CAP 1. FUNCŢIILE PROCEDURALE

După cum se ştie în CLIPS se încearcă abordarea problemelor într-o manieră logică, non-procedurală, mai apropiată de modul de raţionament al omului. Deoarece majoritatea programatorilor ce nu sunt familiarizaţi cu un astfel de tip de programare tind să utilizeze funcţiile procedurale în exces, se recomandă folosirea acestora în interiorul unui program doar în anumite cazuri.

De obicei în funcţie de datele şi cerinţele problemei, programatorul decide mediul de programare în care se pretează a fi rezolvată cel mai bine problema respectivă. În CLIPS în anumite circumstanţe folosirea funcţiilor procedurale devine foarte avantajoasă (de exemplu în condiţiile în care prin folosirea unei astfel de funcţii se pot elimina două sau chiar mai multe reguli).

Deşi funcţiile următoare permit realizarea unei programări procedurale ca în orice alt limbaj de programare (C, Pascal), reamintim faptul că se recomandă folosirea lor doar în cazurile când acestea sunt strict necesare: **if**, **loop-for-count**, **while**, **switch**, **break**, **progn**, **progn$**, **return**.

1.1. Funcţia IF

Atunci când în RHS-ul unei reguli se doreşte efectuarea unui test sau verificarea unei valori se foloseşte funcţia procedurală **if**. Aceasta realizează evaluarea unei expresii şi în funcţie de rezultatul acestei evaluări execută o serie de acţiuni. Sintaxa generală a acestei funcţii este:

```
( if  <predicate-expression>
  then  <expression>+
  [ else  <expression>+ ] )
```

unde <predicate-expression> este fie o funcţie predicativă, fie o variabilă, iar <expression>+ care urmează după cuvintele cheie *then* şi *else* reprezintă una sau mai multe funcţii (expresii) ce vor fi executate în funcţie de valoarea returnată în urma evaluării lui <predicate-expression>. De reţinut faptul că *else* este opţional.

Dacă condiţia instrucţiunii *if* este evaluată la orice *symbol* diferit de simbolul FALSE, atunci acţiunile imediat următoare clauzei *then* sunt executate. Dacă condiţia este evaluată la simbolul FALSE atunci se execută acţiunile clauzei *else*. Dacă nu avem inclusă o clauză *else*, atunci în cazul unei condiţii false nu este executată nici o acţiune. Odată completată execuţia lui *if* se trece la acţiunea următoare din RHS.

De reţinut faptul că în CLIPS nu există o echivalenţă între simbolul FALSE şi valoarea 0, ca în alte limbaje de programare. Din acest motiv în cazul în care <predicate-expression> este exprimat printr-o variabilă care este de tip integer,

float, string, sau eventual un symbol ce va fi întotdeauna diferit de FALSE nu vor fi executate niciodată acțiunile din clauza *else*, devenind inutilă folosirea funcției *if* (expresia sa va fi întotdeauna evaluată ca fiind diferită de simbolul FALSE şi deci va fi întotdeauna adevărată).

```
(defrule functia_if_afiseaza_intotdeauna_TRUE
        =>
        (bind ?val (readline))          ; ?val este un string şi nu un symbol
        (if  ?val  then (printout  t TRUE)  else (printout  t FALSE))  )

CLIPS> (if  0  then (printout  t TRUE)  else (printout  t FALSE))
TRUE
CLIPS> (if  (read)  then (printout  t "Adevarat..." t)  else (printout  t "Fals..." t))
false                                   ;se introduce un simbol de la tastatură
Adevarat…
CLIPS> (if  (read)  then (printout  t "Adevarat…" t)  else (printout  t "Fals…" t))
FALSE                                   ;CLIPS-ul este case-sensitive
Fals…
```

Se observă faptul că atunci când partea condițională din cadrul funcției *if* este specificată printr-o valoare explicită sau printr-o variabilă de tip diferit de simbol, folosirea funcției devine inutilă, fiind cunoscută dinainte clauza ce va fi executată. Din exemplele de mai sus se observă şi faptul că simbolul *FALSE* este diferit de simbolul *false* (case-sensitive) sau şirul de caractere *"FALSE"*. În exemplul următor va fi folosită pentru condiția funcției *if* o funcție predicativă.

```
(defrule este_cifra_hexazecimala_?
        (litera ?c)
        =>
        (if (neq ?c 0 1 2 3 4 5 6 7 8 9 A B C D E F)
         then   (printout  t  "Litera " ?c " NU este o cifra hexazecimala !" t)
         else   (printout  t  "Litera " ?c " este o cifra hexazecimala." t) ))
```

Funcția predicativă **neq** ('not equal') compară primul argument cu toți ceilalți şi returnează TRUE dacă acesta nu este egal cu nici unul din celelalte argumente. Nu s-a folosit funcția <> ('diferit de') deoarece aceasta lucrează numai cu numere şi nu acceptă parametri de tip string sau simbol. Folosirea funcției *if* în acest caz este recomandată, deoarece în lipsa acesteia ar fi trebuit implementate două reguli, care să se aprindă fiecare pentru una din cele două clauze.

```
(defrule nu_este_cifra_hexazecimala
        (litera ?c &: (neq ?c 0 1 2 3 4 5 6 7 8 9 A B C D E F))
        =>
        (printout  t  "Litera " ?c " NU este o cifra hexazecimala !" t) )

(defrule este_cifra_hexazecimala
        (litera ?c & 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F))
        =>
        (printout  t  "Litera " ?c " este o cifra hexazecimala !" t) )
```

Şi în exemplul următor în care se realizează o interogare privind restartarea unui program, utilizarea funcţia *if* este avantajoasă, deoarece pot fi compactate două reguli într-una singură. Dacă ar fi existat şi o clauză *else*,. atunci ar fi fost compactate un număr de trei reguli.

```
(defrule start_again
    (declare (salience –10))
    =>
    (printout  t  "Start again?")
    (bind ?reply (read))
    (if  (or (eq ?reply y)(eq ?reply yes))
     then   (reset)   (run) ))
```

```
(defrule start_again
    (declare (salience –10))
    =>
    (printout  t  "Start again?")
    (assert (reply (read))) )
(defrule restart
    (reply y | yes) => (reset) (run) )
```

1.2. Funcţia WHILE

Funcţia **while** poate fi folosită în RHS-ul unei reguli pentru a realiza o buclă din care se iese doar atunci când este îndeplinită o anumită condiţie (de exemplu introducerea corectă de la tastatură a unei date de intrare). Sintaxa generală a acestei funcţii este următoarea:

(**while** <predicate-expression> [do]
<expression>*)

unde <predicate-expression> este o funcţie predicativă sau o variabilă, iar <expression>* care urmează după cuvântul cheie opţional *do* reprezintă zero sau mai multe acţiuni ce vor fi evaluate în funcţie de valoarea returnată în urma evaluării lui <predicate-expression>.

Partea din *while* reprezentată de condiţia <predicate-expression> este evaluată înaintea execuţiei acţiunilor din corpul buclei. Dacă condiţia este evaluată la orice simbol mai puţin simbolul FALSE, atunci toate expresiile din buclă vor fi executate. Dacă în urma evaluării se obţine simbolul FALSE atunci se trece la instrucţiunea imediat următoare din RHS-ul regulii. Condiţia din funcţia *while* este verificată de fiecare dată când se execută corpul buclei pentru a se stabili dacă acţiunile din buclă vor mai fi executate încă o dată.

Trebuie acordată o atenţie deosebită cazului în care se intră în buclă infinită (condiţia de repetare a buclei este întotdeauna diferită de simbolul FALSE). În exemplul următor funcţia *readline* returnează un string şi acesta nu poate fi comparat cu un symbol (simbolul n este diferit de şirul "n").

```
CLIPS> (while (neq (readline) n N)
            (printout  t  "Apasa tasta 'N'…" t) )          ;bucla infinita

CLIPS> (while (neq (read) "n" "N")
            (printout  t  "Apasa tasta 'N'…" t) )
```

Dacă se înlocuiesc funcția de intrare cu *read* şi parametrii funcției predicative *neq* cu şirurile "n" şi respectiv "N", deşi suntem într-un caz asemănător (se compară un număr sau un simbol cu un şir), nu se mai intră în bucla infinită. Dacă se introduc de la tastatură şi ghilimelele CLIPS-ul va interpreta datele de intrare ca fiind un şir de caractere şi se poate ieşi din buclă..

Funcția *while* poate fi utilizată împreună cu funcția *if* pentru a verifica datele introduse se utilizator de la tastatură, după cum se poate observa din exemplul următor. Atunci când datele de intrare invalide pot conduce la erori în execuția programului este foarte indicată folosirea unei astfel de bucle.

```
(defrule start-again
        (declare (salience –10))
        =>
        (printout t "Doresti sa începi jocul din nou (da / nu) ?")
        (bind ?raspuns (read))
        (while (and (neq ?raspuns da) (neq ?raspuns nu)) do
                (printout t "Introduceti (da / nu)")
                (bind ?raspuns (read)) )
        (if (eq ?raspuns da)
         then (reset) (run)
         else (assert (phase stop)) ))
```

Un alt exemplu foarte bun de utilizare a unei bucle *while* este cel în care se doreşte executarea unei/unor acțiuni un anumit timp (de exemplu un minut). Regula următoare este echivalentă cu o instrucțiune *sleep, delay, pause* din alte limbaje şi poate fi utilizată de exemplu pentru o pauză de vizualizare a unor rezultate.

```
(defrule asteapta_60_secunde
    =>
    (bind ?t (time))
    (while (< (- (time) ?t) 60) do
            (printout t "astept..." t))
    (printout t "A trecut un minut.") )
```

```
(defrule pause
    ?f <- (pause ?N)
    =>
    (retract ?f)
    (bind ?t (time))
    (while (< (- (time) ?t) ?N) do ) )
```

Dacă se doreşte implementarea unei bucle de tip *while* fără a se folosi funcția procedurală se pot folosi fapte de control, care să specifice dacă condiția de repetare a buclei mai este sau nu adevărată. În lipsa faptelor de control pot fi folosite prioritățile.

```
(defrule verificare_condiție_bucla
    (raspuns Y | y | Yes | yes)
    =>
    (assert (bucla TRUE)) )

(defrule actiuni_bucla_while
    ?f <- (bucla TRUE)
    =>
    (retract ?f) .....
    (printout t "Doriti reluare (Y/N)? ")
    (assert (raspuns (read))) )
```

```
(defrule verificare_condiție_bucla
    (declare (salience 10))
    (raspuns Y | y | Yes | yes)
    =>
    (refresh actiuni_bucla) )

(defrule actiuni_bucla_while
    =>
    .....
    (printout t "Reluare (Y/N)? ")
    (assert (raspuns (read))) )
```

1.3. Funcţia LOOP-FOR-COUNT

Pentru realizarea unei bucle de tip *for* avem definită în CLIPS funcţia procedurală **loop-for-count**. O astfel de buclă oferă utilizatorului posibilitatea executării instrucţiunilor din corpul unei bucle de un anumit număr de ori, acesta număr fiind păstrat într-o variabilă de index. Sintaxa instrucţiunii este următoarea:

(**loop-for-count** <range> [do] <action>*)

<range> ::= <end-integer> | <loop-variable> [<begin-integer> <end-integer>]

După cum se vede în <range> se poate specifica doar o valoare finală sau se poate specifica o variabilă şi opţional un interval (o valoare minimă şi o valoare maximă între care variabila poate lua valori). Odată cu execuţia corpului buclei este incrementată şi variabila index, astfel încât în momentul în care valoarea acesteia este mai mare decât valoarea maximă permisă, bucla nu se mai execută. Dacă nu este specificată o valoare de start pentru variabila index, aceasta are asociată automat valoarea 1. Pentru regulile următoare instrucţiunile din buclă vor fi executate de un număr de cinci ori.

```
(defrule bucla-for-1
        =>
        (loop-for-count 5
                (printout t "mesaj…"  )
)       )
```

```
(defrule bucla-for-2
     =>
     (loop-for-count (?i 1 5)
          (printout t "mesaj " ?i crlf)
)       )
```

```
(defrule bucla-for-3
      =>
       (loop-for-count (?i 5)
            (printout t "mesaj " ?i t)
)        )
```

În exemplul următor se vor introduce elementele unei matrice A[NxM], cu ajutorul a două bucle *for* imbricate. Se vor folosi două variabile ?i şi ?j fiecare luând valori în intervalele [1..N] şi respectiv [1..M], variabila ?i păstrând valoarea liniei curente, iar variabila ?j a coloanei curente. Elementele matricei vor fi păstrate în fapte compuse, definite de construcţia deftemplate *tablou*.

```
(detemplate tablou
        (slot linie) (slot coloana) (slot valoare))

(defrule citire_matrice
    (dimensiune-linie ?M)
    (dimensiune-coloana ?N)
    =>
    (loop-for-count (?i 1 ?N)
        (loop-for-count (?j 1 ?M)
                (printout  t  " A[ "  ?i  ","  ?j " ]= ")
                (assert (tablou (linie ?i) (coloana ?j) (valoare (read))))
)   )   )
```

Foarte important de reținut este faptul că într-o buclă de tip *loop-for-count* nu se poate modifica valoarea variabilei de index. Au fost implementate două reguli pentru afişarea numerelor impare de la 1 la 10. Pentru regula în care se foloseşte *(bind ?i (+ ?i 1))* în interiorul buclei *for* a cărei variabilă de index este *?i*, CLIPS-ul va furniza următoarea eroare:

```
(defrule nr_impare_incorect                 (defrule nr_impare_corect
    =>                                          =>
    (loop-for-count  (?i 1 10)                  (loop-for-count (?i 1 10)
        (printout  t  ?i t)                         (if (oddp ?i)
        (bind  ?i  (+ ?i 1)) ) )                     then (printou t ?i t)) ) )
```

[PRCDRPSR1] Cannot rebind loop variable in function loop-for-count.

Dacă se doreşte implementarea unor bucle *for* fără a folosi o funcție procedurală se pot folosi în locul variabilelor index, fapte de control care să contorizeze numărul de execuții al fiecărei bucle.

```
                (deffacts date_initiale
                    (contor 1) )

(defrule bucla_for                          (defrule bucla_for
    =>                                          ?f<-(contor ?c &: (< ?c 10))
    ….....instr. bucla for..... )               =>
(defrule index                                  (retract ?f)
    ?f <- (contor ?c & ~10)                     ….....instr. bucla for.....
    =>                                          (assert (contor (+ ?c 1))) )
    (retract ?f)
    (assert (contor (+ ?c 1)))
    (refresh bucla_for) )
```

1.4. Funcția SWITCH

După cum se ştie un **switch** ar putea fi realizat cu mai multe funcții **if**, din acest motiv putem să ne imaginăm această funcție ca o cascadă de condiții, toate fiind aplicate asupra unei singure variabile sau expresii. Sintaxa funcției este:

```
( switch  <test-expression>
        ( case  <comparison-expression>  then  <action>*  )+
        [ ( default <action>* ) ]   )
```

Partea de *default* a unei funcții *switch* este opțională şi se execută numai atunci când expresia testată nu are nici una din valorile menționate în *case*-uri. În cazul în care partea *default* lipseşte şi expresia nu este egală cu nici una din valorile de comparație din *case*-uri atunci nu se execută nici o acțiune.

În exemplul de mai jos se aşteaptă introducerea unui răspuns de la tastatură şi în funcție de acesta se vor executa diferite acțiuni. Pentru realizarea buclei se va folosi de această dată comanda **refresh** (al cărei efect este acela că reactivează regula în agendă).

```
(defrule start-again
        (declare (salience -10))
        =>
        (printout t "Doresti sa începi jocul din nou (da / nu) ?")
        (switch (read)
                (case da then
                                (reset) (run))
                (case nu then
                                (assert (phase stop)) )
                (default
                        (printout t "Introduceti (da / nu)" crlf)
                        (refresh start-again) )
)       )
```

Spre deosebire de limbajul C unde era necesară execuţia unui *break*, pentru a nu fi executate şi acţiunile din *case*-urile imediat următoare celui pentru care expresia de test este egală cu valoarea de comparat, în CLIPS avem certitudinea că acest lucru nu se întâmplă niciodată.

1.5. Funcţia PROGN

Funcţia **progn** evaluează toate argumentele pe care le are şi returnează valoarea ultimului argument evaluat. Se pot da astfel comenzi la prompter care altfel nu ar fi avut nici un efect. Să presupunem un mic exemplu:

```
CLIPS> (bind ?v 1)
1
CLIPS> (printout t ?v)
[EVALUATN1] Variable v is unbound.
```

Dacă s-ar încerca execuţia celor două funcţii una după alta, fără ca una din funcţii să fie argumentul celeilalte (ca în cazul *(bind ?v (read))*) CLIPS-ul ar executa doar prima funcţie (ignorând tot ce urmează după aceasta). Practic în momentul în care numărul parantezelor închise este identic cu numărul parantezelor deschise, se consideră ca fiind terminat codul funcţiei de executat.

```
CLIPS> (printout t " Pamant " t)   (printout t "Soare" t)
Pamant
CLIPS>
```

Este cât se poate de clar în acest moment, faptul că o variabilă locală căreia i-a fost atribuită o valoare cu ajutorul funcţiei *bind*, este inexistentă în momentul în care se introduce la prompter o a doua funcţie ce are ca argument variabila anterior declarată. Dacă cele două funcţii ar fi făcut parte din RHS-ul unei reguli atunci situaţia ar fi fost cu totul diferită. CLIPS-ul oferă prin intermediul funcţiei **progn** posibilitatea execuţiei mai multor funcţii direct de la prompter.

```
CLIPS> (progn (bind ?v 1) (printout t "Variabila: " ?v "." crlf) )
Variabila : 1.
```

Sintaxa funcției este următoarea:

(**progn** <expression>*)

unde putem folosi ca parametri <expression> orice fel de expresii sau funcții, în cele din urmă funcția returnând valoarea ultimei expresii evaluate. În cazul în care funcția *progn* nu are nici un parametru, valoarea returnată este simbolul FALSE.

```
CLIPS> (progn)
FALSE
CLIPS> (progn (evenp 2)  (printout  t  "Suma : ")  (+ 1 1) )
Suma : 2
CLIPS> (progn (+ 1 1) (mod 3 4) (integer 3.7))
3
CLIPS> (progn (loop-for-count  (?i 1 3) do  (printout  t "Numar " ?i "  ")) )
Numar 1  Numar 2  Numar 3 FALSE
```

Utilitatea funcției *progn* se observă la prima vedere numai la folosirea ei la prompter. În cadrul unei reguli în partea de acțiuni RHS folosirea acesteia este inutilă, dar introducerea funcției *progn* în LHS duce la realizarea unui fapt altfel imposibil de realizat: "execuția unei acțiuni în partea de condiții a unei reguli". Astfel în exemplele următoare deşi RHS-ul regulii nu conține nici o instrucțiune se vor executa acțiuni în LHS:

```
CLIPS> (assert (numar))
<Fact-0>
CLIPS> (defrule incredibil-1
                ?f <- (numar $?)
                (test (progn (printout  t  "Faptul care aprinde regula: " ?f)))
                =>
        )
Faptul care aprinde regula: <Fact-0>
CLIPS> (assert (numar 7))
Faptul care aprinde regula: <Fact-1> <Fact-1>
CLIPS> (run)
```

Se observă faptul că mesajul este afişat în momentul în care regula este activată în agendă şi nu atunci când aceasta este lansată în execuție cu ajutorul comenzii *run*. La a doua activare se observă apariția lui <*Fact-1*> de două ori, deoarece prima dată este afişat de funcția *printout* inclusă în funcția *progn*, iar a doua oară este returnat de funcția *assert*.

```
CLIPS> (assert (lista 1 2 3 4))
CLIPS> (defrule incredibil-2
                (lista $?n)
                (test (progn   (loop-for-count (?i 1 (length ?n))
                (printout  t  ?i ",") ))  ))
                =>
         )
1,2,3,4  CLIPS>
```

1.6. Funcţia PROGN$

Funcţia **progn$** realizează un set de acţiuni pentru fiecare câmp al unei valori multifield. Se realizează deci o buclă cu un număr de iteraţii egal cu numărul valorilor multicâmpului, iar câmpul iteraţiei curente poate fi examinat cu ajutorul lui <*loop-variable*>, dacă acesta este specificat. Pentru a examina indexul iteraţiei curente se poate folosi variabila specială <*loop-variable*>-***index***. Funcţia *progn$* poate folosi variabile pentru scopuri exterioare, iar funcţiile *return* şi *break* pot fi de asemenea folosite atâta timp cât acel scop exterior este valid. Valoarea returnată de această funcţie este valoarea obţinută în urma evaluării ultimei acţiuni efectuate asupra ultimului câmp din valoarea multicâmp-ului. Sintaxa funcţiei este:

(**progn$** <list-spec> <expression>*)

<list-spec> ::= <multifield-expression> |
<list-variable> <multifield-expression>

Un exemplu de utilizare al acestei funcţii, însoţit de exemplificarea folosirii lui *index* poate fi următorul:

```
CLIPS> (progn$ (?field  (create$ abc def ghi))
                (printout  t  "--> "  ?field  " "  ?field-index   " <-- " crlf))
--> abc 1 <--
--> def 2 <--
--> ghi 3 <--
CLIPS>
```

1.7. Funcţia BREAK

Această funcţie este folosită atunci când se doreşte terminarea imediată a unei bucle *while*, a unei bucle *for* sau a execuţiei unei instrucţiuni *progn*. Uneori poate fi foarte utilă ieşirea dintr-o buclă în cazul îndeplinirii unei condiţii, iar acest lucru poate duce uneori la o optimizare a funcţionării unui program şi la obţinerea mai rapidă a unui rezultat. Fie următoarele exemple în care este utilizată oprirea forţată a execuţiei unei bucle.

```
(defrule break-1
        =>
        (loop-for-count (?i 1 5)
                (if (= ?i 3) then (break))
                (printout t "mesaj " ?i crlf)
)       )
(defrule break-2
        =>
        (printout  t  "Apasa Q daca vrei sa iesi din bucla..." crlf)
        (while TRUE do
                (if (eq (read) q) then (break))
)       )
```

Se observă în exemplul al doilea cum se poate realiza o buclă infinită folosindu-se simbolul TRUE în partea de condiție a unei funcții *while*. În bucla infinită pe care programatorul o formează în mod intenționat, se stă până în momentul în care se introduce de la tastatură simbolul *q*.

```
CLIPS> (progn$  (?field  (create$ abc def ghi))
                (if (= ?field-index 2)
                  then (break)
                  else (printout  t  "--> "  ?field  " "  ?field-index   " <-- " crlf)))
--> abc 1 <--
FALSE
```

În acest ultim exemplu se poate observa cum în cadrul funcției *progn$*, instrucțiunea *break* oprește execuția buclei formate datorită evaluării parametrului multifield creat de funcția *(create$ abc def ghi)*. Deoarece execuția funcției *progn$* a fost întreruptă brusc, se va returna simbolul FALSE.

1.8. Funcția RETURN

Funcția **return** are ca efect terminarea imediată a execuției curente a unei *deffunction, generic function method, message-handler* sau *RHS-ul unei reguli*. Fără nici un argument nu întoarce nici o valoare, în caz contrar pentru deffunction, method sau message-handler, dacă este inclus vreun argument se returnează valoarea obținută în urma evaluării acestuia. Sintaxa funcției este:

(**return** [<expression>])

Dacă este utilizată ca acțiune în RHS-ul unei reguli, atunci putem observa cum este şters *focus-ul* curent din **focus-stack**. Trebuie reținut faptul că nu putem folosi funcția *return* ca argument al vreunei funcții.

În funcția următoare definită de utilizator *sign*, dacă nu se execută acțiunile nici unei instrucțiuni *if* se returnează ultima expresie evaluată (respectiv numărul 0). În acest exemplu este echivalentă folosirea valorii explicite 0 cu funcția *(return 0)*, dar utilizarea funcției *return* ca ultimă instrucțiune nu se justifică, dat fiind faptul că aceasta de obicei trebuie să oprească execuția unei funcții sau reguli.

```
CLIPS> (deffunction sign (?num)
                  (if  (> ?num 0)  then (return 1))
                  (if  (< ?num 0)  then  (return –1))
        0 )
CLIPS> (sign 4)
1
CLIPS> (sign –8)
-1
CLIPS> (sign 0)
0
```

1.9. Problema comisului voiajor – abordare procedurală

Pentru clasica problemă a comisului voiajor, în care pentru un număr de oraşe date se pune problema trecerii prin fiecare oraş cu un cost minim, vom încerca o abordare procedurală, după un algoritm care este mai indicat să fie aplicat într-un limbaj procedural (C sau Pascal). Deşi acest algoritm poate fi aplicat şi în CLIPS, după cum se va putea observa parcurgerea codului este foarte anevoioasă (în special datorită excesului de funcţii procedurale). Problema poate fi rezolvată şi folosind funcţii definite de utilizator cu ajutorul construcţiei *deffunction*. Algoritmul scris în pseudocod este următorul:

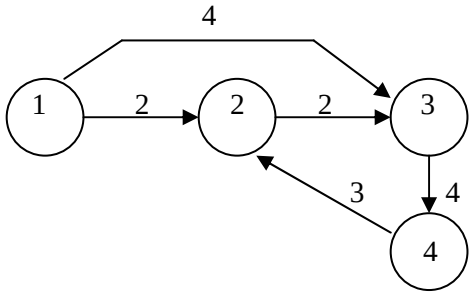
```
Variabile:  matr[n][n], drum[n], stari[n]
            este_valid, are_succesor, rez, min=MAX

drum_minim(n, init)                                  ; n-numărul de oraşe
{       stari[1]=init                                ;se porneşte din oraşul init
        k=2
        stari[k]=1
        while (k>=1)
        {
            do{
              succesor(k ,n)                         ; k este oraşul curent
              if (are_succesor)
                      valid(k, n)
              }while (are_succesor && !este_valid)
              if(are_succesor)
                      if (k==n)
                      {       tipar(k)
                              calc_cost(k)
                              if (rez < min)
                              {       min=rez,  drum <-- stari       }

                      }else
                      {       k++;
                              stari[k]=1
                              calc_cost(k-1)
                              if (rez <= min)
                                      k++
                              else
                                      k--
                      }
              else
                      k--
        }
}
```

```
valid(k. n)
{       este_valid=1
        if (matr[stari[k-1],stari[k]]<=0)          ; dacă nu există legătură
                este_valid=0                       ; între ultimele două oraşe
        if (nodul stari[k] este în stivă)
                este_valid=0
        if(k==n && matr[stari[n],stari[1]]<=0)     ; dacă nu există arc între
                este_valid=0                       ; primul şi ultimul oraş
}
succesor(k, n)
{
        if (stari[k]<=n)
        {       stari[k]++, are_succesor=1      }
        else    are_succesor=0
}
calc_cost(k)
{
        rez=0;
        for(i=1; i<k; i++)
                rez=rez+matr[stari[i], strari[i+1]
}
```

Fără a explica acest algoritm, se precizează doar faptul că în matricea costurilor dacă între două oraşe nu există o legătură directă valoarea elementelor a căror linie şi coloană sunt cele două oraşe este un număr <= 0 (altfel spus dacă între nodurile i şi j nu există un arc, atunci *matr[i, j]<=0* şi *matr[j, i]<=0*).



n = 4 oraşe

matrice cost $\begin{pmatrix} 0 & 2 & 4 & 0 \\ 2 & 0 & 2 & 3 \\ 4 & 2 & 0 & 4 \\ 0 & 3 & 4 & 0 \end{pmatrix}$

Se oferă posibilitatea citirii datelor dintr-un fişier sau direct de la tastatură. O altă posibilitate ar fi fost citirea acestora dintr-o construcție *deffacts*. Deoarece matricea costurilor este simetrică în raport cu diagonala principală, se vor introduce doar elementele de deasupra acesteia. Astfel fişierul *comis_da.txt* poate conține următoarele date:

```
4                   ;numărul de oraşe
0 2 4 0             ;matrice costuri
  0 2 3
    0 4
      0
```

Dacă se doreşte citirea datelor de la tastatură se poate realiza un fişier de comenzi *comis.bat* în care se pot introduce toate comenzile şi datele de intrare, ce ar trebui introduse la prompter, atunci când se lansează programul în execuție.

```
(deffacts init (faza init-date))

(defrule init-date
     ?faza <- (faza init-date)
     (not (citire-tastatura))
     =>
     (retract ?faza)
     (printout t t "Datele de intrare dintr-un fisier (y/n) ? ")
     (while (neq (bind ?answer (read))  y  n) do
          (printout t "Alege (y sau n) ! " crlf) )
     (if (eq ?answer y)
      then (if (open "comis_da.txt" data "r")
             then (assert (citire-din-fisier)) (halt) ) )
     (assert (citire-tastatura))
)

(defrule citire-fisier
     ?input <- (citire-din-fisier)
     =>
     (retract ?input)
     (bind ?eroare FALSE)  (bind ?m 0)  (bind ?max 0)  (bind ?nr_orase (read data))
     (if (or (not (integerp ?nr_orase)) (< ?nr_orase 2))
      then (bind ?eroare TRUE)
      else (printout t "Numar orase : " ?nr_orase "." crlf)
           (assert (nr_orase ?nr_orase))
           (loop-for-count (?i 1 (- (* ?nr_orase ?nr_orase) 1)) do
                 (bind ?m (create$ ?m 0)) )
           (printout t "Distantele intre orase :" crlf)
           (loop-for-count (?i 1 ?nr_orase) do
                (loop-for-count (?j ?i ?nr_orase) do
                     (bind ?valoare (read data))
                     (if (or (not (integerp ?valoare)) (< ?valoare 0))
                      then (bind ?eroare TRUE)
                      else (bind ?loc1 (+ ?j (* ?nr_orase (- ?i 1))))
                           (bind ?loc2 (+ ?i (* ?nr_orase (- ?j 1))))
                           (bind ?m (replace$ ?m ?loc1 ?loc1 ?valoare))
                           (bind ?m (replace$ ?m ?loc2 ?loc2 ?valoare))
                           (bind ?max (max ?max ?valoare))
                           (if (neq ?i ?j)
                            then (printout t "Oras" ?i "- oras" ?j " : " ?valoare t)
     )     )     )   )      )
     (if (eq ?eroare TRUE)
      then(assert (citire-tastatura))
           (close data)
           (printout t "Datele din fisier nu sint corecte ! "crlf)
           (printout t "Introduceti datele de la tastatura ..." crlf)
      else
           (close data)
           (assert (dist_min(* ?max ?nr_orase))   (matrice ?m)   (faza cautare-drum)))
)
```

```
(defrule stergere-date-eronate
    (citire-tastatura)
    ?fact <- (matrice)
    =>
    (retract ?fact)
)

(defrule citire-tastatura
    ?input <- (citire-tastatura)
    =>
    (retract ?input)
    (bind ?m 0)
    (bind ?max 0)
    (printout t crlf "Numar orase : ")
    (while (or (not (integerp (bind ?nr_orase (read)) )) (< ?nr_orase 2)) do
        (printout t "Alege un intreg pozitiv (>2) !" crlf)
        (printout t crlf "Numar orase : ") )
    (assert (nr_orase ?nr_orase))
    (loop-for-count (?i 1 (- (* ?nr_orase ?nr_orase) 1)) do
        (bind ?m (create$ ?m 0)) )
    (printout t "Distantele intre orase :" crlf)
    (loop-for-count (?i 1 ?nr_orase) do
        (loop-for-count (?j ?i ?nr_orase) do
            (if (neq ?i ?j)
             then (printout t "Oras" ?i " - oras" ?j " : ")
                  (while (or(not (integerp (bind ?valoare (read)) )) (< ?valoare 0)) do
                      (printout t "Alege un intreg pozitiv (>=0) !" crlf crlf)
                      (printout t "Oras" ?i " - oras" ?j " : " crlf) )
                  (bind ?max (max ?max ?valoare))
                  (bind ?loc1 (+ ?j (* ?nr_orase (- ?i 1))))
                  (bind ?loc2 (+ ?i (* ?nr_orase (- ?j 1))))
                  (bind ?m (replace$ ?m ?loc1 ?loc1 ?valoare))
                  (bind ?m (replace$ ?m ?loc2 ?loc2 ?valoare)) )))
    (assert (dist_min (* ?max ?nr_orase))
            (matrice ?m)
            (faza cautare-drum))
)

(defrule cautare-drum-minim
    ?faza <- (faza cautare-drum)
    (nr_orase ?nr_orase)
    ?dist <- (dist_min ?dist_min)
    (matrice $?m)
    =>
    (retract ?dist ?faza)
    (open "comis_re.txt" rezultate "w")
    (bind ?st 0)
    (loop-for-count (?i 1 (- ?nr_orase 1)) do
        (bind ?st (create$ ?st 0)))
    (printout t "Din ce oras se pleaca (1-" ?nr_orase ") ? ")
```

```
(while (or (not (integerp (bind ?nod_initial (read)) ))
           (< ?nod_initial 1) (> ?nod_initial ?nr_orase) )
    (printout t "Alege un intreg pozitiv (1-" ?nr_orase ") !" crlf crlf)
    (printout t "Din ce oras se pleaca (1-" ?nr_orase ") ? ") )
(bind ?st (replace$ ?st 1 1 ?nod_initial))
(bind ?k 2)
(bind ?stare_k 0)
(while (> ?k 1) do
   (bind ?st (replace$ ?st ?k ?k ?stare_k))
   (bind ?succesor 1)
   (bind ?valid 0)
   (while (and (eq ?succesor 1) (neq ?valid 1)) do
      (if (< ?stare_k ?nr_orase)
       then (bind ?stare_k (+ ?stare_k 1))
            (bind ?st (replace$ ?st ?k ?k ?stare_k))
            (bind ?succesor 1)
       else (bind ?succesor 0))
      (if (eq ?succesor 1)
       then (bind ?valid 1)
            (if (eq (nth$ (+ ?stare_k (* ?nr_orase (- (nth$ (- ?k 1) ?st) 1))) ?m) 0)
            then (bind ?valid 0)
            else (if (and (eq ?k ?nr_orase) (eq (nth$ (+ ?nod_initial
                                  (* ?nr_orase (-(nth$ ?nr_orase ?st)1))) ?m)0))
                 then (bind ?valid 0)
                 else (loop-for-count (?i 1 (- ?k 1)) do
                         (if (eq ?stare_k (nth$ ?i ?st))
                          then (bind ?valid 0) ))) ))
   )
   (if (eq ?succesor 1)
    then (if (eq ?k ?nr_orase)
         then (bind ?rez 0)
              (loop-for-count (?i 1 (- ?k 1)) do
                 (bind ?rez (+ ?rez (nth$ (+ (nth$ ?i ?st)
                                  (* ?nr_orase (- (nth$ (+ ?i 1) ?st) 1))) ?m))) )
              (bind ?rez (+ ?rez (nth$ (+ (nth$ ?nr_orase ?st)
                                  (* ?nr_orase (- ?nod_initial 1) ) ) ?m)))
              (loop-for-count (?i 1 ?nr_orase) do
                    (printout rezultate (nth$ ?i ?st) " " crlf))
              (printout rezultate ?rez crlf)
              (if (<= ?rez ?dist_min)
               then (bind ?dist_min ?rez)
                    (bind ?dr (create$ ?st)))
         else (bind ?rez 0)
              (loop-for-count (?i 1 (- ?k 1)) do
                    (bind ?rez (+ ?rez (nth$ (+ (nth$ ?i ?st)
                                  (* ?nr_orase (- (nth$ (+ ?i 1) ?st) 1))) ?m))) )
              (if (<= ?rez ?dist_min)
               then (bind ?st (replace$ ?st ?k ?k ?stare_k))
                    (bind ?k (+ ?k 1))
                    (bind ?stare_k 0 ) ))
```

```
            else (bind ?st (replace$ ?st ?k ?k 0))
                 (bind ?k (- ?k 1))
                 (bind ?stare_k (nth$ ?k ?st)))
    )
    (printout t crlf  "Drum minim : " crlf)
    (loop-for-count (?i    1 ?nr_orase) do
          (printout t (nth$ ?i ?dr) " "))
    (printout t " ==> " ?dist_min "." crlf)
    (assert (dist_min ?dist_min)
              (drum_minim ?dr)
              (afisare-rezultate))
    (printout rezultate end)
    (close rezultate)
)

(defrule afisare_solutii
     (afisare-rezultate)
     (dist_min ?dist_min)
     (nr_orase ?nr_orase)
     (drum_minim $?dr)
     =>
     (printout t crlf "Solutii :" crlf)
     (open "comis_re.txt" rezultate "r")
     (bind ?exit 0)
     (while (neq ?exit 1) do
          (loop-for-count (?i 1 ?nr_orase) do
               (bind ?valoare (read rezultate))
               (bind ?dr (replace$ ?dr ?i ?i ?valoare))
               (if (eq ?valoare end)  then  (bind ?exit 1) ) )
          (bind ?rez (read rezultate))
          (if (eq ?rez end)  then  (bind ?exit 1) )
          (if (eq ?rez ?dist_min)
           then (loop-for-count (?i 1 ?nr_orase) do
                        (printout t (nth$ ?i ?dr) " ") )
                (printout t "==> "?rez "." crlf) )
     )
     (close rezultate)
     (assert (reluare))
)

(defrule reluare
     (reluare)
     =>
     (printout t crlf "Doresti sa rulezi din nou acest program (y/n) ? ")
     (while (neq (bind ?raspuns (read)) y n Y N) do
          (printout "Apasa y sau n ...!" crlf) )
     (if (or (eq ?raspuns y) (eq ?raspuns Y))
      then (reset))
)
```

1.10. Problema comisului voiajor – abordare logică

Diferenţele între programarea procedurală şi cea logică, constau în faptul că cea logică este mai apropiată de modul de gândirea al omului, pe când cea procedurală presupune cunoaşterea din partea programatorului a unei rezolvări a problemei şi descrierea acesteia sub forma unei suite de instrucţiuni executabile. Dacă în programarea logică programatorul nu este responsabil de rezolvarea problemei, acesta descriind într-o manieră cât mai declarativă problema şi lăsând sistemul să caute şi să furnizeze soluţiile, într-o descriere procedurală a problemei soluţiile furnizate sunt limitate de corectitudinea programului.

Programatorul trebuie să găsească într-o abordare logică doar legile şi datele ce descriu problema, şi sistemul va căuta şi furniza toate soluţiile găsite. În varianta de mai sus, erau găsite de asemenea toate soluţiile posibile şi scrise într-un fişier "*comis_re.txt*", în final afişându-se doar drumurile care aveau un cost minim.

De această dată se vor introduce datele de intrare cu o construcţie *deffacts*, însă acestea pot fi încărcate şi direct dintr-un fişier cu ajutorul comenzii *load-facts*, (dacă se respectă condiţia ca numărul de paranteze deschise să fie egal cu numărul de paranteze închise). În CLIPS se poate implementa foarte uşor un arbore sau un graf prin simpla asertare a unor fapte care să precizeze legăturile dintre două noduri. Depinde doar de programator dacă graful sau arborele respectiv va avea sau nu deplasarea în lungul arcelor bidirecţională. Se presupune că se pleacă din nodul A, costul iniţial fiind 0.

```
(deffacts date_initiale
        (segm A B 2)  (segm A C 4)  (segm B C 2)  (segm B D 3)  (segm C D 4)
        (nr_noduri 4)  (min 100)  (drum A 0) )

(defrule adauga_oras
        (drum  $?first  ?x  ?cost1)
        (or  (segm  ?x  ?y  ?cost2)
             (segm  ?y  ?x  ?cost2))
        (test (not (member$  ?y  $?first)))              ;orasul ?y nu este pe drum
        =>
        (assert (drum  $?first ?x  ?y (+ ?cost1 ?cost2) )) )

(defrule drum_minim
        (nr_noduri ?n)
        (drum  $?dr &:(= ?n (length$ $?dr)) ?cost)
        ?f <- (min ?m)
        =>
        (if (< ?cost ?m)  then  (retract ?f)  (assert (min ?cost)) ))

(defrule afisare_solutii
        (declare(salience –10))
        (min ?m)
        (nr_noduri ?n)
        (drum  $?dr &:(= ?n (length$ $?dr)) ?m)
        =>
        (printout  t "Drum minim: " $?dr " Cost: " ?m crlf) )
```

În acest moment programul va furniza drumul (drumurile) ce trec prin cele N oraşe cu un cost minim. Se observă faptul că sensul de deplasare pe un arc este bidirecțional deoarece prin CE *(or (segm ?x ?y ?cost2) (segm ?y ?x ?cost2))* din regula *adaugă_oraş*, se acceptă deplasarea între două oraşe în ambele sensuri.

Spre deosebire de soluțiile furnizate de programul anterior (varianta procedurală), soluțiile furnizate de acest program nu respectă şi condiția ca din ultimul oraş din listă să se poată ajunge şi în primul. Dacă dorim să introducem şi această condiție este suficient să modificăm pattern-urile regulii *drum_minim*, adăugând faptului *drum* o restricție suplimentară.

```
(defrule drum_minim
        (nr_noduri ?n)
        ?f <- (min ?m)
        (drum  ?first $?dr &:(= (length$ $?dr) (- ?n 2))  ?last  ?cost &:(< ?cost ?m))
        (or  (segm  ?first  ?last  ?)
             (segm  ?last  ?first  ?))
        =>
        (retract ?f)
        (assert (min ?cost)) )
```

Spre deosebire de regula anterioară care era activată de toate faptele *drum* existente, introducerea în LHS a testului *(< ?cost ?m)* legat de variabilă *?cost* şi nu folosind o funcție *if* în RHS, scade foarte mult numărul faptelor ce vor activa regula (doar faptele *drum* cu un cost mai mic decât *min*).

După cum se poate observa în acest moment deşi ambele programe furnizează aceleaşi soluții, în costul final nu este adăugat şi costul dintre ultimul şi primul oraş (acesta este ignorat datorită variabilei libere). Dacă se doreşte problema comisului voiajor poate fi modificată, astfel încât să se caute drumul minim prin toate cele N oraşe, pornind dintr-un oraş şi ajungând înapoi în acesta.

De asemenea se mai pot aduce unele optimizări, dacă costul unui drum depăşeşte valoarea minimă din acel moment să nu se mai adauge nici un oraş în lista respectivă (soluția nu mai poate fi minimă). Această condiție suplimentară nu ar trebui adăugată regulii adaugă oraş, dacă se doreşte afişarea tuturor soluțiilor valide şi nu doar a drumului minim.

```
(deffacts date_initiale
        (segm A B 1) (segm A C 3) (segm A D 5) (segm A E 7) (segm B C 4)
        (segm B D 5) (segm B E 8) (segm C D 3) (segm C E 5) (segm D E 9)
        (nr_noduri 5)
        (drum A 0)  (min 100) )

(defrule adauga_oras
        (drum  $?first  ?x  ?c1)
        (or  (segm  ?x  ?y  ?c2)
             (segm  ?y  ?x  ?c2))
        (test (not (member$  ?y  $?first)))          ;orasul ?y nu este pe drum
        (min ?m&:(< (+ ?c1 ?c2) ?m) )                ; cost actual mai mic decât min
        =>
        (assert (drum  $?first ?x  ?y (+ ?c1 ?c2)) ) )
```

```
(defrule drum_minim
        (nr_noduri ?n)
        ?f <- (min ?m)
        (drum  ?first $?dr &:(= (length$ $?dr) (- ?n 2))  ?last  ?cost1)
        (or  (segm  ?first  ?last  ?cost2)
             (segm  ?last  ?first  ?cost2))
        =>
        (bind ?cost (+ ?cost1 ?cost2))
        (assert  (solutie ?first  $?dr  ?last ?cost))
        (if(< ?cost ?m)
         then (retract ?f) (assert (min (+ ?cost1 ?cost2))) ) )
(defrule afisare_solutii
        (declare(salience –10))
        (min ?m)
        (solutie $?dr ?m)
        =>
        (printout  t "Drum minim: " $?dr " Cost: " ?m crlf) )
```

În acest exemplu au fost folosite litere mari pentru numele oraşelor. Dacă se doreşte, numele acestora pot fi înlocuite cu simboluri, şiruri sau chiar numere. Astfel în interiorul construcției *deffacts* faptele ce descriu distanțele dintre oraşe ar putea arătă astfel: *(segm Iasi Bucuresti 350)* sau *(segm 1 2 50)*. Se poate realiza o regulă care să stabilească în mod interactiv oraşul din care se pleacă, acest fapt presupunând mai apoi verificarea existenței oraşului respectiv, sau altfel spus a corectitudinii datelor de intrare. Printr-un fapt *(nr_orase 5)* se poate specifica numărul de oraşe prin care comisul voiajor trebuie să treacă. Dacă acest număr este mai mic decât numărul total de oraşe (sau chiar mai mare), se obține o variantă originală a problemei în care comisului voiajor i se impune trecerea printr-un anumit număr de oraşe.

Realizând o comparație între cele două abordări se poate observa eleganța şi simplitatea variantei logice în comparație cu cea procedurală. Nu toate problemele pot fi rezolvate aşa de uşor folosind un mediu de programare non-procedural, şi de aceea programatorul în funcție de abilitățile şi experiența sa trebuie să aleagă acel mediu de programare, care este cel mai potrivit cerințelor problemei.

1.11. Întrebări

1. Dacă într-o funcție *if* expresia condițională este evaluată la numărul 0 cum se vor executa acțiunile clauzei *then* sau cele ale clauzei *else* ?
2. Există vreo diferență între simbolul FALSE şi string-ul "FALSE" în execuția unei instrucțiuni condiționale *if* ?
3. Într-o funcție *if* cuvântul cheie *else* este obligatoriu sau opțional?
4. Implementați o buclă de tip DO_WHILE (o buclă ce se execută cel puțin o singură dată) folosind o buclă *while*?

5. Precizați dacă bucla următoare este infinită sau nu ?

   ```
   (while (neq (read) "n" "N") do …)
   ```

   Dar bucla `(while (neq (readline) n N) do …)` ?

6. Care sunt diferențele între o buclă de tip *loop-for-count* şi o buclă de tip *while*? Puteți implementa un tip de buclă folosind celălalt tip şi eventual folosind şi instrucțiunea *break*?

7. Într-o buclă de tip *for* se poate modifica variabila index? Dacă da precizați care va fi efectul rulării buclei din exemplul următor?

   ```
   (loop-for-count  (?i 1 10)
         (printout  t  ?i t)
         (bind  ?i  (+ ?i 1)) )
   ```

8. Care este efectul obținut în urma execuției în RHS-ul unei reguli (sau într-o instrucțiune *progn*) a următoarelor instrucțiuni?

   ```
   (bind  ?i  TRUE)
   (loop-for-count  ?i  (printout  t  "Mesaj…" t))
   ```

9. Care este efectul unei instrucțiuni *switch* în care partea *default* lipseşte şi expresia de test nu este identică cu nici una din valorile din *case*-uri ?

10. Într-o instrucțiune *switch* pentru a nu fi executate şi acțiunile din *case*-urile imediat următoare celui la care expresia de test este egală cu valoarea de comparat este necesară execuția unui *break* ?

11. Ce fac funcțiile *progn* şi *progn$* şi în ce condiții este avantajoasă folosirea acestor funcții într-un program?

12. Ce returnează funcția *progn* şi *progn$*?

13. Care este valoarea returnată de funcția *progn* în cazul în care nu este folosit nici un parametru? Dar funcția *progn$* ?

14. Pot fi folosite acestea în LHS-ul regulii (partea premizelor), dar în RHS (partea acțiunilor)? Care sunt avantajele obținue în urma folosirii acestor instrucțiuni în RHS-ul unei reguli?

15. Care dintre cele două funcții are nevoie de argumente? De ce tip trebuie să fie acestea? Pot fi folosite mai multe argumente sau doar un singur argument?

16. Argumentul unei funcții *progn$* trebuie obligatoriu să fie de tip multicâmp ? Pot fi folosite mai multe argumente multicâmp pentru aceiaşi funcție?

17. Intr-o instrucțiune *progn$* pot fi examinate câmpul iterației curente şi indexul iterației curente? Dacă da care sunt condițiile pentru a face aceasta?

18. Ce semnificație are variabila *?field*, dar *?field-index* în construcția următoare?

    ```
    (progn$  (?field  (create$ abc def ghi))
          (printout  t  "--> "  ?field  " "  ?field-index  " <-- " crlf))
    ```

Care ar fi fost efectul obţinut dacă variabilele ar fi avut o altă denumire decât cea a variabilei folosită la argumentul funcției? Această variabilă este de tip multicâmp sau de un singur câmp?

19. În exemplul de mai sus se realizează o buclă? Argumentaţi?
20. Funcția *break* poate fi folosită şi pentru oprirea execuției unei instrucțiuni *progn*, sau numai pentru terminarea unor bucle de tip *for* sau *while*? Ce efect are această instrucțiune folosită în interiorul unei bucle care se află într-o instrucțiune progn? Ce valoare va avea variabila *?i* după rulare ?

```
(progn (loop-for-count  (?i 1 10)
            (if (= ?i 3) then (break)) else (printout t ?i))
)
(bind ?i 5) )
```

21. Este utilă folosirea funcției *break* într-o buclă infinită formată în mod intenționat de programator ? Argumentați ? Puteți realiza o buclă infinită de tip *loop-for-count* ?
22. Care este efectul folosirii funcției *return* în RHS-ul unei reguli? Dar dacă se afla într-o buclă?
23. Dacă în se specifică o valoare pentru o funcție *return* în RHS-ul unei reguli aceasta este returnată? Dacă da cui? Dar în interiorul unei funcții definite de utilizator cu *deffunction*?

Probleme rezolvate

1. Limbaj Morse – Scrieți un program cu ajutorul căruia să poată fi citit un mesaj codificat în limbajul morse. După cum se ştie fiecare literă alfabetică are un cod corespunzător în acest limbaj format din puncte şi linii. Dacă se face un mic istoric al acestui cod în 1937, Samuel Finley Bresse, un inventator şi pictor american, a construit telegraful electromagnetic şi codul asociat, numit codul Morse. Alfabetul Morse codifică fiecare literă a alfabetului latin printr-un şir de semnale scurte (puncte) şi semnale lungi (linii), separate prin spaţii de durată variabilă. Lungimea unei linii este în telecomunicație de aproximativ trei ori mai mare decât lungimea unui punct. Semnalele care compun o literă sunt separate prin durata unui punct. Durata care separă două litere este cea a unei linii. Două cuvinte sunt separate prin durata a cinci puncte, iar sfârşitul unei propoziții este marcat de durata a şapte puncte.

Toate aceste informații legate de durata dintre două semnale, litere, cuvinte etc. nu sunt importante pentru aplicația noastră. Dacă s-ar fi citit un mesaj în cod Morse reprezentat printr-un fişier binar (0 reprezentând spațiul şi 1 un semnal) în care numărul de caractere 0 sau 1 consecutive ar fi fost proporționale cu durata semnalelor şi a spațiilor dintre acestea, s-ar fi putut încerca o decodificare mai apropiată de cea reală (partea HIGH a semnalului fiind reprezentată prin secvențe de 1 şi partea LOW prin secvențe de 0).

Codul Morse şi caracterele echivalente sunt reprezentate mai jos:

A • —	H • • • •	O — — —	V • • • —
B — • • •	I • •	P • — — •	W • — —
C — • — •	J • — — —	Q — — • —	X — • • —
D — • •	K — • —	R • — •	Y — • — —
E •	L • — • •	S • • •	Z — — • •
F • • — •	M — —	T —	
G — — •	N — •	U • • —	

Pentru secvențele de intrare şi de ieşire se vor folosi caracterele **'*'** reprezentând punctul, '-' pentru linie şi '/' pentru a delimita două litere din codul morse. Dacă la intrare nu se tastează nici un caracter, atunci programul de opreşte. La prompter ar trebui să se obțină o secvență de genul următor:

```
Enter a message (<CR> to end): * * * / - - - / * * * ┘
The message is  S O S
Enter a message (<CR> to end): ┘
CLIPS>
```

Codurile morse corespunzătoare celor 26 de litere ale alfabetului latin pot fi foarte simplu stocate cu ajutorul unei construcții *deffacts* ca în exemplul următor: *(deffacts code_morse (code A "*-") (code B "-***") (code C "-*-*") …*

Problema va fi structurată pe mai multe etape, astfel încât în momentul în care s-ar dori implementarea acesteia într-o variantă modulară (cu ajutorul construcției *defmodule*), fiecărui modul să îi corespundă o etapă. Într-o primă fază sunt asertate codurile morse (dacă ar fi fost introduse cu un *deffacts* această etapă nu ar fi fost necesară). După aceasta urmează o fază de introducere a datelor, şi imediat după aceasta o fază de verificare a corectitudinii datelor de intrare. Dacă unele caractere nu sunt recunoscute (sunt diferite de caracterele valide "*", "-", "\") se vor afişa mesaje de eroare. În faza imediat următoare se afişează mesajul descifrat. În ultima fază *restart* este interogat utilizatorul şi în funcție de răspunsul acestuia se reia sau nu programul (această fază este opțională).

```
(deffacts init     (phase initial))

(defrule morse-code
        ?fact <- (phase initial)
        =>
        (retract ?fact)
        (bind  ?s1  "A *-   | B -*** | C -*-* | D -**  | E *    | F **-* | G --*  | ")
        (bind  ?s2  "H **** | I **   | J *--- | K -*-  | L *-** | M --   | N -*   | ")
        (bind  ?s3  "O ---  | P *--* | Q --*- | R *-*  | S ***  | T -    | U **-  | ")
        (bind  ?s4  "V ***- | W *--  | X -**- | Y -*-- | Z --** |")
        (printout t t t t "       The Morse Code"  t  t  ?s1 t  ?s2  t  ?s3  t  ?s4  t  t)
        (printout t "For example the message  "***/---/***"  is  S.O.S." t t)
        (bind  $?val (explode$ (str-cat ?s1 ?s2 ?s3 ?s4)))
        (loop-for-count (?i 0 (- (/ (length$ $?val) 3) 1))
                (assert(code (nth$ (+(* ?i 3) 1) $?val)  (nth$ (+(* ?i 3) 2) $?val)) ))
        (assert (phase message)) )
```

```
(defrule insert_message
        ?fact <- (phase message)
        =>
        (retract ?fact)
        (printout t "Enter a message (<CR> to end) :")
        (if (eq (bind ?str (readline)) “”)
         then (return))
        (bind ?str (str-cat (expand$ (explode$ ?str))))
        (loop-for-count (?i 1 (str-length ?str)) do
                (if (neq (sub-string ?i ?i ?str)  "*"  "-"  "/" " ")
                 then (assert (phase message))
                        (printout t "Please use only  the characters '*'  '-'  '/'..." crlf)
                        (break)
        )       )
        (assert (message ?str)
                (phase verify)) )

(defrule transforme_message
        (declare (salience 5))
        (phase verify)
        ?msg <- (message ?str)
        =>
        (retract ?msg)
        (while (bind  ?poz  (str-index / ?str))
                (bind  ?str  (str-cat   (sub-string 1 (- ?poz 1) ?str)   " "
                                        (sub-string (+ ?poz 1) (str-length ?str) ?str)) ) )
        (assert (message (explode$ ?str))) )

(defrule verify_message
        ?fact <- (phase verify)
        ?msg <- (message $? ?morse_letter $?)
        (not (exists (code ? ?morse_letter)))
        =>
        (retract ?fact ?msg)
        (printout t "I don`t recognize \"" ?morse_letter "\" !" t) )

(defrule valid_message
        (declare (salience -5))
        ?fact <- (phase verify)
        =>
        (retract ?fact)
        (assert (phase print)) )

(defrule replace_letter
        (phase print)
        (code ?letter ?morse)
        ?msg <- (message $?first ?morse $?last)
        =>
        (retract ?msg)
        (assert (message $?first ?letter $?last)) )
```

```
(defrule print_message
        (declare (salience -5))
        ?fact <- (phase print)
        ?msg <- (message $?str)
        =>
        (retract ?fact ?msg)
        (assert (phase restart))
        (printout t "The message is : " ?str crlf) )

(defrule start_again
        ?fact <- (phase restart)
        =>
        (retract ?fact)
        (printout  t  "Try another message (Y/N)? ")
        (if (neq (read) Y  y  yes  Yes  YES)
         then else (reset) (run)) )
```

Încă de la început s-a dorit realizarea unei optimizări, în sensul că pentru a nu se mai introduce toate literele şi codul morse asociat lor într-o construcție *deffacts*, dat fiind faptul că oricum ar fi trebuit realizată şi o afişare a acestora pe ecran (existând o mică probabilitate ca utilizatorul programului să cunoască codul morse), se puteau folosi aceste mesajele, şi după concatenarea acestora cu funcția **str-cat,** şirul de caractere format fiind transformat într-un multifield cu ajutorul funcției **explode$**. În acest moment poate fi foarte uşor de extras într-o buclă *for* fiecare literă şi corespondentul ei în codul morse. Şirul de caractere a fost împărțit la trei deoarece pentru fiecare fapt *code* asertat se extrag (1) litera alfabetului latin şi (2) codul morse corespunzător (acesta formează un singur simbol deoarece nu s-au introdus spații între linii '-' şi puncte '*'), (3) bara '|' introdusă pentru obținerea unui efect vizual tabelar fiind neglijată.

În cea de a doua fază în regula *insert_message* se introduce de la tastatură mesajul ce trebuie tradus. Dacă s-au introdus de la tastatură şi caractere nepermise atunci prin reasertarea faptului *(phase message)* forțăm regula să se reaprindă şi astfel să se reia operația de introducere a mesajului. Se produce astfel o buclă din care se iese doar dacă mesajul introdus conține numai caractere valide. Se verifică şi dacă şirul de caractere introdus este vid (dacă s-a tastat doar *Enter*).

(if (eq (bind ?str (readline)) "") then (return))

Dacă mesajul este vid, atunci funcția *return* opreşte execuția regulii şi în acest caz nu mai sunt asertate faptele de control *phase* ce vor aprinde celelalte reguli. Dacă în locul faptelor de control ar fi fost folosite priorități şi fiecărei faze i s-ar fi asociat un anumit salience, funcția *return* nu ar fi putut opri decât execuția regulii curente, nu şi execuția programului (celelalte reguli aprinzându-se). Pentru eliminarea spațiilor dintre caracterele introduse a fost folosit un mic artificiu:

(bind ?str (str-cat (expand$ (explode$?str))))

Funcția *explode$* transformă şirul de caractere într-un multicâmp, care este mai apoi 'rupt' în valori individuale cu funcția *expand$*, valori ce pot fi concatenate cu

funcţia *str-cat* fără a se mai lua în calcul şi spaţiile individuale. Dacă această linie a programului este comentată, se poate observa cum pentru o secvenţă de intrare: “ *** / --- / * * * “ şirul descifrat este S O E E E, fiind luate în consideraţie şi spaţiile dintre ultimele asteriscuri.

În faza imediat următoare de verificare se fac asupra şirului o serie de transformări, şi anume înlocuirea într-o buclă *while* caracterului ‘/’ cu un spaţiu. Se foloseşte funcţia **str-index**, care returnează un număr întreg reprezentând prima poziţie în care a fost găsit caracterul în şir, sau simbolul FALSE dacă caracterul căutat nu se află în şir. În interiorul condiţiei buclei *while* se observă folosirea funcţiei **bind** (expresia returnată de aceasta fiind comparată cu simbolul FALSE).

```
(while (bind  ?poz  (str-index / ?str))
       (bind  ?str  (str-cat  (sub-string 1 (- ?poz 1) ?str)   " "
                              (sub-string (+ ?poz 1) (str-length ?str) ?str)) ) )
```

În final cu ajutorul funcţiei **explode$** este asertat noul mesaj sub forma unei valori multicâmp *(assert (message (explode$?str)))*. Atunci când se doreşte fragmentarea unui mesaj în părţi mai mici este mai avantajoasă folosirea valorilor multicâmp, decât a şirurilor. S-a văzut mai sus cum .pentru o simplă operaţie de schimbare a unui caracter cu un altul s-au folosit o mulţime de funcţii.

Pentru a putea se realiza o comparaţie între modul în care se lucrează cu şirurile de caractere şi modul în care lucrează cu multicâmp-urile, vom introduce o variantă a aceluiaşi program care realizează decodificarea folosind numai şirurile de caractere. De această dacă nu se vor mai folosi priorităţi sau fapte de control pentru a împărţi problema în mai multe etape (module). Pentru a introduce literele din alfabetul latin şi codurile morse corespunzătoare, se foloseşte o construcţie *deffacts*. Pentru a vedea diferenţele între cele două programe acestea pot fi rulate pas cu pas *(run 1)* şi urmărite faptele ce sunt asertate sau şterse din lista de fapte.

```
(deffacts code_morse
        (code A "*-") (code B "-***")  (code C "-*-*")  (code D "-**")  (code E "*")
        (code F "**-*")(code G "--*")  (code H "****")  (code I "**")    (code J "*---")
        (code K "-*-") (code L "*-**") (code M "--")    (code N "-*")   (code O "---")
        (code P "*--*")(code Q "--*-") (code R "*-*")   (code S "***")  (code T "-")
        (code U "**-") (code V "***-") (code W "*--")   (code X "-**-")  (code Y "-*--")
        (code Z "--**")
)

(defrule get-message
        =>
        (printout t t t t "       The Morse Code" t t )
        (printout t "A *-  | B -*** | C -*-* | D -**  | E *    | F **-* | G --*  |" t)
        (printout t "H **** | I **   | J *--- | K -*-  | L *-** | M --   | N -*   |" t)
        (printout t "O ---  | P *--* | Q --*- | R *-*  | S ***  | T -    | U **-  |" t)
        (printout t "V ***- | W *--  | X -**- | Y -*-- | Z --** |" t t)
        (printout t "For example the message  "***/---/***"  is  S.O.S." t t)
        (printout t "Enter a message (<CR> to end) :")
        (assert (input (readline)))
)
```

```
(defrule verify_message
        ?f <- (input ?mesg)
        =>
        (loop-for-count (?i 1 (str-length ?mesg)) do
                (bind ?char (sub-string ?i ?i ?mesg))
                (if (not (or (eq (str-compare ?char  * ) 0)
                             (eq (str-compare ?char  - ) 0)
                             (eq (str-compare ?char  / ) 0)))
                 then (printout t "I don‘t recognize \""  ?char  "\" !" t)
                      (assert (error))  (return) ) )
        (printout t "The message is : ")
        (retract ?f) (assert (message ?mesg)) )

(defrule get-letter-code
        (not (error))
        (not (letter ?))
        ?f <- (message ?mesg)
        =>
        (retract ?f)
        (bind ?poz (str-index / ?mesg))   (bind ?len (str-length ?mesg))
        (if  ?poz
         then (if (> ?poz 1)
                 then (assert (letter (sub-string 1 (- ?poz 1) ?mesg))))
                (if (neq ?poz ?len)
                 then (assert (message (sub-string (+ ?poz 1)  ?len  ?mesg))))
        else (assert (letter ?mesg)) ) )

(defrule valid-code-letter
        ?letter <- (letter ?code_letter)
        (code ?l ?code_letter)
        =>
        (retract ?letter)  (printout t ?l) )                    ;afisare litera

(defrule invalid-code-letter
        ?letter <- (letter ?code_letter)
        (not (code ?l ?code_letter))
        =>
        (printout t t t "I don’t recognaize ”' ?code_letter "’ !" t)
        (retract ?letter )  (assert (error)) )

(defrule message-error
        (error)
        =>
        (printout t t "Invalid message..." t) )
```

Diferențele principale dintre cele două variante, exceptând faptul că una din variante lucrează numai cu şiruri de caractere, constau în modul în care este prelucrat textul codificat. Dacă în prima variantă se modifica tot mesajul o dată, în cea dea doua variantă se extrage şi se decodifică fiecare literă una după alta. După afişarea unei litere în cazul în care nu s-a ajuns la sfârşitul mesajului, sau altfel spus dacă mai există faptul *message*, se reiau operațiile de extragere, validare,

decodificare şi afişare pentru următoarea literă. Faptul *(error)* este cel care va permite sau nu extragerea următoarei litere şi deci cel care va opri, dacă este cazul, execuţia programului.

Testele care se fac în regula *verify-message* asupra fiecărui caracter sunt realizate cu ajutorul funcţiei *str-compare*, care returnează 0 în caz de egalitate între şirurile sau simbolurile comparate, -1 dacă primul este mai mic decât al doilea şi 1 dacă primul şir este mai mare decât cel de-al doilea şir. Instrucţiunile următoare

```
(bind ?char (sub-string ?i ?i ?mesg))
(if (not (or (eq (str-compare ?char *) 0)
             (eq (str-compare ?char -) 0)
             (eq (str-compare ?char /) 0)))
```

putea fi scrise sub o formă mai simplă folosindu-ne de faptul că în CLIPS se foloseşte forma prefixată şi o funcţie acceptă de cele mai multe ori mai mulţi parametri, cum este cazul şi funcţiei *neq*:

```
(if (neq (sub-string ?i ?i ?str)  "*"  "-"  "/" " ")
```

Funcţia *str-compare* a fost folosită în acest caz pentru a se evidenţia faptul că aceasta poate accepta ca argumentele şiruri şi simboluri în acelaşi timp (ca şi în cazul de faţă când se compară un şir (caracterul returnat de funcţia sub-string) şi un simbol (* fără ghilimele). În multe situaţii folosirea acestei funcţii poate elimina erorile ce ar fi rezultat în urma unor astfel de comparaţii ilegale.

Trebuie precizat şi faptul că dacă în prima variantă se puteau introduce spaţii între litere şi chiar între simboluri (spaţiile suplimentare erau eliminate), în această variantă introducerea unui singur spaţiu duce la obţinerea unui mesaj invalid. Devine necesară în aceste condiţii folosirea aceluiaşi artificiu. Astfel înlocuirea în funcţia *get-message* a liniei în care se introduce mesajul cu linia următoare conduce la aceiaşi flexibilitate în folosirea unor spaţii suplimentare.

```
(assert (input (str-cat (expand$ (explode$ ?(readline) ) ) ) ) )
```

Se permite introducerea în mesaj a mai multor caractere '\' consecutive, deoarece prin testul *(if (> ?poz 1) then ...)* se asigură faptul că nu se vor aserta fapte de genul *(letter "/")*.

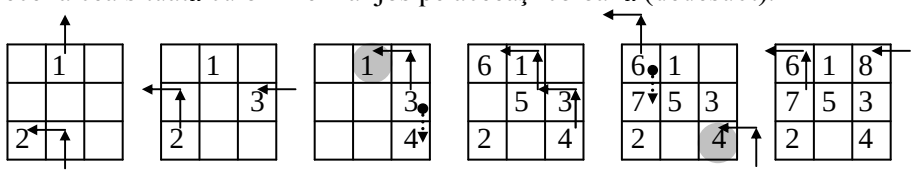
2. **Problema pătratului magic** – Să se scrie un program care creează un pătrat magic de dimensiune prestabilită. Un pătrat magic de latură N, are proprietatea că include în locaţiile sale toate numerele întregi din intervalul $1\text{-}N^2$ şi că sumele numerelor de pe fiecare linie, fiecare coloană sau fiecare diagonală sunt egale. De exemplu

6	1	8
7	5	3
2	9	4

suma = 15

Pentru crearea unui pătrat magic (de dimensiune impară) se poate folosi metoda lui N. Coxeter. Astfel se porneşte cu mijlocul primei linii, apoi se merge la

fiecare pas cu o linie în sus şi cu o coloană la stânga, scriind numerele în ordine crescătoare în locațiile libere. De la prima linie se va trece la ultima, din prima coloană în ultima. Dacă o locație ce ar urma în acest mod este deja ocupată, se trece la cea situată cu o linie mai jos pe aceeaşi coloană (dedesubt).




```
(deftemplate patrat
        (slot linie (type INTEGER))
        (slot coloana (type INTEGER))
        (slot valoare (type INTEGER)))
(deftemplate verifica
        (slot linie (type INTEGER))
        (slot coloana (type INTEGER)))
(deftemplate afisare
        (slot linie (type INTEGER))
        (slot coloana (type INTEGER)))
 (defrule dimensiune-patrat
        =>
        (printout t crlf "    Introduceti dimensiunea patratului magic " crlf)
        (printout t "Trebuie tinut cont de faptul ca nu poate fi decit impara" crlf)
        (printout t "Dimensiune (n>=3,impar) :")
        (while (or (not (integerp (bind ?n (read)) )) (not (oddp ?n)) (< ?n 3))
                (printout t "Dimensiune (n>=3,impar) :") )
        (assert (dimensiune ?n)) )
; Se incepe din prima linie pe coloana din mijloc  linie=1  si coloana= (n+1)/2 .
(defrule punct-de-inceput
        (dimensiune ?n)
        =>
        (assert (patrat (linie 1) (coloana (integer (/ (+ ?n 1) 2))) (valoare 1))
                (contor 1)) )
(defrule patrat_magic
        (dimensiune ?n)
        (contor ?nr &: (< ?nr (* ?n ?n)))
        (patrat (linie ?l) (coloana ?c) (valoare ?nr))
        =>
        (bind ?l (- ?l 1))
        (bind ?c (- ?c 1))
        (if (<= ?l 0)  then (bind ?l ?n))                    ;se trece la ultima linie
        (if (<= ?c 0)  then (bind ?c ?n))                    ;se trece la ultima coloana
        (assert (verifica (linie ?l) (coloana ?c))) )
```

```
(defrule exista_element
        ?f <- (verifica (linie ?l) (coloana ?c))
        (patrat (linie ?l) (coloana ?c))
        =>
        (retract ?f)
        (assert (exista)) )

(defrule daca_locul_NU_e_ocupat
        ?f1 <- (verifica (linie ?l) (coloana ?c))
        ?f2 <- (contor ?nr)
        (not (exists (patrat (linie ?l) (coloana ?c))))
        =>
        (retract ?f1 ?f2)
        (assert (patrat (linie ?l) (coloana ?c) (valoare (+ ?nr 1)))
                (contor (+ ?nr 1))) )

(defrule daca_locul_e_ocupat
        ?f <- (exista)
        (dimensiune ?n)
        ?val <- (contor ?nr)
        (patrat (linie ?l) (coloana ?c) (valoare ?nr))
        =>
        (retract ?f ?val)
        (if (> (+ ?l 1) ?n)
         then (assert (patrat (linie 1) (coloana ?c) (valoare (+ ?nr 1))))
         else (assert (patrat (linie (+ ?l 1)) (coloana ?c) (valoare (+ ?nr 1)))) )
        (assert (contor (+ ?nr 1))) )

(defrule sfirsit
        (dimensiune ?n)
        (contor =(* ?n ?n))
        =>
        (printout t crlf "   Patrat magic :" crlf)
        (assert (afisare (linie 1) (coloana 1))) )

 (defrule afisare-rezultate
        (afisare (linie ?l) (coloana ?c))
        (patrat (linie ?l) (coloana ?c) (valoare ?v))
        (dimensiune ?n)
        =>
        (format t "%4d" ?v)
        (if (or (neq ?l ?n) (neq ?c ?n))
         then (if (eq ?c ?n)
                then (assert (afisare (linie (+ ?l 1)) (coloana 1)))
                     (printout t crlf)
                else (assert (afisare (linie ?l) (coloana (+ ?c 1)))) )
         else (printout t crlf)) )
```

După ce a fost definit cu ajutorul construcţiei *deftemplate* şablonul pentru definirea elementelor matricii (prin fapte compuse caracterizate prin sloturile *linie*, *coloană* şi *valoare*) şi după ce s-a stabilit punctul de start, regula *patrat_magic* va

furniza rând pe rând pozițiile următoarelor elemente, atât timp cât nu s-a ajuns la ultimul element. Faptul *contor* se va incrementa odată cu adăugarea fiecărui element nou în matrice. Dacă poziția noului element este deja ocupată, atunci se va ocupa câmpul aflat dedesubtul elementului curent (conform algoritmului descris mai sus). De remarcat faptul că în regula *sfârşit* se verifică cu ajutorului câmpului condițional valoare returnată = , dacă s-a ajuns la ultimul element: *(contor =(* ?n ?n))* şi se trece la afişarea matricii.

Regula *afisare-rezultate* realizează afişarea matricială fără să folosească bucle for imbricate, folosindu-se doar de un fapt prin care specifică următorul element de afişat *(afisare (linie ?l) (coloana ?c))*. În plus se foloseşte pentru realizarea unei afişări formatate funcția *(format t "%4d" ?v)*, fiecare număr întreg corespunzător unui element al matricii fiindu-i rezervați un număr de 4 spații (caractere). Prin algoritmul de rezolvare al problemei putem afirma că problema este mult mai apropiată de programarea procedurală, decât de cea logică.

Din acest motiv pentru a încerca implementarea unei variante logice, ar trebui să gândim problema bazându-ne doar pe informațiile strict generale, presupunând că nu cunoaştem nici un algoritm care să furnizeze vreo soluție. Din datele problemei se ştie că matricea este pătratică de N linii şi N coloane, elementele sale fiind toți întregii dintre 1 şi N^2. Cunoscând faptul că suma numerelor de pe fiecare din cele N linii este egală şi că suma tuturor elementelor este: $1+2+3+\ldots+ N^2 = N^2*(N^2+1)/2$,se obține faptul că suma valorilor elementelor de pe o linie, coloană sau o diagonală este de **$N*(N^2+1)/2$.**

În aceste condiții o persoană s-ar apuca să caute acele perechi a căror sumă dă rezultatul de mai sus şi ar încerca aranjeze toate numerele din aceste perechi pe linii şi pe coloane în aşa fel încât fiecare număr să apară o singură dată. Se poate presupune faptul că în valoarea din centrul pătratului va fi egală cu media tuturor elementelor. Se va studia în continuare cazul pătratului magic cu latura de 3.

```
(defrule group3x3
        =>
        (bind ?suma (integer (/ (* 3 (+ 1 (* 3 3))) 2)) )                    ;suma=15
        (loop-for-count (?i 1 9)
                (loop-for-count (?j (+ ?i 1) 9)
                        (if (> (+ ?i ?j) ?suma) then (break))
                        (loop-for-count (?k (+ ?j 1) 9)
                                (if (> (+ ?i ?j ?k) ?suma) then (break))
                                (if (= (+ ?i ?j ?k) ?suma)
                                 then (assert (group ?i ?j ?k ))
)       )       )       )       )
```

În Regula de mai sus se asertează faptele *group*, ce au proprietatea că au elementele distincte, în ordine strict crescătoare şi că suma acestora este egală cu 15. Toate acestea se pot observa din indecşii buclelor imbricate (*j* porneşte de la *i+1*, iar *k* de la *j+1*). După execuția regulii faptele asertate în listă sunt următoarele:

```
(group 1 5 9)   (group 2 4 9)   (group 3 4 8)   (group 4 5 6)
(group 1 6 8)   (group 2 5 8)   (group 3 5 7)
                (group 2 6 7)
```

Se numără combinaţiile ce conţin elementul 1 şi aşa mai departe până la elementul al 9-lea şi se observă faptul că într-adevăr elementul 5 (media tuturor valorilor) apare în cele mai multe fapte *group*. Ştiind că într-un pătrat magic suma elementelor de pe o linie, coloană sau diagonală este aceiaşi, se calculează pentru fiecare locaţie din matrice numărul de grupări cu proprietatea enunţată mai sus, ce trec prin acea locaţie.

3	2	3
2	4	2
3	2	3

Elemente:	1	2	3	4	5	6	7	8	9
Nr. fapte group:	2	3	2	3	4	3	2	3	2

Realizând o corespondenţă între tabela elementelor şi pătratul magic se observă cum poziţia centrală poate fi ocupată doar de elementul 5, colţurile doar de elementele {2, 4, 6, 8} şi mijloacele de {1, 3, 7, 9}. Aşezând elementele în matrice pe poziţiile pe care le pot ocupa se obţine pătratul magic rezultat şi în varianta anterioară, precum şi alte 7 pătrate rezultate din rotiri şi oglindiri. Regula care înglobează toate aceste cunoştinţe este foarte simplă:

```
(defrule permută
        (group  $?first  ?n1  $?middle  ?n2  $?last)
        (not (group  $?first  ?n2  $?middle  ?n1  $?last))
        =>
        (assert (group  $?first  ?n2  $?middle  ?n1  $?last)) )

(defrule magic3x3
        (group ?n1&2|4|6|8 ?n2&1|3|7|9 ?n3)
        (group ?n4&1|3|7|9 ?n5&5 ?n6)
        (group ?n7&2|4|6|8 ?n8&1|3|7|9 ?n9)
        (group ?n1 ?n4 ?n7)  (group ?n2 ?n5 ?n8)  (group ?n3 ?n6 ?n9)   ;coloanele
        (group ?n1 ?n5 ?n9) (group ?n3 ?n5 ?n7)                         ;diagonalele
         =>
        (printout t ?n1 ?n2 ?n3 t ?n4 ?n5 ?n6 t ?n7 ?n8 ?n9 t t) )
```

Regula *permută* după cum sugerează şi numele asertează restul faptelor *group* prin permutarea elementelor. Singura condiţie este aceea ca noul fapt să nu existe deja în lista de fapte. Pentru a găsi toate cele 8 soluţii regula *magic3x3* poate păstra numai pattern-ul primei sau ultimei coloane, sau poate elimina pattern-urile diagonalelor (în acest caz sunt necesare toate pattern-urile coloanelor). Pattern-urile corespunzătoare diagonalelor se pot elimina şi prin adăugarea unui test suplimentar:

(test (and (neq ?n4 ?n2) (neq ?n6 ?n2) (neq ?n8 ?n6 ?n4)))

Regula poate ajunge astfel la un număr minim de 4 pattern-uri:

```
(defrule magic3x3
        (group ?n1&2|4|6|8 ?n2&1|3|7|9 ?n3)                    ;pattern 1
        (group ?n4&1|3|7|9 ?n5&5 ?n6)                          ;pattern 2
        (group ?n7&2|4|6|8 ?n8&1|3|7|9 ?n9)                    ;pattern 3
        (group ?n1 ?n4 ?n7)                                    ;pattern 4
        (test (and (neq ?n4 ?n2) (neq ?n6  ?n2) (neq ?n8 ?n6 ?n4) ))
         =>  (printout t ?n1 ?n2 ?n3 t ?n4 ?n5 ?n6 t ?n7 ?n8 ?n9 t t) )
```

Pentru a observa diferența de timp obținută la găsirea soluțiilor între regula de mai sus, vom introduce şi o variantă a regulii *magic3x3* în care se presupune a nu se şti faptul că elementul central este 5, în colțuri pot fi puse valorile {2, 4, 6, 8}....

```
(defrule magic3x3
        (group ?n1 ?n2 ?n3)
        (group ?n4&~?n2 ?n5 ?n6&~?n2)
        (group ?n7&~?n3 ?n8&~?n4&~?n6 ?n9)
        (group ?n1 ?n4 ?n7)  (group ?n2 ?n5 ?n8)  (group ?n3 ?n6 ?n9)  ;coloane
        (group ?n1 ?n5 ?n9) (group ?n3 ?n5 ?n7)                        ;diagonalele
         =>
        (printout t ?n1 ?n2 ?n3 t ?n4 ?n5 ?n6 t ?n7 ?n8 ?n9 t t) )
```

Dacă pentru un pătratului magic de latură 3 este posibilă o abordare logică a problemei, odată cu creşterea dimensiunii matricei numărul faptelor creşte atât de mult încât motorului de inferență îi este necesar un timp foarte mare pentru găsirea soluțiilor (de cele mai multe ori CLIPS-ul se blochează). Dacă s-ar realiza o regulă *group5x5* numărul faptelor asertate ar fi de 1393. După rularea regulii de permutare numărul acestora ar creşte de 5! ori şi s-ar trebui să se ajungă la un număr de 167160 fapte. CLIPS-ul fie nu va reuşi să ruleze regula *permută*, fie timpul necesar pentru a realiza aceasta va fi foarte mare.

```
(defrule group5x5
    =>
    (bind ?suma (integer (/ (* 5 (+ 1 (* 5 5))) 2)) )
    (loop-for-count (?i 1 25)
        (loop-for-count (?j (+ ?i 1) 25)
            (loop-for-count (?k (+ ?j 1) 25)
                (if (> (+ ?i ?j ?k) ?suma) then (break))
                (loop-for-count (?l (+ ?k 1) 25)
                    (if (> (+ ?i ?j ?k ?l) ?suma) then (break))
                    (loop-for-count (?m (+ ?l 1) 25)
                        (if (> (+ ?i ?j ?k ?l ?m) ?suma) then (break))
                        (if (= (+ ?i ?j ?k ?l ?m) ?suma)
                         then (assert (group ?i ?j ?k ?l ?m))
)   )   )   )   )   )   )
```

O soluție pentru a reuşi permutarea faptelor *group* este să ne folosim de proprietatea de unicitate a faptelor (conform acesteia nu pot fi asertate două fapte identice în lista faptelor). Dacă nu s-a dat anterior comanda **set-fact-duplication** atunci se poate elimina din regula permută testul de existență:

```
(defrule permută
        (group  $?first  ?n1  $?middle  ?n2  $?last)
        =>
        (assert (group  $?first  ?n2  $?middle  ?n1  $?last)) )
```