista de fapte (sau cu watch facts) modificările care sunt efectuate asupra stivelor. Dacă se doreşte o reprezentare grafică a stivelor pentru a putea urmări mai uşor mutările discurilor, putem realiza o regulă care să se activeze după fiecare modificare a stivelor (adică după fiecare activare a regulii *mutare_disc*).

```
(defrule vizualizare_1
        (declare (salience 5))
        (stiva (numar 1) (valori $?v1) )
        (stiva (numar 2) (valori $?v2) )
        (stiva (numar 3) (valori $?v3) )
        =>
        (loop-for-count (?i 1 ?*N*)
                (bind ?level (- (+ ?*N* 1) ?i) )
                (bind ?l1 (nth$  ?level  $?v1) )
                (bind ?l2 (nth$  ?level  $?v2) )
                (bind ?l3 (nth$  ?level  $?v3) )
                (if (eq ?l1 nil)
                        then (format  t  "     ")
                        else (format  t  "%5d" ?l1) )
                (if (eq ?l2 nil)
                        then (format  t  "     ")
                        else (format  t  "%5d" ?l2) )
                (if (eq ?l3 nil)
                        then (format  t  "     ")
                        else (format  t  "%5d" ?l3) )
                (printout t crlf)
        )
        (printout t "_____" crlf)
)
```

Dacă se doreşte o în locul numărului asociat discului o reprezentare a formei acestora sugerându-se astfel mai bine dimensiunea discului. Putem folosi pentru aceasta un mic truc. Cracterul **Tab** este reprezentat în CLIPS ca şi toate celelalte caractere neprintabile printr-un mic dreptunghi. Un disc poate fi reprezentat prin mai multe astfel dreptunghiuri.

```
(defrule vizualizare_2
        (declare (salience 5))
        (stiva (numar 1) (valori $?v1) )
        (stiva (numar 2) (valori $?v2) )
        (stiva (numar 3) (valori $?v3) )
        =>
        (loop-for-count (?i 1 ?*N*)
                (bind ?level (- (+ ?*N* 1) ?i) )
                (bind ?l1 (nth$  ?level  $?v1) )
                (bind ?l2 (nth$  ?level  $?v2) )
                (bind ?l3 (nth$  ?level  $?v3) )
                (loop-for-count(?j 1 (+ ?*N* 2))
                        (if (eq ?l1 nil)
                         then (printout t " ")
                         else (if (>= ?l1 ?j)
                                 then (printout t "          ")     ;caracterul TAB
                                 else (printout t " ") ) ) )
```

```
        (loop-for-count(?j 1 (+ ?*N* 2))
                (if (eq ?l2 nil)
                 then (printout t " ")
                 else (if (>= ?l2 ?j)
                        then (printout t "        ")    ;caracterul TAB
                        else (printout t " ") ) ) )
        (loop-for-count(?j 1 (+ ?*N* 2))
                (if (eq ?l3 nil)
                 then (printout t " ")
                 else (if (>= ?l3 ?j)
                        then (printout t "        ")    ;caracterul TAB
                        else (printout t " ") ) ) )
                (printout t crlf)
        )
        (printout t "_____" crlf)
)
```

Se poate interoga eventual utilizatorul asupra modului de vizualizare pe care îl doreşte şi astfel regula *interogare_utilizator* va arăta după cum urmează. Pentru a se aprinde doar una din regulile de vizualizare este necesară existența unui fapt care să apară în LHS-ul unei reguli negat şi în antetul celeilalte normal. Ex: *(view graphic)* şi *(not(view graphic))*.

```
(defrule interogare_utilizator
        (declare (salience 10))
        =>
        (printout t "Doriti vizualizare grafica Y/N ? ")
        (bind ?flag (readline))
        (if (neq ?flag "y" "Y" "yes" "Yes" "YES")
                then else (assert (view graphic) ) )
        (printout t "Doriti un alt numar de discuri (N=3) Y/N ? ")
        (bind ?flag (readline))
        (if (neq ?flag "y" "Y" "yes" "Yes" "YES")
         then else (assert (schimba_numar_discuri) ) )
        )
```

În acest moment calculatorul este capabil să ofere soluția de rezolvare a problemei turnurilor din Hanoi. Dacă se doreşte transformarea acestui program într-un joc se pot introduce noi reguli care să permită jucătorului să-şi exerseze dexteritatea. Mutările în care este rezolvată problema pot fi contorizate şi dacă la final de joc vor fi egale cu $2^N - 1$, atunci jucătorul va fi felicitat şi va primi eventual un punctaj maxim. În momentul în care se vor intoduce de la tastatură mutările de efectuat, trebuiesc realizate verificări de genul: mutare peste un disc mai mic, stiva de start vidă, date introduse valide (numărul discului de mutat între 1 şi N, numărul sau numele stivelor de start şi de end valide). Se va realiza o interogare în regula *interogare_utilizator* dacă se doreşte sau nu o demonstrație. Un fapt *(demo)* va fi asociat tuturor regulilor de mai sus (*start_program*, *afisare_mutare*) şi negata acestuia *(not (demo))* tuturor regulilor în care joacă omul.

```
(defrule interogare_utilizator
        (declare (salience 10))
        =>
        (printout t “Doriti o demonstrație Y/N ? “)
        (bind ?flag (readline))
        (if (neq ?flag "y" "Y" "yes" "Yes" "YES")
                then else (assert (demo) ) )
        (printout t “Doriti vizualizare grafica Y/N ? “)
        (bind ?flag (readline))
        (if (neq ?flag "y" "Y" "yes" "Yes" "YES")
                then else (assert (view graphic) ) )
        (printout t "Doriti un alt numar de discuri (N=3) Y/N ? ")
        (bind ?flag (readline))
        (if (neq ?flag "y" "Y" "yes" "Yes" "YES")
                then else (assert (schimba_numar_discuri) ) )
)

(defrule introducere_mutare
        (not (demo))
        =>
        (printout t  “Introduceti discul (1-“ ?*N* “): ”)
        (bind ?disc (read))
        (while (or (not (integerp ?disc)) (< ?disc 1) (> ?disc ?*N*))
                (printout t  “Introduceti discul (1-“ ?*N* “): ”)
                (bind ?disc (read)) )
        (printout t  “Introduceti tija de start  (1 2 3): ”)
        (bind ?start (read))
        (while (neq ?start 1 2 3)
                (printout t  “Introduceti tija de start (1 2 3): ”)
                (bind ?start (read)) )
        (printout t  “Introduceti tija de end (1 2 3): ”)
        (bind ?end (read))
        (while (neq ?end 1 2 3)
                (printout t  “Introduceti tija se end (1 2 3): ”)
                (bind ?end (read)) )
        (assert (curent_move ?disc ?start ?end))
)

(defrule mutare_invalida_1
        (declare (salience 10))
        (not (demo))
        ?f <- (curent_move ?disc ?start ?end)
        (stiva (numar ?start) (valori $?v &:(= (length$ $?v) 0) ))
        =>
        (retract ?f )
        (refresh introducere_mutare)
        (printout  t  “Stiva de start este vida ...” crlf)
)
```

```
(defrule mutare_invalida_2
        (declare (salience 10))
        (not (demo))
        ?f <- (curent_move ?disc ?start ?end)
        (stiva (numar ?start) (valori $? ?last &:(neq ?last ?disc)))
        =>
        (retract ?f )
        (refresh introducere_mutare)
        (printout  t  "Discul " ?disc " nu este în varful stivei de start..." rlf)
)

(defrule mutare_invalida_3
        (declare (salience 10))
        (not (demo))
        ?f <- (curent_move ?disc ?start ?end)
        (stiva (numar ?start) (valori $? ?disc))
        (stiva (numar ?end) (valori  $? ?last &:(> ?disc ?last)))
        =>
        (retract ?f )
        (refresh introducere_mutare)
        (printout  t  "Nu se poate muta peste un disc mai mic ..." crlf)
)

(defglobal ?*move* = 1)

(defrule not_finish
        (declare (salience –5))
        (not (demo))
        (not (stiva (valori $?v &:(eq ?*N* (length$ $?v)) )))
        =>
        (bind ?*move* (+ ?*move* 1))
        (refresh introducere_mutare)
        (refresh not_finish)
)

(defrule finish
        (declare (salience –5))
        (not (demo))
        (stiva (valori $?v &:(eq ?*N* (length$ $?v)) ))
        =>
        (if (eq (float ?*move*) (- (** 2 ?*N*) 1))
         then (printout t "Felicitari ai reusit numarul minim de mutari !" t)
         else (printout t "Se poate şi mai bine. Mai incearca..." t) )
        (printout t "Vrei sa rulezi progrmul inca o data Y/N ? ")
        (if (neq (read) y Y yes Yes YES)
                then else (reset) (run) )
)
```

3.3. Simplificarea regulilor complexe

Limbajele bazate pe reguli permit exprimarea a numeroase probleme într-un mod foarte simplu şi elegant. De foarte multe ori şi mai ales când avem un număr mare de fapte şi când programatorul doreşte o rezolvare cât mai rapidă a problemei este necesară o regândire a algoritmului. In exemplul următor se va realiza o buclă cu ajutorul unei funcţii **while** (sau **loop-for-count**) şi se vor aserta un grup de numere.

```
(deffacts max-num
        (loop-max 100) )

(defrule loop-assert
        (loop-max  ?n)
        =>
        (bind ?i 1)                                     ; i=1
        (while (<= ?i ?n) do
                (assert (number ?i) )
                (bind ?i  (+ ?i 1) ) ) )                ; i=i+1
```

Cu ajutorul acestui grup de numere asertate se pot face o serie de teste, cum ar fi de exemplu găsirea celui mai mare număr dintre acestea. De exemplu:

```
(defrule largest-number
        (number ?n1)
        (not (number ?n2 &:(> ?n2 ?n1) ) )
        =>
        (printout  t  "Largest number is " ?n1 crlf)
)
```

Această regulă, deşi este foarte simplă nu este şi cea mai rapidă metodă de găsire a celui mai mare număr. Dacă N reprezintă numărul faptelor cu relaţia *number* atunci între primul şi al doilea pattern al regulii vom avea un număr egal cu N de pattern-uri care se încearcă a fi potrivite. Se vor efectua astfel nici mai mult nici mai puţin decât NxN comparaţii pentru a se găsi soluţia finală. Crescând valoarea lui *loop-max* la valori mai mari 200, 300, 400, etc. se poate observa cum timpul de rulare, proporţional cu pătratul lui N este din ce în ce mai mare.

Acest tip de comparaţie este ineficient deorece orice număr nou adăugat este comparat cu toate numerele existente, pentru a se stabili dacă acesta este sau nu cel mai mare şi nu cu valoarea maximă actuală. Un algoritm în care valoarea maximă este păstrată într-un fapt, care este modificat doar în cazul găsirii unui număr mai mare duce doar la un număr de N comparaţii şi nu de NxN ca în cazul precedent. In acest fel în cazul introducerii unui nou număr sunt eliminate comparaţiile care nu sunt necesare. Următoarele reguli ilustrează acest mod de găsire a celui mai mare număr.

```
(defrule try-number
        (number ?n)
        =>
        (assert (try-number ?n) ) )

(defrule largest-unknown
        ?f <- (try-number ?n)
        (not (largest ?) )                      ; daca faptul largest nu exista
        =>
        (retract ?f)                            ; atunci acesta esta asertat.
        (assert  (largest ?n) ) )

(defrule largest-smaller
        ?f1 <- (largest ?n1)                    ; daca s-a gasit o valoare mai
        ?f2 <- (try-number ?n2 &:(> ?n2 ?n1))   ; mare atunci maximul anterior
        =>                                      ; este sters si apoi asertat cu
        (retract ?f1 ?f2)                       ; valoarea nou gasita.
        (assert (largest ?n2) ) )


(defrule largest-bigger
        (largest ?n1)                           ; daca nu s-a gasit o valoare mai
        ?f <- (try-number ?n2 &:(<= ?n2 ?n1))   ; mare decat actualul maxim
        =>                                      ; atunci nu se face nici o
        (retract ?f) )                          ; modificare.

(defrule print-largest
        (declare (salience –1) )
        (largest ?max)
        =>
        (printout  t  "Largest number is "  ?max  crlf ) )
```

Prima regulă deşi la prima vedere pare a nu fi necesară, asigură comparația unui fapt o singură dată. După cum se ştie o regulă nu se poate aprinde în mod normal decât o singură dată pentru un anumit fapt, dacă se doreşte reaprinderea ei în acelaşi context fiind necesară fie ştergerea şi reasertarea faptului respectiv, fie reîmprospătarea regulii cu ajutorul comenzii **refresh**, sau (ca o măsură extremă) aplicarea comenzii **reset**, care duce un program la starea inițială prin modificarea listei de fapte şi a agendei.

Pentru un număr mic de numere prima variantă poate părea mai rapidă, deoarece în cea de a doua metodă se fac şi o serie de operații suplimentare, cum ar fi asertarea şi ştergerea faptului *try-number,* aprinderea unor reguli complementare, cum ar fi regula *try-number* . Motorul de inferență s-ar putea dovedi mai rapid, chiar dacă teoretic numărul de comparații realizat este mai mare şi asta pentru că la o simplă operație de asertare se fac o serie de verificări (legate de existența unui

fapt similar), se asociază faptului o adresă unică, se incrementează contorul listei de fapte, etc. De asemenea operația de aprindere a unei reguli este însoțită de verificarea patern-urilor din LHS-ul regulii, de scrierea în agendă a numelui regulii, execuția comenzilor din RHS şi apoi de ştergerea din agendă a regulii.

Exemplul următor deşi pare mai simplu, nu este şi funcțional datorită faptului că se intră într-o buclă infinită, de fiecare dată când maximul este modificat, în patern-ul regulii *find-max* la potrivire vom avea pentru faptul *max* un alt număr de fapt (un alt context) şi se va aprinde pentru acelaşi fapt *number*. A se rula pas cu pas (run 1).

```
(defrule init-max
        (number ?n)
        (not (max ?)
        =>
        (assert  (max ?n) ) )

(defrule find-max
        ?f <- (max ?max)                ;se intră într-o buclă infinită
        (number ?n &:( < ?max ?n) )
        =>
        (retract ?f)
        (assert (max ?n) ) )
```

De reținut este faptul că în general nu se doreşte alterarea bazei de cunoştințe şi că de cele multe ori nu este permisă ştergerea faptelor care au fost utilizate de o regulă, în eventualitatea în care acestea vor fi folosite ulterior şi de o altă regulă. Dacă nu avem această restricție putem elimina faptele care au fost comparate şi în acest fel nu se va mai intra într-o buclă infinită.

De asemenea se va vedea că utilizarea unor variabile globale poate conduce la evitarea unor astfel de bucle infinite, prin stocarea valorilor care sunt frecvent modificate în astfel de variabile. Introducerea într-un limbaj bazat pe reguli cum este CLIPS-ul a variabilelor globale, precum şi a funcțiilor procedurale vine în ajutorul celor familiarizați cu programarea în limbaje de nivel înalt (*C, Pascal*) , ducând şi la o scădere semnificativă a numărului de reguli, fapt demn de luat în considerație în sistemele cu sute sau mii de reguli.

3.4. Încărcarea şi salvarea faptelor

Viteza unui program poate fi mărită prin scăderea numărului de fapte din *fact-list*. O metodă pentru reducerea numărului de fapte este aceea de a încărca faptele în CLIPS doar în momentul în care acesta are nevoie de ele. De exemplu, un program care realizează un diagnostic pentru problemele unei maşini, ar putea mai întâi să ceară informații referitoare la tipul maşinii şi apoi să încarce datele

specifice acesteia. Funcţiile *load-facts* şi *save-facts* sunt oferite de CLIPS special pentru a se permite încărcarea faptelor dintr-un fişier sau salvarea acestora într-un fişier. Sintaxa acestor două funcţii este următoarea:

(**load-facts** <file-name>)

(**save-facts** <file-name>
[<save-scope> <deftemplate-nam,es>*])

,unde <save-scope> este definit ca: **visible** | **local**

Funcţia *load-facts* va încărca un grup de fapte stocat într-un fişier specificat de <*file-name*>. Faptele din fişier trebuie să fie în formatul standard al faptelor ordonate sau faptelor *deftemplate*. De exemplu, dacă fişierul "f.dat" conţine următoarele date:

(word calculator)
(lista 2 4 6 8 0)
(word program)

atunci comanda *(load-facts "f.dat")* va încărca toate datele conţinute în acest fişier.

Funcţia *save-facts* poate fi utilizată pentru a salva faptele din listă într-un fişier specificat de <file-name>. Faptele vor fi stocate în formatul cerut de funcţia *load-facts*. Dacă <save-scope> nu este specificat sau este specificat ca fiind *local*, doar faptele ce corespund construcţiilor deftemplate definite în modulul curent sunt salvate în fişier. Dacă <save-scope> este specificat ca fiind *visible*, atunci toate faptele ce corespund construcţiilor deftemplate, ce sunt vizibile în modulul curent vor fi salvate în fişier. Dacă <save-scope> este specificat şi de asemenea sunt specificate şi una sau mai multe construcţii deftemplate, în acest caz doar faptele ce corespund construcţiilor deftemplate specificate vor fi salvate. Oricum numele acelor construcţii trebuie să satisfacă specificaţia de *visible* sau *local*.

Ambele funcţii **load-facts** şi **save-facts** returnează TRUE, dacă fişierul cu fapte a fost deschis şi operaţiile de încărcare sau salvare au fost efectuate cu succes. În caz contrar este returnat simbolul FALSE. Este obligaţia programatorului să se asigure că în modulul curent construcţiile deftemplate corespunzătoare faptelor deftemplate dintr-un fişier sunt vizibile şi comanda *load-facts* poate fi executată. Folosirea acestor comenzi oferă un acces mult mai rapid la datele dintr-un fişier, decât în cazul folosirii funcţiilor *printout* şi *read*. În multe situaţii se recomandă prelucrarea datelor în formatul specific faptelor şi încărcarea rapidă a acestora.

Întrebări

1. Ce este agenda? Modificările efectuate în agenda sunt statice sau dinamice? În cazul în care se practică o programare modulară se va folosi o singură agendă?
2. Ce criteriu dintre prioritatea regulii şi strategia motorului de inferenţă este mai important în amplasarea unei reguli în agendă?

3. Care sunt cele şapte strategii ale motorului de inferență? Care este strategia implicită a CLIPS-ului?
4. Dacă avem două reguli cu aceiaşi prioritate care sunt criteriile ce vor stabili ordinea acestora în agendă?
5. Care sunt diferențele obținute în urma folosirii strategiei *depth* sau *breadth*? Dar între strategiile *simplicity* şi *complexity*?
6. Care este complexitatea regulii următoare? Se reaminteşte faptul că folosirea funcțiilor booleene and, or, not nu sporesc complexitatea unei reguli şi că funcțiile apelate din interiorul unei alte funcții nu sunt luate în calcul.

```
(defrule find_word_1
        (litera ?n1 ?l1)
        (litera ?n2 ?l2 &:(or(neq ?l2 ?l1) (<> ?n2 ?n1)) )
        (word ?l1 ?l2)
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2) t) )
```

7. Complexitatea regulii următoare este mai mică sau mai mare decât cea a regulii anterioare?

```
(defrule find_word_2
        ?f1<- (litera ?n1 ?l1)
        ?f2<- (litera ?n2 ?l2)
        (test (neq ?f1 ?f2))
        (or (word ?l1 ?l2)
            (word ?l2 ?l1))
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2) t) )
```

8. În calcularea complexității sunt studiate şi funcțiile din LHS şi funcțiile din RHS? De ce?
9. Strategia *random* asociază fiecărei reguli un număr aleator. Schimbarea strategiei şi revenirea apoi la strategia random va schimba ordinea activărilor în agenda?
10. Care din următoarele variante are cele mai puține potriviri parțiale şi deci va fi executată mai rapid? Argumentați răspunsul dat.

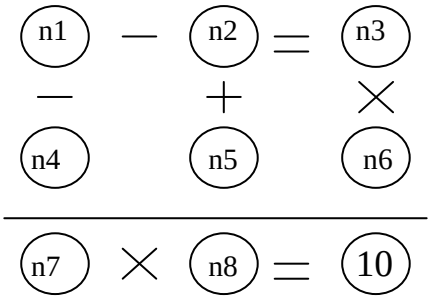
```
(defrule find_number_1                  (defrule find_number_2
    (lista ?n1 ?n2)                         ?f1 <- (number ?n1)
    ?f1 <- (number ?n1)                     ?f2 <- (number ?n2 &:(neq ?n2 ?n1))
    ?f2 <- (number ?n2 &:(neq ?f2 ?f1))     (lista ?n1 ?n2)
    => .....                                => .....
)                                       )
```

11. Pentru ce sunt folosite funcțiile load-facts şi save-facts? Ce semnifică opțiunile de local şi visible în instrucțiunea save-facts?
12. Care este efectul obținut după lansarea în execuție a funcției următoare?

(save-facts "f.dat" local person value)

Probleme rezolavate

1. **Problema aritmetică** – Fiind date 8 cercuri legate între ele prin operații aritmetice, ce cifre trebuiesc puse în acestea pentru a se obține ca rezultat 10.



Soluție:
3 – 1 = 2
– + x
1 4 5
2 x 5 =10

Pentru început se încearcă scrierea unei reguli care să cuprindă în pattern-urile sale toate cele opt variabile ?n1,...?n8 şi condițiile legate de operațiile aritmetice ce se efectuează între ele. Sunt asertate cu ajutorul unei construcții deffacts toate cele 9 cifre arabe.

Regula va conține opt patern-uri şi cinci teste, fiecare dintre ele verificând o operație aritmetică dintre cifrele înscrise în cercuri (variabilele ?n1,..?n8). În această variantă a programului CLIPS-ul nu furnizează nici o eroare, dar după lansarea programului în execuție timpul în care va fi furnizată o soluție este foarte mare (peste 20 minute). Se pot vedea numărul uriaş de posibile potriviri, cu ajutorul funcției *(matches calculare_soluție)*.

Varianta 1.

```
(deffacts date-initiale
        (numar 0)   (numar 1)   (numar 2)   (numar 3)   (numar 4)
        (numar 5)   (numar 6)   (numar 7)   (numar 8)   (numar 9))

(defrule calculare_solutie
        (numar ?n1) (numar ?n2) (numar ?n3)
        (test (= ?n3 (- ?n1 ?n2)))
        (numar ?n4) (numar ?n5) (numar ?n6)
        (numar ?n7) (numar ?n8)
        (test (and (= ?n7 (- ?n1 ?n4))
                   (= ?n8 (+ ?n2 ?n5))
                   (= 10 (* ?n3 ?n6) (* ?n7 ?n8)) ))
        =>
        (format t "%n%2d - %2d = %2d %n" ?n1 ?n2 ?n3) (printout t " -   +   x" t)
        (format t "%2d   %2d   %2d %n" ?n4 ?n5 ?n6) (printout t "_____" t)
        (format t "%2d x %2d = %2d %n" ?n7 ?n8 10)
        (printout t "Apasa o tasta...") (readline)
)
```

Studiind datele problemei cu atenție se observă faptul că variabilele ?n3, ?n7 şi ?n8, sunt obținute în urma operațiilor de scădere sau adunare dintre alte variabile. Din acest motiv ele pot fi omise, în acest fel simplificându-se numărul de patter-uri şi prin urmare şi numărul de potriviri posibile ale faptelor cu patter-urile regulilor. În plus se mai observă faptul că aceste operații de adunare şi de scădere au ca rezultat o cifră şi deci nu pot depăşi marginile intervalului [0-9].

În urma realizării acestor îmbunătățiri, timpul de găsire a soluțiilor a scăzut foarte mult la (5-10 secunde). Putem afirma că acest timp este acceptabil în comparație cu timpul obținut anterior.

Varianta 2.

```
(deffacts date-initiale
        (numar 0)   (numar 1)   (numar 2)   (numar 3)   (numar 4)
        (numar 5)   (numar 6)   (numar 7)   (numar 8)   (numar 9))

(defrule calculare_solutie
        (numar ?n1) (numar ?n2) (numar ?n4) (numar ?n5) (numar ?n6)
        (test (and (< (+ ?n2 ?n5) 10) (>= (- ?n1 ?n4) 0) ))
        (test (= 10 (* (- ?n1 ?n2) ?n6) (* (- ?n1 ?n4) (+ ?n2 ?n5))) )
        =>
        (format t "%n%2d - %2d = %2d %n" ?n1 ?n2 (- ?n1 ?n2))
        (printout t " -    +    x" t)
        (format t "%2d   %2d   %2d %n" ?n4 ?n5 ?n6)
        (printout t "_____________" t)
        (format t "%2d x %2d = %2d %n" (- ?n1 ?n4) (+ ?n2 ?n5) 10)
        (printout t "Apasa o tasta...") (readline)
)
```

Intrând şi mai adânc în datele problemei se observă faptul că ?n3, ?n6, ?n7, ?n8 sunt divizori ai lui 10, dar nu pot lua decât valorile de 2 şi 5. Deoarece suma dintre valorile lui ?n2 şi valorile lui ?n5 este maxim 5 se obține pentru aceste variabile intervalul [0-5]. Din ?n1-?n2 ={2,5}, rezultă pentru variabila ?n1 intervalul [2-7]. Se obține automat şi pentru variabila ?n4 intervalul [0-5]. Mai mult se poate merge până la înglobarea într-un element condițional *or* a celor patru cazuri când (*?n3,?n6,?n7,?n8*) iau valorile (2,5,2,5); (2,5,5,2); (5,2,2,5) şi (5,2,5,2). Pentru fiecare dintre aceste cazuri se pot determina intervale de existență mai exacte pentru variabilele rămase. Se observă faptul că variabilele ?n2, ?n4 şi ?n5 pot fi calculate în funcție de variabila ?n1, ceea ce simplifică şi mai mult problema.

Varianta 3.

```
(defrule generare_numere
        =>
        (loop-for-count (?i 0 9)
                (assert (numar ?i)) )
)
```

```
(defrule calculare_solutie
   (or (and (numar ?n1&:(>= ?n1 2)&:(<= ?n1 7)) (numar ?n2&=(- ?n1 2))
            (numar ?n4&=(- ?n1 2)) (numar ?n5&=(- 7 ?n1)) (numar ?n6&5))
       (and (numar ?n1&:(>= ?n1 5)) (numar ?n2&=(- ?n1 5))
            (numar ?n4&=(- ?n1 2)) (numar ?n5&=(- 10 ?n1)) (numar ?n6&2))
       (and (numar ?n1&:(>= ?n1 2)&:(<= ?n1 4)) (numar ?n2&=(- ?n1 2))
            (numar ?n4&=(- ?n1 5)) (numar ?n5&=(- 4 ?n1)) (numar ?n6&5))
       (and (numar ?n1&:(>= ?n1 5)&:(<= ?n1 7)) (numar ?n2&=(- ?n1 5))
            (numar ?n4&=(- ?n1 5)) (numar ?n5&=(- 7 ?n1)) (numar ?n6&2)) )
   =>
   (format t "%n%2d - %2d = %2d %n" ?n1 ?n2 (- ?n1 ?n2)) (printout t " -   +   x" t)
   (format t "%2d   %2d   %2d %n" ?n4 ?n5 ?n6) (printout t "___________" t)
   (format t "%2d x %2d = %2d %n" (- ?n1 ?n4) (+ ?n2 ?n5) 10)
   (printout t "Apasa o tasta...") (readline)
)
```

Test următor devine inutil, deoarece au fost aleşi doar divizorii lui 10 şi condițiile sunt îndeplinite din start.

(test (= 10 ((- ?n1 ?n2) ?n6) (* (- ?n1 ?n4) (+ ?n2 ?n5))))*

Timpul de răspuns pentru această ultimă variantă se poate afirma că este aproape instantaneu. Apropiindu-ne foarte mult de performanţele ultimei variante putem scrie şi o variantă mai clară:

```
(defrule calculare_solutie
        (numar ?n1)
        (numar ?n3 &2|5)
        (numar ?n6 &=(- 7 ?n3))
        (numar ?n2 &=(- ?n1 ?n3))
        (numar ?n7 &2|5)
        (numar ?n8 &=(- 7 ?n7))
        (numar ?n4 &=(- ?n1 ?n7))
        (numar ?n5 &=(- ?n8 ?n2))
        =>
        (format t "%n%2d - %2d = %2d %n" ?n1 ?n2 ?n3) (printout t " -   +   x" t)
        (format t "%2d   %2d   %2d %n" ?n4 ?n5 ?n6) (printout t "___________" t)
        (format t "%2d x %2d = %2d %n" ?n7 ?n8 10)
        (printout t "Apasa o tasta...") (readline)
)
```

S-au scris mai multe variante ale aceluiaşi program pentru a se observa faptul că de cele mai multe ori se pot aduce îmbunătăţiri unui program. În urma acestor optimizări programul poate câştiga un plus de viteză, şi acest lucru este foarte util în programele care au număr mare de reguli. Se pot urmări numărul de potriviri pentru toate aceste variante folosind *(matches calculare_soluție)* şi se pot observa câteva procedee simple de optimizare a codului unui program în CLIPS.

2. <u>Căutare într-o bază de cuvinte</u> – Să presupunem faptul că am dori să realizăm un joc de Scrable şi că avem nevoie de un program care pentru literele existente în mână şi pentru literele de pe tabla de joc, să furnizeze toate soluțiile găsite. După găsirea acestor soluții se poate practic realiza o selectarea a soluțiilor în funcție de punctaj sau de alte criterii.

Programul dispune de un dicționar cu zeci de mii de cuvinte, iar căutarea soluțiilor trebuie să fie rapidă şi eficientă. În momentul în care se pune problema lucrului cu fişiere atât de mari, dat fiind faptul că instrucțiunile de scriere şi citire dintr-un fişier, *read* şi *printout* nu sunt suficient de rapide pentru o astfel de aplicație, se recomandă utilizarea funcțiilor *load-facts* şi *save-facts* pentru încărcarea (sau salvarea) faptelor direct dintr-un fişier. În prealabil trebuie însă realizată o citirea datelor din dicționar cu funcția *read* şi asertarea unui fapt pentru fiecare cuvânt citit. După ce au fost citite toate datele din fişier se pot salva toate faptele din lista de fapte într-un fişier "dict.sav", care apoi poate fi folosit ori de câte ori se doreşte căutarea unui cuvânt în dicționar.

Când baza de cunoştințe este aşa de mare, orice regulă prost scrisă poate consuma un timp foarte mare sau chiar reseta programul CLIPS în momentul în care are loc o depăşire a capacității stivei. Din considerente practice în urma mai multor încercări nereuşite, s-a optat pentru despărțirea fiecărui cuvânt pe litere. În acest fel se poate verifica mai uşor existența sau absența unei litere într-un cuvânt. Regula următoare realizează citirea dicționarului şi asertează pentru fiecare cuvânt citit faptul *word* a cărui valoare de tip multicâmp conține toate literele acelui cuvânt.

```
(defrule deschidere_fisier
        (declare (salience 10))
        =>
        (printout t "Nume fisier : ") (bind ?name (readline))
        (open ?name file)
        (while (neq (bind ?word (read file)) EOF)
                (bind $?w (create$ ))
                (loop-for-count (?i 1 (length ?word))
                        (bind $?w (create$ $?w (sym-cat (sub-string ?i ?i ?word)))) )
                (assert (word $?w))
        )
   (close file)
)
```

Pentru segmentarea unui cuvânt se foloseşte într-o buclă de lungime egală cu numărul literelor *(bind $?w (create$ $?w (sym-cat (sub-string ?i ?i ?word))))*. Ca şi în cazul problemei anterioare nu se va încerca un algoritm procedural. După cum se va putea observa modul în care vor fi realizate regulile *find_word* este foarte facil. Practic scrierea în pseudocod a unei astfel de funcții s-ar putea realiza în felul următor:

Dacă litera $?l_1$, litera $?l_2$ diferită de litera $?l_1$,...,litera $?l_N$ diferită de toate celellate litere $?l_1$,$?l_2$,...,$?l_{N-1}$ sunt incluse în cuvântul ?w
Atunci afişează cuvântul găsit.

```
(deffacts date_initiale
        (litere_alfabet a b c d e f g h i j k l m n o p q r s t u v x z w y) )

(defrule deschidere_fisier
        (declare (salience 10))
        =>
        (load-facts "dict.sav") )

(defrule introducere_litere_cautare
        (litere_alfabet $?val)
        =>
        (printout t "Introduceti literele (".“ pt sfarsit)" t)
        (bind ?n 0)
        (while (neq (bind ?lit (read)) .)
                (if (member$ ?lit $?val)
                 then (assert (litera (bind ?n (+ ?n 1)) ?lit)) )
        )
)
(defrule find_word_2
        (litera ?n1 ?l1)
        (litera ?n2&~?n1 ?l2)
        (word ?l1 ?l2)
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2) t)
)
(defrule find_word_3
        (litera ?n1 ?l1) (litera ?n2&~?n1 ?l2) (litera ?n3&~?n2&~?n1 ?l3)
        (word ?l1 ?l2 ?l3)
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2 ?l3) t)
)
.....
(defrule find_word_6
   (litera ?n1 ?l1) (litera ?n2&~?n1 ?l2) (litera ?n3&~?n2&~?n1 ?l3)
   (litera ?n4&~?n3&~?n2&~?n1 ?l4) (litera ?n5&~?n4&~?n3&~?n2&~?n1 ?l5)
   (litera ?n6&~?n5&~?n4&~?n3&~?n2&~?n1 ?l6)
   (word ?l1 ?l2 ?l3 ?l4 ?l5 ?l6)
   =>
   (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2 ?l3 ?l4 ?l5 ?l6) t)
)
```

După cum se poate observa după ce au fost încărcate faptele *word* din fișierul “dict.dat” se introduc literele pentru care se va face căutarea în dicționar. Se pot introduce litere până la introducerea caracterului ‘.’, moment în care se iese

din buclă. Odată cu asertarea fiecărei litere CLIPS-ul încercă să găsească acele cuvinte ce conțin literele deja existente.

În jocul de scrable un jucător poate forma un cuvânt care să conțină mai multe litere de acelaşi fel (de exemplu cuvântul 'perene' în care litera e apare de trei ori). Deşi nu există restricția de a se forma cuvinte în care fiecare literă apare o singură dată, totuşi există o limită dată de numărul maxim de litere pe care jucătorul le poate folosi la un moment dat. Din acest motiv a fost asociat în mod unic un număr fiecărei litere, fapt ce duce la folosirea fiecărei litere o singură dată. Dacă de exemplu avem doi de 'A', vor fi folosiți dacă este cazul, fiecare pe rând fără a încurca unul cu celălalt. Existența acestui număr este transparentă pentru cuvântul a cărui litere sunt comparate.

Aşa se explică faptul că în pseudocodul regulilor *find_word* s-a menționat faptul că fiecare literă trebuie să fie diferită de celelalte (nu prin valoarea ei, ci prin numărul ei ca şi piesele de joc).

După rularea programului se observă faptul ca pentru un număr de litere mai mic decât 5, programul funcționează acceptabil. Odată cu mărirea numărului de piese creşte şi timpul de căutare.

Îmbunătățirea ce ar putea fi adusă funcției în acest moment constă în faptul că în cazul în care avem două litere diferite este inutilă compararea numerele pieselor. Aceasta ar trebui realizată doar în cazul în care avem două litere identice şi vrem să fim siguri fiecare literă va fi folosită doar o singură data. Modificările care se fac funcțiilor *find-word* constau în verificare valorilor literelor înaintea numerelor acestora.

```
(defrule find_word_2
        (litera ?n1 ?l1)
        (litera ?n2 ?l2&:(or(neq ?l2 ?l1) (<> ?n2 ?n1)) )
        (word ?l1 ?l2)
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2) t)
)
.....

(defrule find_word_5
        (litera ?n1 ?l1)
        (litera ?n2 ?l2&:(or(neq ?l2 ?l1) (<> ?n1 ?n2)) )
        (litera ?n3 ?l3&:(or(neq ?l3 ?l2 ?l1) (<> ?n3 ?n2 ?n1)) )
        (litera ?n4 ?l4&:(or(neq ?l4 ?l3 ?l2 ?l1) (<> ?n4 ?n3 ?n2 ?n1)) )
        (litera ?n5 ?l5&:(or(neq ?l5 ?l4 ?l3 ?l2 ?l1) (<> ?n5 ?n4 ?n3 ?n2 ?n1)) )
        (word ?l1 ?l2 ?l3 ?l4 ?l5)
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2 ?l3 ?l4 ?l5) t)
)
```

Şi în acest caz nu au fost scrise toate funcțiile *find-word,* deoarece ele pot fi uşor obținute urmărind un singur model. În momentul în care se rulează programul astfel modificat se simte o creştere a vitezei de căutare.

Înainte de a face următoarea optimizare a regulilor se va aminti faptul că ordinea pattern-urilor în LHS-ul unei reguli poate influența viteza şi eficiența folosirii memoriei. Prin aducerea ultimului pattern, corespunzător faptului *word,* pe prima poziție în LHS, scade foarte mult numărul potrivirilor parțiale şi prin urmare viteza de răspuns creşte foarte mult. Găsirea aproape instantanee a cuvintelor chiar şi pentru un număr mare de litere la intrare, oferă certitudinea că un program bine realizat în CLIPS poate fi foarte performant. Un program realizat în CLIPS ar furniza soluțiile găsite pentru valorile primite la intrare, unui program realizat în C de exemplu. La ora actuală există în lume tendințe de înglobare a unor astfel de module non-procedurale. De exemplu în Java, pot fi folosite de alte aplicații în mod transparent, programele realizate în JESS (compatibil 100% cu limbajul CLIPS).

```
(defrule find_word_2
        (word ?l1 ?l2)
        (litera ?n1 ?l1)
        (litera ?n2 ?l2&:(or(neq ?l2 ?l1) (<> ?n2 ?n1)) )
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2) t)
)

(defrule find_word_3
        (word ?l1 ?l2 ?l3)
        (litera ?n1 ?l1)
        (litera ?n2 ?l2&:(or(neq ?l2 ?l1) (<> ?n1 ?n2)) )
        (litera ?n3 ?l3&:(or(neq ?l3 ?l2 ?l1) (<> ?n3 ?n2 ?n1)) )
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2 ?l3) t)
)

(defrule find_word_7
        (word ?l1 ?l2 ?l3 ?l4 ?l5 ?l6 ?l7)
        (litera ?n1 ?l1)
        (litera ?n2 ?l2&:(or(neq ?l2 ?l1) (<> ?n1 ?n2)) )
        (litera ?n3 ?l3&:(or(neq ?l3 ?l2 ?l1) (<> ?n3 ?n2 ?n1)) )
        (litera ?n4 ?l4&:(or(neq ?l4 ?l3 ?l2 ?l1) (<> ?n4 ?n3 ?n2 ?n1)) )
        .....
        =>
        (printout t "Gasit cuvant: " (sym-cat ?l1 ?l2 ?l3 ?l4 ?l5 ?l6 ?l7) t)
)

(defrule stergere_litere
        ?f <- (litera ? ?)
        =>
        (retract ?f))
```

```
(defrule reintroducere_litere
        (declare (salience -10))
        ?f <- (reluare)
        (not (litera ? ?))
        =>
        (retract ?f)
        (printout t "Doriti reluare ? ") (bind ?r (read))
        (if (neq ?r  Y y Yes YES yes D d Da DA da)
         then (return)
         else (refresh introducere_litere)) )
```

Dacă se doreşte efectuarea unei noi căutări şi în mod normal introducerea unor litere noi atunci se pot adăuga şi ultimele doua funcţii *sterge_litere* şi *reintroducere_litere*.

Dacă toate variantele de mai sus aveau proprietatea că puteau fi rulate, diferenţele între acestea constând în timpii de răspuns, vom oferi şi o variantă care deşi este corectă din punct de vedere sintactic şi la prima vedere ar fi cea mai potrivită, blochează sistemul la lansarea în execuţie.

În această variantă cuvintele din dicţionar nu mai sunt despărţite pe litere, şi după cum poate fi observat cu ajutorul comenzii *(matches <rule-name>)*, pentru un dicţionar nu un număr mic de cuvinte numărul potrivirilor este foarte mare.

```
(defrule open_dictionary
        =>
        (open "C:\\CLIPS\\roman.dat" ro)
        (while (neq (bind ?word  (read ro)) EOF)
                (assert  (word ?word)) )
        (close ro)
)
```


```
(defrule find_word
        (word ?w)
        (not (exists (my $? ?l&:(eq (str-index ?l ?w) FALSE)) $?) ))
        =>
        (assert (solution ?w))
)
```

Deşi programul rulează pentru un număr mic de cuvinte, în momentul în care dimensiunea dicţionarului creşte programul se blochează. Regula *find_word* caută cuvintele care conţin toate literele jucătorului (se încearcă găsirea cuvintelor cât mai mari). Odată asertat un fapt ordonat de genul *(my a e l m p)* CLIPS-ul va cauta toate faptele ce se potrivesc cu pattern-urile regulii. Se observă faptul că nu este avantajoasă folosirea elementelor condiţionale *not* sau *exists* în cazurile în care baza de cunoştinţe este foarte mare.