CAP.5 Controlul execuţiei regulilor

5.1. Atributele Deftemplate

In momentul definirii unei construcţii *deftemplate* CLIPS-ul oferă posibilitatea specificării pentru fiecare slot a unor *atribute* şi astfel se pot defini restricţii foarte puternice, specificându-se tipurile de date primare şi chiar valorile pe care le poate lua un slot. Pentru valorile numerice se poate specifica un rang şi pentru multisloturi numărul maxim şi numărul minim de câmpuri. Atributul *default* oferă o valoare implicită, ce va fi folosită dacă nu se specifică o valoare explicită. Aceste atribute din pattern-urile din LHS-ul regulii sunt analizate pentru a se stabili dacă constrângerea specificată împiedică sau nu regula să se aprindă. Există două tipuri de restricţii :

- *statice*
- *dinamice*.

Dacă restricţiile sunt statice construcţiile sunt analizate gramatical, fiind verificate restricţiile doar în momentul în care funcţia este apelată. Dacă restricţiile sunt dinamice verificarea acestora se realizează pentru fiecare dată sau obiect nou creat, altfel spus restricţiile statice sunt verificate la încărcarea unui program, iar cele dinamice în momentul când acesta se execută. In mod implicit sunt activate cele statice şi dezactivate cele dinamice. Pentru activarea sau dezactivarea uneia dintre acestea se pot folosi funcţiile **set-static-constraint-checking** şi **set-dynamic-constraint-checking**. Trebuie reţinut faptul că atunci când sunt folosite restricţiile dinamice, informaţiile legate de atributele asociate construcţiilor nu sunt salvate atunci când se realizează o imagine binară cu ajutorul comenzii **bsave**.

```
<deftemplate-attribute> ::=   <constraint-attribute> |
                              <default-attribute>

<constraint-attribute>  ::=   <type-attribute> |
                              <allowed-constant-attribute> |
                              <range-attribute> |
                              <cardinality-attribute>

<default-attribute>     ::=   <default> |
                              <default dynamic>
```

5.1.1. Atributul TYPE

Atributul **type** defineşte tipurile de date primare care pot fi plasate într-un slot. Formatul general al acesteia este *(type <type-specification>)* ,unde *<type-specification>* poate fi unul (sau mai multe) dintre simbolurile SYMBOL, STRING, LEXEME, INTEGER, FLOAT, NUMBER, INSTANCE-NAME, INSTANCE-ADDRESS, INSTANCE, FACT-ADDRESS, EXTERNAL-ADDRESS, sau poate fi ?VARIABLE. In ultimul caz, atunci când se utilizează cuvântul ?VARIABLE se poate folosi orice tip

de dată. Folosirea lui NUMBER pentru acest atribut este echivalentă cu folosirea lui INTEGER şi FLOAT; a lui LEXEME este echivalentă cu STRING şi FLOAT; iar a lui INSTANCE este echivalentă cu a lui INSTANCE-NAME şi INSTANCE-ADDRESS. De exemplu pentru o adresă se pot restricţiona valorile pe care le ia slotul *strada* să fie simboluri şi cele ale slotului *număr* să fie întregi.

```
(deftemplate adresa
        (multislot strada (type SYMBOL))
        (slot numar (type INTEGER))  )
```

Din momentul în care o astfel de construcţie deftemplate a fost definită, CLIPS-ul va forţa automat fiecare fapt deftemplate introdus să respecte restricţiile date de slotul atribut. De exemplu încercând introducerea unui şir în locul unui simbol s-ar obţine:

```
CLIPS> (assert (adresa  (strada "Alexandru cel Bun")
                        (numar 47)))
```
[CSTRNCHK1] A literal slot value found in the assert command does not match the allowed types for the slot strada.

In plus CLIPS-ul face verificarea şi pentru o variabilă în LHS-ul unui reguli. Astfel dacă avem o construcţie deftemplate *zi_saptămână* şi tipurile de date primare permise pentru sloturile *număr* din cele două deftemplate-uri nu se potrivesc, se va genera o eroare pentru o regulă în care se va realiza compararea directă a două astfel de sloturi. Acest lucru poate fi observat în exemplul următor:

```
(deftemplate zi_saptamana
        (slot numar (type STRING))  )

CLIPS>(defrule traseu_postas
        ?f <- (adresa (strada ?nume) (numar ?no))
        (zi_saptamana (numar ?no))
        =>
        (retract ?f)
        (printout  t  "Distribuit posta pe strada " ?nume crlf) )
```
[RULECSTR1] Variable ?no in CE#2 slot numar has constraint conflicts whitch make the pattern unmatchable.

In mod evident eroarea apare din cauză că slotul *număr* al faptului *zi-săptămână* trebuie să fie un şir de caractere, iar slotul *număr* al faptului adresa trebuie să fie un întreg. Nu este posibil ca variabila *?no* să satisfacă ambele restricţii din LHS-ul regulii. Acelaşi tip de eroare s-ar fi obţinut şi pentru o regula în care se încearcă în RHS-ul regulii o operaţie de adunare în care unul dintre argumente este un şir de caractere şi nu un număr aşa cum cere această funcţie:

```
(defrule ziua_de_maine
        (zi_saptamana (slot ?no))
        =>
        (printout  t  "Ziua de maine este : "  (+ ?no 1) crlf) )
```

5.1.2. Atributul allowed values

Împreună cu atributul *type* care restricționează tipul de dată se poate specifica şi o listă de valori pentru un anumit slot. Formatul acestui atribut este:

<allowed-constant-attribute> ::=

(allowed-lexemes <lexeme-list>) | (allowed-symbols <symbol-list>) |
(allowed-strings <string-list>) | (allowed-integers <integer-list>) |
(allowed-numbers <number-list>) | (allowed-floats <float-list>) |
(allowed-instance-names <instance-list>) | (allowed-values <value-list>)

<lexeme-list> ::= <lexeme>+ | ?VARIABLE
<symbol-list> ::= <symbol>+ | ?VARIABLE
<string-list> ::= <string>+ | ?VARIABLE
<number-list> ::= <number>+ | ?VARIABLE
<integer-list> ::= <integer>+ | ?VARIABLE
<float-list> ::= <float>+ | ?VARIABLE
<instance-name-list> := <instance-name>+ | ?VARIABLE
<value-list> ::= <constant>+ | ?VARIABLE

Dacă s-a folosit cuvântul ?VARIABLE atunci orice constantă de tipul specificat este permisă. Folosirea atributului *allowed-lexemes* este echivalentă cu specificarea restricțiilor de constantă simbol sau constantă şir. Conversia din simbol în şir sau din şir în simbol însă nu este realizată. In mod similar folosirea atributului *allowed-numbers* este echivalentă cu specificarea restricțiilor de constantă întreg sau flotantă. De această dată însă se realizează conversia din întreg în flotant şi respectiv din flotant în întreg (din acest motiv folosirea atributului *allowed-numbers* nu este echivalentă cu folosirea atributelor *allowed-integers* şi *allowed-floats* împreună). De exemplu în deftemplate-ul următor :

```
(deftemplate culori_primare
        (slot  culoare    (type SYMBOL)
                          (allowed-symbols galben rosu albastru) ) )
```

atributul *allowed-symbols* **nu** restricționează tipurile de date pe care le ia un slot. Dacă de exemplu acesta ar fi folosit fără atributul *type* ,atunci orice şir, întreg sau flotant care ar fi fost introdus, ar fi fost valid. Din acest motiv este recomandată folosirea acestor două atribute împreună. Se poate însă folosi şi atributul *allowed-values* care restricționează complet folosirea pentru un slot a listei specificate. Chiar dacă în lista de valori permisă sunt numai simboluri, după cum s-a explicat anterior **nu** este echivalentă în acest caz folosirea atributului *allowed-symbols* cu cea a atributului *allowed-values*.

```
(deftemplate culori_primare
        (slot culoare  (allowed-values galben rosu albastru) ) )
```

5.1.3. Atributul RANGE

Acest atribut permite specificarea valorii minime şi a valorii maxime pentru un câmp numeric. Formatul general al atributului *range* este următorul:

(range <lower-limit> <upper-limit>)

,unde <*lower-limit*> şi <*upper-limit*> specifică valoarea minimă şi respectiv valoarea maximă pentru un slot, şi acestea pot lua o valoare numerică sau cuvântul ?VARIABLE, atunci când se doreşte ca orice valoare să fie validă pentru câmpul astfel specificat. Dacă ?VARIABLE se foloseşte pentru limita inferioară, atunci valoarea minimă este minus infinit **(_-)**. Dacă ?VARIABLE se foloseşte pentru limita superioară, atunci valoarea maximă este plus infinit **(_+)**.

Atributul *range* nu poate fi folosit în conjuncţie cu atributele *allowed-values* , *allowed-integers* , *allowed-floats* , *allowed-numbers*. Dacă este necesară realizarea unei conversii temporare din întreg în flotant pentru realizarea unei comparaţii cu rangurile specificate în atribut, atunci aceasta va fi efectuată.

De exemplu slotul *număr* din construcţia deftemplate *adresa* poate fi modificat pentru a se evita valorile nule sau negative după cum urmează:

```
(deftemplate adresa
        (multislot strada (type SYMBOL))
        (slot numar (type INTEGER)
                    (range 1 ?VARIABLE) ) )
```

La fel ca şi în cazul atributului *allowed-value*, atributul *range* nu restricţionează tipul unei date să fie de tip integer sau float. Astfel dacă am şterge din construcţia de mai sus pe *(type INTEGER)* ,orice dată nenumerică introdusă este validă, în schimb pentru fiecare dată numerică introdusă verificându-se dacă aceasta se află în intervalul impus de atributul *range*. Valorile implicite pentru limitele acestui atribut sunt:

(range ?VARIABLE ?VARIABLE)

5.1.4. Atributul CARDINALITY

Se foloseşte într-un multislot pentru a specifica numărul maxim şi respectiv numărul minim de câmpuri pe care acesta le poate lua:

(cardinality <lower-limit> <upper-limit>)

De această dată limitele pot fi specificate doar de numere întregi pozitive (nu şi de numere flotante cum era posibil la atributul *range*). Dacă este folosit cuvântul ?VARIABLE pentru <*lower-limit*> ,atunci valoarea minimă este zero. Dacă este folosit cuvântul ?VARIABLE pentru <*upper-limit*> ,atunci valoarea maximă este plus infinit(_+). Valorile implicite ale limitei acestui atribut sunt:

(cardinality ?VARIABLE ?VARIABLE)

De exemplu următoarea construcție deftemplate descrie o echipă de volei care are trebuie să aibă şase jucători şi eventual două rezerve. De reținut că un atribut *range* sau un atribut *allowed-values* se aplică pentru toate valorile conținute de un multislot.

```
(deftemplate echipa_de_volei
        (multislot nume (type SIMBOL)
        (multislot jucatori (type STRING)
                        (cardinality 6 6)
        (multislot rezerve (type STRING)
                        (cardinality 0 2) ) )
```

5.2. Atributul DEFAULT

De obicei faptele deftemplate sunt asertate având valori explicite pentru fiecare slot. De multe ori însă poate fi convenabilă atribuirea unei anumite valori pentru un slot, pentru care nu s-a dat nici o valoare explicită în comanda *assert*. Atributul *default* permite realizarea acestui lucru, formatul general al acestuia este:

```
        (default  <default-specification>)
<default-specification>  ::=  ?NONE | ?DERIVE  | <expression> * )
```

,unde <*default-specification*> poate fi o singură expresie (pentru un slot simplu), zero sau mai multe expresii (pentru un slot multicâmp), ?DERIVE sau ?NONE.

Dacă ?DERIVE este specificat în atributul *default* ,atunci se obține o valoare pentru slotul care satisface toate sloturile atribute. Dacă nu avem specificat un atribut *default* pentru un slot, atunci acesta este presupus a fi *(default* ?DERIVE*)*. Pentru un slot singur-câmp este selectată o valoare care să satisfacă toate atributele *type* , *range*, *allowed* ale slotului. Pentru un slot multicâmp se va obține însă o listă de valori identice care sunt minimum numărului permis de atributul *cardinality* al slotului (zero dacă acesta nu este specificat). Dacă vom avea una sau mai multe valori *default* pentru slotul multicâmp atunci acestea trebuie de asemeni să satisfacă atributele *type* , *range*, *allowed* ale slotului. Un exemplu pentru a exemplifica cele spuse poate fi:

```
CLIPS>(deftemplate exemplu
        (slot a)
        (slot b (type INTEGER))
        (slot c (allowed-values rosu galben albastru))
        (multislot d)
        (multislot e (cardinality 2 2)  (type FLOAT)  (range 3.5 10.0)) )
CLIPS> (assert (exemplu)                ;nu se specifică nici un slot
<Facts-0>
CLIPS> (facts)
f-0     (exemplu (a nil)  (b 0)  (c rosu)  (d)  (e 3.5 3.5))
For a total of 1 fact.
```

Pentru determinarea valorii *default* se utilizează următoarele reguli (în ordine):

1. Tipurile permise pentru atributul *default*, în ordinea precedenţei lor sunt următoarele: SYMBOL, STRING, INTEGER, FLOAT, INSTANCE-NAME, INSTANCE-ADDRESS, FACT-ADDRESS, EXTERNAL-ADDRESS.

2. Dacă tipul *default* are o restricţie *allowed* specificată (cum ar fi *allowed-integers* pentru tipul INTEGER), atunci prima valoare specificată în atributul *allowed* este aleasă ca valoare *default*.

3. Dacă valoarea *default* nu a fost obţinută la pasul 2 şi avem tipul *default* INTEGER sau FLOAT, fiind specificat atributul *range* ,atunci rangul minim este folosit ca valoare *default* dacă acesta nu este ?VARIABLE, altfel este folosit rangul maxim dacă acesta nu este ?VARIABLE.

4. Dacă valoarea default nu a fost obţinută nici la pasul 2, nici la pasul 3, atunci valoarea *default* este **nil** pentru tipul SYMBOL, **“”** pentru tipul STRING, **0** pentru tipul INTEGER, **0.0** pentru tipul FLOAT, **[nil]** pentru tipul INSTANCE-NAME, un pointer spre o copie a unei instanţe pentru tipul INSTANCE-ADDRESS, un pointer spre o copie a unui fapt pentru tipul FACT-ADDRESS, şi pointerul **NULL** pentru tipul EXTERNAL-ADDRESS.

5. Odată obţinută valoarea *default* pentru un slot singur-câmp din paşii 1-4, valoarea *default* pentru un slot multicâmp se obţine prin multiplicare de un număr egal cu minimul cardinalului (dacă acesta este mai mare egal cu zero), sau în caz contrar valoarea *default* este egală cu multicâmpul de lungime zero.

Dacă în atributul *default* este specificat ?NONE, atunci pentru acel slot trebuie introdusă obligatoriu o valoare explicită în comanda *assert*. De exemplu:

```
CLIPS> (deftemplate exemplu
                     (slot a)
                     (slot b (default ?NONE) ) )
CLIPS> (assert (exemplu))
[TMPLTRHS1]  Slot b requires a value because of its (default ?NONE) attribute.
CLIPS> (assert (exemplu (b  1) ) )
<Facts-0>
CLIPS> (facts)
f-0     (exemplu (a nil)
                 (b 1) )
For a total of 1 fact.
```

Dacă una sau mai multe expresii sunt utilizate cu atributul *default* , atunci expresia este evaluată în momentul analizării slotului, când acesta nu are specificată nici o valoare explicită în comanda *assert*. Atributul *default* pentru un slot singur-câmp nu trebuie să conţină exact o expresie, pe când pentru un slot multicâmp se pot specifica zero (caz în care valoarea default este multicâmpul de lungime 0) sau mai multe expresii grupate împreună. Exemplul următor utilizează expresii în atributul *default* după cum urmează:

```
CLIPS> (deftemplate exemplu
                (slot a  (default 3))
                (slot b  (default (+ 3 4))
                (multislot c  (default rosu galben albastru))
                (multislot d  (default (+1 2) (+ 3 4) ) ) )
CLIPS> (assert (exemplu) )
<Facts-0>
CLIPS> (facts)
f-0     (exemplu (a 3)  (b 7)
                 (c  rosu galben albastru)  (d 3 7) )
For a total of 1 fact.
```

5.3. Atributul Default-Dynamic

De obicei valoarea *default* specificată prin atributul *default* este determinată în faza de analizare a definiției unui slot. Uneori este posibil să avem generată valoarea *default* în momentul în care faptul care foloseşte această valoare este asertat. Aceasta este dată împreună cu atributul *default-dynamic*. O valoare a unui slot care utilizează un atribut *default-dynamic* rămâne nespecificată într-o comandă assert , până când expresia specificată cu acest atribut este evaluată şi folosită pentru valoarea slotului.

Pentru a exemplifica acest lucru, să considerăm problema ştergerii unor fapte după trecerea unui anumit interval de timp. Pentru început avem nevoie de o posibilitate de a afla când a fost asertat faptul nostru. Funcția **time** returnează numărul de secunde care au trecut de când sistemul dependent arată timpul. Această valoare returnată este utilă atunci când este comparată cu alte valori. Următorul deftemplate conține un slot *creation-time* care va păstra timpul creării faptului, iar slotul valoare păstrează o valoare asociată faptului.

```
(deftemplate data
        (slot  creation-time  (default-dynamic  (time)) )
        (slot  value ) )
```

De fiecare dată când este creat un fapt *data*, pentru slotul *creation-time* nu este specificată o valoarea, păstrându-se valoarea returnată de funcția time.

```
CLIPS> (watch facts)
CLIPS> (assert (data (value 3)) )
==>  f-0   (data (creation-time) (value 3))
<Facts-0>
CLIPS> (assert (data (value b)) )
==>  f-1   (data (creation-time) (value b))
<Facts-1>
```

După asertarea faptului *current-time* ce conține timpul curent al sistemului, putem realiza o regulă care va şterge un faptul *data*, la un minut după asertatarea acestuia. Pentru obținerea faptului *current-time* se poate folosi comanda *assert* :

```
CLIPS>(assert  (current-time (time) ) )

        (defrule retract-data-facts-after-one-minute
                ?f <- (data (creation-time ?t1) )
                (current-time ?t2)
                (test (> (- ?t2 ?t1) 60) )
                =>
                (retract ?f) )
```

Trebuie menţionat faptul că schimbând regula *retract-data-facts-after-one-minute* cu următoarea regulă nu se produc aceleaşi rezultate.

```
        (defrule retract-data-facts-after-one-minute
                ?f <- (data (creation-time ?t1) )
                (test (> (- (time) ?t1) 60) )
                =>
                (retract ?f) )
```

Funcţia *time* din elementul condiţional test va fi verificată doar când primul pattern va fi potrivit cu faptul *data*. Dacă valoarea returnată de funcţia *time* nu va fi mai mare decât cel din faptul *data* cu 60 secunde (acest lucru presupune introducerea regulii după un minut de la asertarea faptului *data*), CLIPS-ul nu va continua reverificarea *CE test* pentru a determina dacă acesta este evaluat altfel şi regula nu se va mai aprinde (programul nu are nici un efect). Deoarece verificarea unui *CE test* nu se realizează decât o singură dată trebuie găsită o altă soluţie în cazul în care se doreşte folosirea directă a funcţiei *time*.

5.4. Verificarea restricţiilor statice şi dinamice

După cum s-a mai spus CLIPS-ul oferă două nivele de verificare a restricţiilor, primul nivel cel de verificare a restricţiilor statice realizându-se în momentul analizării unei expresii sau a unei construcţii (în momentul încărcării unui program), al doilea nivel cel de verificare a restricţiilor dinamice realizându-se în momentul creării unui nou obiect sau a unei noi date (în momentul execuţiei unui program sau comenzi). Pentru a determina nivelul actual se pot folosi funcţiile **get-static-constraint-checking** (returnează TRUE dacă nivelul actual de verificare este cel al restricţiilor statice) şi **get-dynamic-constraint-checking** (returnează simbolul TRUE dacă nivelul actual de verificare este cel al restricţiilor dinamice şi FALSE în caz contrar). Pentru a schimba nivelul actual se poate folosi una dintre funcţiile **set-static-constraint-checking** şi **set-dinamic-constraint-checking** (pentru argumentul TRUE activează nivelul de verificare al restricţiilor statice, iar pentru argumentul FALSE dezactivează nivelul de verificare al restricţiilor statice).

Uneori nu este posibilă detectarea unei erori în momentul analizării gramaticale. Astfel pentru regula următoare *create-person* (corectă din punct de vedere sintactic) se pot lega de variabilele *?age* şi *?name* valori ilegale (utilizatorul poate introduce de la tastatură datele de intrare corect sau nu).

```
(deftemplate person
        (multislot name (type SYMBOL))
        (slot age (type INTEGER)) )

(defrule create-person
        =>
        (printout  t  "What is your name? ")
        (bind ?name (explode$ (readline))
        (printout  t  "What is your age? ")
        (bind ?age (read))
        (assert (person (name ?name) (age ?age))) )
```

Funcția *readline* este folosită pentru a introduce numele întreg al persoanei ca un şir, iar funcția *explode$* pentru a converti şirul într-o valoare multicâmp, ce va fi plasată în slotul *name*. Deşi datele introduse sunt incorecte şi violează restricțiile impuse construcției deftemplate, noul fapt va fi asertat în lista de fapte fără a se semnala vreo eroare.

```
CLIPS> (reset)
CLIPS> (run)              ; s-a trecut de faza de analizare gramaticală a construcțiilor
What is your name? Costin Ciprian
What is your age? eleven
CLIPS> (facts)
f-0     (initial-fact)
f-1     (person (name Costin Ciprian) (age eleven))
For a total of 2 facts.
```

Pentru aceiaşi regulă şi aceleaşi date de intrare ilegale, dar de această dată pentru nivelul de verificare al restricțiilor dinamice, în momentul creării unui nou fapt deftemplate acesta este verificat, semnalându-se dacă este cazul eroare. Execuția programului este oprită, chiar dacă noul fapt este introdus în lista de fapte.

```
CLIPS> (set-dynamic-constraint-checking TRUE)
FALSE
CLIPS> (reset)
CLIPS> (run)
What is your name? Costin Ciprian
What is your age? eleven
[CSTRNCHK1] Slot value (Costin Ciprian) found in fact f-1 does not match the
allowed types for slot age.
[PRCCODE4] Execution halted during the actions of defrule create-person.
CLIPS> (facts)
f-0     (initial-fact)
f-1     (person (name Costin Ciprian) (age eleven))
For a total of 2 facts.
```

5.5. Prioritatea unei reguli (SALIENCE)

CLIPS-ul pune la dispoziția utilizatorului pentru controlul execuției unui program în afară faptelor de control, alte două tehnici de control diferite:

- prioritatea (salience)
- modulele (controlul modular va fi discutat mai târziu).

Prin folosirea lui **salience** se realizează specificarea în mod explicit a priorităților regulilor. In mod normal ***agenda*** se comportă ca o stivă în care activările cele mai recente ale regulilor sunt plasate în vârful acesteia. Folosindu-se *salience,* regula cu prioritatea cea mai mare va fi plasată prima în *agenda,* ordinea celorlalte regulilor din agenda fiind dată acum şi de prioritatea acestora.

Valoarea numerică explicită a priorității poate fi între –10000 şi 10000, în cazul în care nu se menționează o valoare explicită cea implicită fiind 0 (la mijloc între cea mai mare şi cea mai mică valoare posibilă). Una din utilizările lui *salience* este aceea de a forța regulile să se aprindă într-un model secvențial. Considerăm pentru început următorul set de reguli care nu au declarată nici o prioritate.

```
(defrule fire-first
        (priority first)
        =>
        (printout  t  “Print first” crlf) )

(defrule fire-second
        (priority first)
        =>
        (printout  t  “Print second” crlf) )

(defrule fire-third
        (priority first)
        =>
        (printout  t  “Print third” crlf) )
```

Ordinea în care regulile se aprind este dependentă numai de ordinea în care faptele satisfac CE din LHS-ul regulilor. Se poate observa ordinea activărilor urmărindu-se conținutul agendei. De exemplu dacă regulile sunt deja introduse, atunci pentru următoarele comenzi se va observa următorul efect.

```
CLIPS> (reset)
CLIPS> (assert (priority first))
<Fact-1>
CLIPS> (assert (priority second))
<Fact-2>
CLIPS> (assert (priority third))
<Fact-3>
CLIPS> (run)
Print third.
Print second.
Print first.
```

Motivul activării regulilor în această ordine este faptul că agenda se comportă ca o stivă LIFO (ultimul intrat, primul ieşit). Primul fapt *priority-first* activează regula *fire-first*. Când este asertat al doilea fapt, acesta va activa regula *fire-second*, ce va fi aşezată în vârful stivei şi deci înaintea lui *fire-first*. La fel se va întâmpla şi pentru cea de-a treia regulă *fire-third*, astfel încât în momentul în care se va lansa în execuție programul aceasta va fi şi prima regulă care se va aprinde.

Dacă ordinea asertării faptelor a determinat aprinderea regulilor contrar aşteptărilor inițiale, prin introducerea faptelor în ordine inversă se va obține aprinderea regulilor în ordinea fire-first (vârful stivei), fire-second, fire-third (ultima în stiva).

```
CLIPS> (reset)
CLIPS> (assert (priority third))
<Fact-1>
CLIPS> (assert (priority second))
<Fact-2>
CLIPS> (assert (priority first))
<Fact-3>
CLIPS> (run)
Print third.
Print second.
Print first.
```

Un dezavantaj pentru programator este acela că pentru mai multe reguli de aceiaşi prioritate şi activate de aceleaşi fapte nu este garantată plasarea acestora în agendă într-o anumită ordine (de ex. alfabetică). Folosirea priorităților are ca efect forțarea aprinderii regulilor în ordinea *fire-first, fire-second, fire-third* independent de ordinea asertării faptelor. Declararea priorității se face întotdeauna la începutul unei reguli înaintea elementelor condiționale CE.

```
(defrule fire-first
        (declare (salience 30))
        (priority first)
        =>
        (printout  t  “Print first” crlf) )

(defrule fire-second
        (declare (salience 20))
        (priority first)
        =>
        (printout  t  “Print second” crlf) )

(defrule fire-third
        (declare (salience 10))
        (priority first)
        =>
        (printout  t  “Print third” crlf) )
```

In acest moment ordinea amplasării regulilor în agendă va fi întotdeauna aceiaşi (numai mai este dependentă de ordinea asertării faptelor). După asertarea faptelor *priority* comanda *agenda* ne poate confirma acest lucru:

```
CLIPS> (reset)
CLIPS> (assert  (priority second)  (priority first)  (priority third))
CLIPS> (agenda)
30      fire-first:  f-2
20      fire-second:  f-1
10      fire-third:  f-3
For a total of 3 activations.
```

Revenind la problema în care se cerea ştergerea unui fapt după un anumit timp (60 secunde), prin folosirea priorităţilor se pot elimina faptele de control. Astfel regula *time-upper* ar trebui să fie pe un nivel mai mic de prioritate.

```
(deftemplate data  (slot creation-time (default-dynamic (time)))
                           (slot  value ) )

(defrule verify-time
        (data (creation-time ?t) )
        (test (< (- (time) ?t) 60) )
        =>
        (assert (time lower)) )

(defrule time-lower
        ?f1 <- (time lower)
        ?f2 <- (data (creation-time ?t) (value ?v))
        =>
        (retract ?f1 ?f2)
        (assert (data (creation-time ?t) (value ?v))))

(defrule time-upper
        (not (time lower) )
        ?f <- (data (creation-time ?) )
        =>
        (retract ?f) )
```

Prima regulă de testare timp se reaprinde după ce a doua regulă reasertează faptul *(data (creation-time ?t))*. In lipsa priorităţilor se foloseşte un fapt de control *(time lower)*, ce condiţionează ştergerea faptului *data* în cea de-a treia regulă. Prin folosirea nivelelor de prioritate se pot elimina fapte de control în totalitate.

```
(defrule verify-time
        ?f <- (data  (creation-time ?t&:(< (- (time) ?t) 60))
                     (value ?v))
        =>
        (retract ?f)
        (assert (data (creation-time ?t) (value ?v))) )
```

```
(defrule retract-data-facts
        (declare (salience –10))
        ?f <- (data)
        =>
        (retract ?f) )
```

5.6. Problema becurilor

Intr-un tablou de NxN becuri schimbând starea unui bec (aprins sau stins) se schimbă automat şi starea celor 4 vecini, aflaţi la stânga, la dreapta, deasupra şi dedesubtul acestuia (în cazul în care aceştia există, elementul poate fi situat şi într-un colţ sau pe marginea matricii). Problema cere găsirea unei soluţii minimale, respectiv găsirea numărului minim de becuri a căror stare fiind schimbată, se obţine ca efect aprinderea tuturor becurilor din tablou. Starea iniţială din care se pleacă este cea în care toate becurile sunt stinse (problema poate fi modificată în acest sens, plecându-se de la o stare iniţială în care nu avem toate becurile stinse, odată cu această modificare crescând şi gradul de dificultate al problemei).

Problema de mai jos oferă soluţii doar pentru o matrice 3x3, pentru alte dimensiuni ale matricii fiind necesar un alt algoritm. Se poate observa faptul că aprinderea şi stingerea aceluiaşi bec este inutilă, deoarece se schimbă de două ori a starea vecinilor săi (efectul este nul). Soluţia finală nu va avea deci decât elemente distincte. Pentru matricea de 3x3 soluţia optimă găsită este cea în care sunt aprinse becurile din colţuri şi din mijloc (suma liniei şi a coloanei dă un număr par).

X	x	
x		

x	x	
X	x	

x		
x	**X**	x
x		

x		
x	x	
x	x	**X**

x	x	**X**
x	x	x
x	x	x

1		5
	3	
2		4

Tabloul este implementat cu ajutorul unei construcţii deftemplate *tab* în componenţa căruia avem sloturile *linie*, *coloană* şi *stare* (0-bec stins, 1-bec aprins). Pentru becurile aflate în colţuri sau pe margine numărul şi poziţia vecinilor a căror stare se modifică variază, pentru a nu realiza o regulă pentru fiecare caz în parte se va borda matricea cu două linii şi coloane a căror elemente nu ne interesează.

```
(deftemplate tab
        (slot linie (type INTEGER))
        (slot coloana (type INTEGER))
        (slot stare (allowed-values 1 0)) )

 (defrule date-initiale
        =>
        (printout t "Introduceti dimensiunea tabloului N: ")
        (assert (dimensiune (+ (read) 2))
                (contor 0)) )
```

```
(defrule creare-tablou
        (dimensiune ?d)
        ?f <- (contor ?c&:(< ?c (* ?d ?d)))
        =>
        (retract ?f)
        (assert (tab (linie (integer (/ ?c ?d)))
                     (coloana (mod ?c ?d))
                     (stare 0))
                (contor (+ ?c 1))) )
```

Regula *creare-tablou* se comportă ca o buclă ***for*** de la 1 la NxN (unde N-2 reprezintă dimensiunea reală a tabloului), în care este creat la fiecare activare a regulii câte un element (starea inițială a becurilor este 0). Elementele din prima şi ultima linie şi coloană, care abordează de fapt tabloul real nu ne interesează, existența acestora fiind necesară doar pentru aplicarea aceleiaşi reguli pentru toate elementele tabloului real.

```
(defrule aprindere-bec
        (dimensiune ?d)
        ?f1 <- (tab (linie ?lin &:(> ?lin 0)&:(< ?lin (- ?d 1)) )
                    (coloana ?col &:(> ?col 0)&:(< ?col (- ?d 1))
                                  &:(= (mod (+ ?lin ?col) 2) 0))
                    (stare ?v1&:(<> ?v1 1)))
        ?f2 <- (tab  (linie ?lin)  (coloana =(- ?col 1))  (stare ?v2))
        ?f3 <- (tab  (linie ?lin)  (coloana =(+ ?col 1))  (stare ?v3))
        ?f4 <- (tab  (linie =(- ?lin 1))  (coloana ?col)  (stare ?v4))
        ?f5 <- (tab  (linie =(+ ?lin 1))  (coloana ?col)  (stare ?v5))
        =>
        (modify ?f1 (stare (- 1 ?v1)))          ;se trece prin operatia de (1 – v)
        (modify ?f2 (stare (- 1 ?v2)))          ;din starea de aprins în stins 1 -> 0
        (modify ?f3 (stare (- 1 ?v3)))          ;din starea de stins în aprins 0 -> 1
        (modify ?f4 (stare (- 1 ?v4)))
        (modify ?f5 (stare (- 1 ?v5)))
        (assert (citire-camp 1 1)
                (valori-linie)) )
```

In regula *aprindere-bec* se pune condiția ca becul a cărui stare se modifică să nu se afle pe prima sau ultima linie sau coloană (*>0* sau *<d-1*, unde d este dimensiunea matricii bordate), suma liniei şi a coloanei să fie un număr par (restul împărțirii la 2 să fie 0) şi starea acestuia să fie 0 (bec stins).

```
(defrule stare-finala
        (dimensiune ?d)
        (not (exists (tab (linie ?lin&:(> ?lin 0)&:(< ?lin (- ?d 1)) )
                          (coloana ?col&:(> ?col 0)&:(< ?col (- ?d 1)) )
                          (stare 0)) ))
        =>
        (printout t "S-a ajuns in stare finala!" crlf) )
```

Regula *stare-finală* se aprinde în cazul în care, în tabloul real (mai puțin prima şi ultima linie şi coloană) nu mai există nici un bec stins, deci s-a ajuns în starea în care toate sunt aprinse. Regula are nevoie însă de o prioritate maximă pentru a se aprinde înaintea celorlalte reguli şi în acelaşi timp ea trebuie să încheie execuția programului lucru care nu se realizează în situația de față. Realizarea acestor modificări rămân ca temă pentru cei ce vor implementa o astfel de regulă.

Următoarele trei reguli sunt folosite pentru a afişa starea curentă a tabloului după fiecare modificare apărută. Prima regulă *citire-linie-tablou* va înlocui un bec aprins cu "x" şi respectiv un bec stins cu "." , păstrând aceste rezultate într-un fapt ordonat *valori-linie*. A doua regulă *afişare-linie-tablou* nu face altceva decât să afişeze fiecare linie a tabloului, folosindu-se de faptele ordonate *valori-linie*, până în momentul în care se ajunge la ultima linie a matricii reale. A treia regulă va aştepta apăsarea unei taste pentru continuarea programului .

```
(defrule citire-linie-tablou
        (dimensiune ?d)
        ?f1 <- (citire-camp ?lin ?c&:(< ?col (- ?d 1)) )
        (tab (linie ?l) (coloana ?col) (stare ?v))
        ?f2 <- (valori-linie $?val)
        =>
        (retract ?f1 ?f2)
        (assert (valori-linie $?val   (nth$ (+ ?v 1) (create$ . x)))
                (citire-camp ?l (+ ?col 1)) ) )

(defrule afisare-linie-tablou
        (dimensiune ?d)
        ?f1 <- (citire-camp ?lin&:(< ?lin (- ?d 1)) ?c)
        ?f2 <- (valori-linie $?val)
        =>
        (retract ?f1 ?f2)
        (assert (citire-camp (+ ?lin 1) 1)
                (valori-linie))
        (printout  t  (implode$ $?val)  crlf) )

(defrule pauza-pt-vizualizare
        (dimensiune ?d)
        ?f1 <- (citire-camp ?lin&:(= ?lin (- ?d 1)) ?)
        ?f2 <- (valori-linie $?)
        =>
        (retract ?f1 ?f2)
        (printout t crlf)  (readline)
        (refresh aprindere-bec))
```

Au fost folosite în regulile de afişare o serie de funcții noi şi anume funcțiile multicâmp **create$**, **nth$**, **implode$**. Funcția *create$* formează o valoare multicâmp din toți parametrii pe care îi primeşte, funcția *implode$* transformă un multicâmp într-un şir, iar funcția *nth$* returnează valoarea de pe poziția specificată de primul argument dintr-un multicâmp. In cazul nostru *(nth$(+ ?v 1)(create$. x))*

va returna primul câmp 'x' dacă variabila ?v este 0, şi al doilea câmp '.' dacă ?v este 1 din multicâmpul creat de funcţia *create$*.

Funcţia *implode$* este folosită în cadrul funcţiei *printout* pentru a se realiza afişarea unui şir de caractere şi nu a unui multicâmp, deoarece un multicâmp este afişat între două paranteze (dacă cele două paranteze nu deranjează atunci nu mai este necesară folosirea acestei funcţii). Vor fi prezentate în tabelul de mai jos funcţiile predicative ale CLIPS-ului.

Funcţii multicâmp

(**create$** <expression>*)	Adună toate argumentele împreună pentru a crea o valoare multicâmp.
(**delete$** <multifield > <begin> <end>)	Şterge toate câmpurile specificate între cei doi întregi dintr-o expresie multicâmp.
(**explode$** <string >)	Creează un multicâmp dintr-un şir
(**first$** <multifield >)	Returnează primul câmp dintr-un multicâmp.
(**implode$** <multifield >)	Creează un şir dintr-un multicâmp.
(**insert$** <multifield> <pos> <single-or-multifield-expr.>)	Introduce una sau mai multe valori într-un multicâmp la poziţia specificată.
(**length$** <multifield >)	Returnează numărul de câmpurilor unui multicâmp (lungimea acestuia).
(**member$** <single-field > <multifield >)	Returnează poziţia câmpului într-o expresie multicâmp (FALSE dacă acesta nu este găsit).
(**nth$** <integer> <multifield>)	Returnează câmpul specificat dintr-un multicâmp.
(**replace$** <multifield > <begin> <end> <single-or-multifield-exp>+)	Inlocuieşte valorile specificate dintr-un multicâmp cu un nou set de valori.

5.7. Intrebări

1. Câte tipuri de restricţii există în CLIPS ? Care sunt diferenţele obţinute în urma utilizării acestora. Care sunt funcţiile folosite pentru activarea sau dezactivarea unuia dintre cele două tipuri de restricţii ?
2. Care sunt atributele ce pot fi asociate unui slot ?
3. Ce restricţie are atributul **type** asupra valorilor luate de un slot ? Care sunt simbolurile pe care acesta le poate lua acesta ?
4. De ce atribut este legat de obicei atributul **allowed** ? Care sunt toate formele pe care acesta le poate lua şi care este semnificaţia cuvântului ?VARIABLE când acesta este folosit în atribut ?

5. Care este singurul atribut allowed care restricţionează tipurile de date? Este echivalentă folosirea atributului *allowed-symbols* cu cea a atributului *allowed-values* ? De ce ?
6. Ce semnificaţie au parametrii atributului **range** ? Care este efectul folosirii cuvântului ?VARIABLE în locul unuia (sau ambilor) parametri ai atributului ?
7. Care sunt atributele cu care nu poate fi folosit în conjuncţie atributul *range* ? Acest atribut restricţionează tipurile de date introduse ?
8. Poate fi folosit atributul **cardinality** şi pentru sloturi singur-câmp ? Care este efectul folosirii cuvântului ?VARIABLE în cadrul acestuia ?
9. Pentru ce este utilizat atributul **default** ? Care sunt efectele obţinute la folosirea lui ?NONE sau ?DERIVE ca parametri (într-un multislot şi într-un slot) ?
10. Când trebuie folosit atributul **default-dynamic**, în locul atributului **default** ?
11. Pentru a detecta o eroare în momentul asertării unui fapt sau la încărcarea faptelor cu **reset** ce fel de restricţii trebuiesc folosite (statice sau dinamice) ?
12. Cum se pot realiza mai multe nivele de prioritate, în care regulile sunt executate în funcţie de acest nivel înaintea celorlalte? Unde va fi plasată în *agenda* o regulă care are prioritatea cea mai mare ?
13. Poate fi prioritatea unei reguli dinamică, adică poate fi schimbată pe parcursul execuţiei unui program ?

5.8. Aplicaţii

1. Scrieţi un program în CLIPS, care să adune numere binare şi să afişeze rezultatul operaţiei într-un nou fapt. Faptele iniţiale ar putea fi:

   ```
   (binary  (name A)  (digits  1  0  1  1  1) )
   (binary  (name B)  (digits  1  1  1  1  0) )
   (add-binary  (name1 A)  (name-2 B) )
   ```

   In urma execuţiei programului va fi asertat un nou fapt în lista de fapte:

   ```
   (binary  (name A+B)  (digits 1  0  0  1  0  1) )
   ```

2. Scrieţi un program care să furnizeze în funcţie de tipul de sânge al unui pacient ce are nevoie de o transfuzie posibilii donatori. Programul trebuie să ţină cont de diferitele tipuri de sânge şi de faptul că o grupa O poate primi doar de la grupa O; grupa A de la grupa O şi de la grupa A; grupa B de la grupa O, grupa A şi de la grupa B; iar grupa AB de la oricare alta.
3. Se consideră N persoane (N>0) şi un tabel de dimensiune NxN în care la intersecţia liniei I cu coloana J se află 1 dacă persoana I o cunoaşte pe persoana şi 0 altfel. Dacă se dau două persoane x, y care nu se cunosc între ele se cere:
   a) Numărul minim de persoane (şi care sunt acestea) prin intermediul cărora y poate face cunoştinţă cu x;
   b) Toate posibilităţile ca y să poată face cunoştinţă cu x.
   c) Grupul maximal de influenţă din matrice.

5.9. Probleme rezolvate

1. **Probleme Becurilor** – De această dată nu se mai cere utilizatorului să introducă de la tastatură poziţia fiecărui bec ce se va aprinde. Calculatorul va fi acela care va furniza soluţiile de rezolvare a problemei pentru o tablă de 3x3, 4x4 sau 5x5. Pentru table de dimensiuni mai mari problema poate fi abordată identic sau se poate gândi un algoritm de rezolvare diferit. In această problemă, ca şi în problema damelor de altfel se vede puterea unui limbaj bazat pe reguli, cum este CLIPS-ul de a rezolva o problemă, care într-o abordare tipică unui limbaj procedural s-ar fi realizat recursiv sau cu for-uri imbricate. Practic se enunţă regulile clar şi motorul CLIPS-ului va fi cel care va căuta toate soluţiile.

   In problema noastră, dacă dimensiunea tabloului este diferită de 3, 4 sau 5, faptul asertat de regula *input_dimensiune* nu va aprinde nici o regulă şi programul nu va face nimic.

```
(deffacts date-initiale
        (stare 0)
        (stare 1))

(defrule input_dimensiune
        =>
        (printout t "Introduceti dimensiunea tabloului : " crlf)
        (assert (dimensiune (read))) )

(defrule solutie_3x3
        (dimensiune 3)
        (stare ?s1)  (stare ?s2)  (stare ?s3)  (stare ?s4)  (stare ?s5)
        (stare ?s6)  (stare ?s7)  (stare ?s8)  (stare ?s9)
        (test  (and  (oddp (+ ?s1 ?s2 ?s4))
                     (oddp (+ ?s2 ?s1 ?s3 ?s5))
                     (oddp (+ ?s3 ?s2 ?s6))
                     (oddp (+ ?s4 ?s1 ?s5 ?s7))
                     (oddp (+ ?s5 ?s2 ?s4 ?s6 ?s8))
                     (oddp (+ ?s6 ?s3 ?s5 ?s9))
                     (oddp (+ ?s7 ?s4 ?s8))
                     (oddp (+ ?s8 ?s5 ?s7 ?s9))
                     (oddp (+ ?s9 ?s6 ?s8)) ))
        =>
        (printout t ?s1 ?s2 ?s3 crlf ?s4 ?s5 ?s6 crlf ?s7 ?s8 ?s9 crlf crlf))

(defrule solutie_4x4
       (dimensiune 4)
       (stare ?s1)  (stare ?s2)  (stare ?s3)  (stare ?s4)
       (stare ?s5&:(oddp (+ ?s1 ?s2 ?s5)) )
       (stare ?s6&:(oddp (+ ?s2 ?s1 ?s3 ?s6)) )
       (stare ?s7&:(oddp (+ ?s3 ?s2 ?s4 ?s7)) )
       (stare ?s8&:(oddp (+ ?s4 ?s3 ?s8)) )
       (stare ?s9&:(oddp (+ ?s5 ?s1 ?s6 ?s9)) )
```

```
        (stare ?s10&:(oddp (+ ?s6 ?s2 ?s5 ?s7 ?s10)) )
        (stare ?s11&:(oddp (+ ?s7 ?s3 ?s6 ?s8 ?s11)) )
        (stare ?s12&:(oddp (+ ?s8 ?s4 ?s7 ?s12)) )
        (stare ?s13&:(oddp (+ ?s9 ?s5 ?s10 ?s13)) )
        (stare ?s14&:(oddp (+ ?s10 ?s6 ?s9 ?s11 ?s14))&:(oddp (+ ?s13 ?s9 ?s14)) )
        (stare ?s15&:(oddp (+ ?s11 ?s7 ?s10 ?s12 ?s15))&:(oddp (+ ?s14 ?s10 ?s13 ?s15)) )
        (stare ?s16&:(oddp (+ ?s12 ?s8 ?s11 ?s16))&:(oddp (+ ?s15 ?s11 ?s14 ?s16))
                &:(oddp (+ ?s16 ?s12 ?s15)) )
          =>
          (printout t ?s1 ?s2 ?s3 ?s4 t ?s5 ?s6 ?s7 ?s8 t ?s9 ?s10 ?s11 ?s12 t ?s13 ?s14
          ?s15 ?s16 t t))
(defrule solutie_5x5
        (dimensiune 5)
        (stare ?s1) (stare ?s2) (stare ?s3) (stare ?s4) (stare ?s5)
        (stare ?s6&:(oddp (+ ?s1 ?s2 ?s6)) )
        (stare ?s7&:(oddp (+ ?s2 ?s1 ?s3 ?s7)) )
        (stare ?s8&:(oddp (+ ?s3 ?s2 ?s4 ?s8)) )
        (stare ?s9&:(oddp (+ ?s4 ?s3 ?s5 ?s9)) )
        (stare ?s10&:(oddp (+ ?s5 ?s4 ?s10)) )
        (stare ?s11&:(oddp (+ ?s6 ?s1 ?s7 ?s11)) )
        (stare ?s12&:(oddp (+ ?s7 ?s2 ?s6 ?s8 ?s12)) )
        (stare ?s13&:(oddp (+ ?s8 ?s3 ?s7 ?s9 ?s13)) )
        (stare ?s14&:(oddp (+ ?s9 ?s4 ?s8 ?s10 ?s14)) )
        (stare ?s15&:(oddp (+ ?s10 ?s5 ?s9 ?s15)) )
        (stare ?s16&:(oddp (+ ?s11 ?s6 ?s12 ?s16)) )
        (stare ?s17&:(oddp (+ ?s12 ?s7 ?s11 ?s13 ?s17)) )
        (stare ?s18&:(oddp (+ ?s13 ?s8 ?s12 ?s14 ?s18)) )
        (stare ?s19&:(oddp (+ ?s14 ?s9 ?s13 ?s15 ?s19)) )
        (stare ?s20&:(oddp (+ ?s15 ?s10 ?s14 ?s20)) )
        (stare ?s21&:(oddp (+ ?s16 ?s11 ?s17 ?s21)) )
        (stare ?s22&:(oddp (+ ?s17 ?s12 ?s16 ?s18 ?s22)) &:(oddp (+ ?s21 ?s16 ?s22)) )
        (stare ?s23&:(oddp (+ ?s18 ?s13 ?s17 ?s19 ?s23))&:(oddp (+ ?s22 ?s17 ?s21 ?s23)) )
        (stare ?s24&:(oddp (+ ?s19 ?s14 ?s18 ?s20 ?s24))&:(oddp (+ ?s23 ?s18 ?s22 ?s24)) )
        (stare ?s25&:(oddp (+ ?s20 ?s15 ?s19 ?s25))&:(oddp (+ ?s24 ?s19 ?s23 ?s25))
                &:(oddp (+ ?s25 ?s20 ?s24)) )
        =>
        (printout t ?s1 ?s2 ?s3 ?s4 ?s5 t ?s6 ?s7 ?s8 ?s9 ?s10 t ?s11 ?s12 ?s13 ?s14 ?s15
        t ?s16 ?s17 ?s18 ?s19 ?s20 t ?s21 ?s22 ?s23 ?s24 ?s25 t t))
```

In soluția finală se poate observa dacă se analizează problema că trebuie să avem suma stărilor vecinilor şi a câmpului pe care se face testul impară. Din acest motiv orice stare nou adăugată va încerca să nu modifice imparitatea stările deja existente. S-a preferat la un moment dat legarea condițiilor de variabile şi nu un test final care să înglobeze toate condițiile pentru că în felul acesta se va ieşi din regulă în momentul în care unul din pattern-uri nu este adevărat. Această soluție oferă în cazul de față o viteză mai mare în găsirea soluțiilor.

2. **Problema calului** - Pe o tablă de şah se doreşte mutarea unui cal pe toată tabla de joc, trecându-se o singură dată prin fiecare câmp. Se ştie faptul că o astfel de piesă poate fi mutată doar în "L" şi că dintr-un câmp central există 8 posibilităţi de a muta, cu cât ne apropiem de marginile tablei numărul mutărilor posibile scade. Se încearcă rezolvarea metodei prin **back-traking** , astfel încât în momentul în care se ajunge într-o poziţie în care nu se mai poate muta, se vor face mutări înapoi până când putem muta pe un alt câmp decât cel anterior. Pentru a realiza aceasta se memorează ultima mutare efectuată în slotul *mutare* şi se va realiza o mutare a cărei număr de ordine este mai mare decât cel al mutării realizate anterior. Evident că la început pentru fiecare mutare nouă acest număr este 0, altfel spus toate celelalte 8 mutări fiind posibile de realizat. Pentru fiecare regulă se vor face teste ca mutarea de realizat să fie realizabilă în interiorul tablei şi pe un câmp pe care nu s-a mai mutat Faptul flag *poziţie* ne ajută să continuăm mutările de unde am rămas, totodată el memorând şi numărul mutărilor efectuate (din cele 64 posibile).

```
(deftemplate tabla
        (slot lin (type INTEGER) (range 1 8))
        (slot col (type INTEGER) (range 1 8))
        (slot val (type INTEGER) (range 1 64))
        (slot mutare (type INTEGER) (default 0)) )

(deftemplate afisare
        (slot linie (type INTEGER) (range 1 8))
        (slot coloana (type INTEGER) (range 1 8)))

(deffacts init
        (tabla (lin 1) (col 1) (val 1) (mutare 0))
        (pozitie 1)
        (spatiu 0))

(defrule mutare_1                    ; (linie =linie-1) (coloana=coloana+2);
        (declare (salience -1))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n))
        (test (and (>= ?l 2) (<= ?c 6) (< ?n 1) ))
        (not (tabla (lin =(- ?l 1)) (col =(+ ?c 2)) ))
        =>
        (modify ?f (mutare 1))
        (assert (mutare (- ?l 1) (+ ?c 2) )) )

(defrule mutare_2                    ; (linie =linie+1) (coloana=coloana+2);
        (declare (salience -2))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n) )
        (test (and (<= ?l 7) (<= ?c 6) (< ?n 2) ))
        (not (tabla (lin =(+ ?l 1)) (col =(+ ?c 2)) ))
        =>
        (modify ?f (mutare 2))
        (assert (mutare (+ ?l 1) (+ ?c 2) )) )
```

```
(defrule mutare_3                    ; (linie =linie+2) (coloana=coloana+1);
        (declare (salience -3))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n) )
        (test (and (<= ?l 6) (<= ?c 7) (< ?n 3) ))
        (not (tabla (lin =(+ ?l 2)) (col =(+ ?c 1)) ))
        =>
        (modify ?f (mutare 3))
        (assert (mutare (+ ?l 2) (+ ?c 1) )) )

(defrule mutare_4                    ; (linie =linie+2) (coloana=coloana-1);
        (declare (salience -4))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n))
        (test (and (<= ?l 6) (>= ?c 2) (< ?n 4) ))
        (not (tabla (lin  =(+ ?l 2)) (col =(- ?c 1)) ))
        =>
        (modify ?f (mutare 4))
        (assert (mutare (+ ?l 2) (- ?c 1) )) )

(defrule mutare_5                    ; (linie =linie+1) (coloana=coloana-2);
        (declare (salience -5))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n))
        (test (and (<= ?l 7) (>= ?c 3) (< ?n 5) ))
        (not (tabla (lin =(+ ?l 1)) (col =(- ?c 2)) ))
        =>
        (modify ?f (mutare 5))
        (assert (mutare (+ ?l 1) (- ?c 2) )) )

(defrule mutare_6                    ; (linie =linie-1) (coloana=coloana-2);
        (declare (salience -6))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n))
        (test (and (>= ?l 2) (>= ?c 3) (< ?n 6) ))
        (not (tabla (lin =(- ?l 1)) (col =(- ?c 2)) ))
        =>
        (modify ?f (mutare 6))
        (assert (mutare (- ?l 1) (- ?c 2) )) )

(defrule mutare_7                    ; (linie =linie-2) (coloana=coloana-1);
        (declare (salience -7))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n))
        (test (and (>= ?l 3) (>= ?c 2) (< ?n 7) ))
        (not (tabla (lin =(- ?l 2)) (col =(- ?c 1)) ))
        =>
        (modify ?f (mutare 7))
        (assert (mutare (- ?l 2) (- ?c 1) )) )
```

```
(defrule mutare_8                    ; (linie =linie-2) (coloana=coloana+1);
        (declare (salience -8))
        (pozitie ?poz)
        ?f <- (tabla (lin ?l) (col ?c) (val ?poz) (mutare ?n) )
        (test (and (>= ?l 3) (<= ?c 7) (< ?n 8) ))
        (not (tabla (lin =(- ?l 2)) (col =(+ ?c 1)) ))
        =>
        (modify ?f (mutare 8))
        (assert (mutare (- ?l 2) (+ ?c 1) )) )

(defrule trecere_un_cimp_inapoi
        (declare (salience -10))
        ?f1 <- (pozitie ?poz&:(< ?poz 62))
        ?f2 <- (tabla (val ?poz) (mutare ?n))
        (not (mutare))
        =>
        (retract ?f1 ?f2)
        (assert (pozitie (- ?poz 1))) )

(defrule trecere_cimp_urmator
        ?f1 <- (pozitie ?poz&:(< ?poz 62))
        ?f2 <- (mutare ?l ?c)
        =>
        (retract ?f1 ?f2)
        (assert  (pozitie (+ ?poz 1))
                 (tabla (lin ?l) (col ?c) (val (+ ?poz 1)) (mutare 0)) ) )

(defrule gasit_solutie
        (pozitie ?poz&:(eq ?poz 62))
        =>
        (printout t crlf " ____ ____ ____ ____ ____ ____ ____ ____" crlf)
        (printout t "|    |    |    |    |    |    |    |"crlf)
        (assert (afisare (linie 1) (coloana 1)) ) )

(defrule afisare_solutie
        ?f1 <- (afisare (linie ?l) (coloana ?c))
        (or (tabla (lin ?l) (col ?c) (val ?v))
            (spatiu ?v))
        =>
        (retract ?f1)
        (format t "| %-2d " ?v)
        (if (eq ?c 8)
         then (printout  t  t "|____|____|____|____|____|____|____|____|" t)
                 (if (neq ?l 8)
                  then (printout t "|    |    |    |    |    |    |    |    |" crlf)
                       (assert (afisare (linie (+ ?l 1)) (coloana 1)) )
                 )
         else (assert (afisare (linie ?l) (coloana (+ ?c 1)) ))
        )
)
```

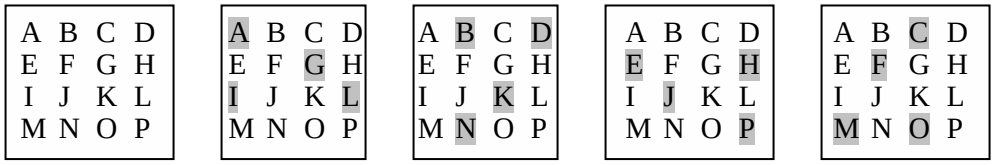
Dacă se doreşte se pot scoate din regulile *mutare_1…mutare_8* declaraţiile de prioritate. Acestea sunt puse doar pentru a ne asigura că regulile se vor aprinde în ordinea pe care o dorim (de la 1 la 8). In mod normal regulile se activează în agenda în ordinea în care acestea au fost încărcate.
Ultimele două reguli afişează soluţia găsită, folosindu-se de caracterele '_', '|' şi de spaţiul. Se preferă folosirea lui format pentru realizarea unei afişări pe un număr fix de caractere. Pentru a continua cu următoarea soluţie ar trebui realizată o regulă cu prioritatea cea mai mică, care reaprinde regula de trecere cu un câmp înapoi. Se poate observa faptul că putem comanda oprirea după un număr fixat de mutări, în cazul nostru 62: *(pozitie ?poz&:(< ?poz 62))*. Varianta de backtracking nu este cea mai avantajoasă metodă de rezolvare pe care o putem folosi într-un limbaj neprocedural. In cazul nostru un algoritm mai apropiat de problema damelor ar fi mult mai avantajos.

3. **Criptare Richelieu** - Spre deosebire metodele obişnuite de criptare a unor date, metode care în general folosesc o funcţie de tranziţie, această criptare se foloseşte de nişte şabloane la citirea datelor. Metoda poartă numele celebrului cardinal al Franţei, deoarece în timpul acestuia spionii foloseau uneori pentru a afla coordonatele şi scopul următoarei lor misiuni o foaie perforată., pe care după ce o puneau peste o anumită pagină dintr-o anumită carte, puteau citi prin perforaţii literele vizibile. In cazul nostru şablonul folosit trebuie să ducă la o criptare a informaţiei într-un mod unic şi care să permită totodată şi decriptarea (obţinerea pe cale inversă a şirului original).

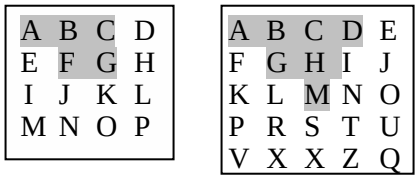
In condiţiile în care o persoană doreşte protejarea datelor dintr-un fişier de persoane neautorizate, în condiţiile în care o instituţie are date strict secrete, a căror conţinut nu trebuie să poată fi descifrat, se folosesc astfel de metode de criptare a informaţiei. De obicei cel mai simplu se poate modifica conţinutul unui octet prin adăugarea unei valori la valoarea iniţială, prin shiftarea la dreapta sau la stânga sau prin efectuarea unei funcţii matematice simple. Aceste metode din păcate nu schimbă şi locul caracterelor (octeţilor) şi din acest motiv un program de decriptare folosit de un haker ar putea viza în primul rând astfel de operaţii de codificare. In condiţiile în care se schimbă şi locul octeţilor (şi eventual se mai aplică şi o funcţie de tranziţie) informaţia devine aproape imposibil de decodificat dacă persoana neautorizată nu cunoaşte modul în care şi-au schimbat locul octeţii între ei.

Metoda lui Richelieu schimbă locul octeţilor prin simpla citire a datelor dintr-o matrice pătratică N*N după un şablon perforat şi rotirea acestuia pe rând la 90º, 180º şi 270º. Condiţia ca un şablon să poată duce la o criptare corectă a informaţiei constă în citirea fiecare element o singură dată. Altfel spus trebuie să ne asigurăm că rotind şablonul citim toate elementele matricii o singură dată. Din acest motiv într-un şablon pentru o matrice de NxN nu trebuie să avem decât o singură perforaţie pentru cele patru poziţii ale unui element *(i,j)*; *(N+1-j,N+1-i)*; *(j,i)*; *(N+1-i,N+1-i)* corespunzătoare rotirii şablonului. Informaţia dintr-un fişier este segmentată în şiruri de NxN, care odată criptate pot fi puse în fişierul criptat. Se precizează faptul că pentru o matrice pătratică cu un număr impar de linii

(coloane) elementul central trebuie citit în afara şablonului şi adăugat primul sau eventual ultimul în şirul codificat. Se poate alege orice şablon care respectă condiţia de mai sus şi din acest motiv numărul şabloanelor valide creşte odată cu dimensiunea N a matricii (pentru un număr mai mare de linii şi coloane şi numărul de şabloane pe care programul le poate utiliza creşte). Pentru a creşte complexitatea problemei se poate utiliza câte un şablon diferit pentru fiecare şir succesiv codificat, după un număr de astfel de operaţii revenind la primul şablon utilizat. Pentru o secvenţă de intrare (A B C D E F G H I J K L M N O P) se obţine la ieşire un şir codificat în mod unic dependent de şablonul utilizat.



După cum se poate observa şirul criptat obţinut folosind şablonul de mai sus cu perforaţiile în (1,1); (2,3); (3,1); (3,4) este (A G I L B D K N E H J P C F M O). După cum se poate observa una din caracteristicile şablonului este aceea că are N*N/4 perforaţii (în cazul nostru 4), fapt normal ţinând cont că în urma celor 4 rotaţii ale şablonului trebuiesc citite toate elementele matricii. Cel mai simlu şablon de obţinut ar fi acela în care avem selectate elemente:



Pentru matricile patratice cu un numar impar de linii şi coloane după cum s-a mai spus elementul central este citit o singură dată.

```
(deftemplate r   (slot lin) (slot col))          ;deftemplate pentru sablon Richelieu

(deftemplate m  (slot lin) (slot col)            ;pentru fiecare element al matricii se
                (slot val) (slot poz))           ; specifică şi poziţia sa.

(deffacts date_init
        (data A B C D E F G H I J K L M N O P)
        (rotate 0) (rotate 90) (rotate 180) (rotate 270)
        (r (lin 1) (col 1))
        (r (lin 2) (col 3))                      ;pozitii perforatii
        (r (lin 3) (col 1))
        (r (lin 3) (col 4))
        (nr_linie 1) )
```

Sunt folosite faptele de control *rotate* pentru regulile care vor citi din matrice elementele după şablonul rotit cu un număr de grade şi sunt specificate poziţiile perforaţiilor în faptele deftemplate *r*. Pentru aşezarea elementelor şirului ce va fi codificat în matricea pătratică NxN este folosit faptul *(nr_linie 1)*.

```
(defrule matrice
        (data $?d)
        ?f <- (nr_linie ?n&:(<= ?n 4))
        =>
        (retract ?f)
        (bind ?x (* 4 (- ?n 1)))
        (assert  (m (lin ?n) (col 1) (poz (+ ?x 1)) (val (nth$ (+ ?x 1) $?d)) )
                 (m (lin ?n) (col 2) (poz (+ ?x 2)) (val (nth$ (+ ?x 2) $?d)) )
                 (m (lin ?n) (col 3) (poz (+ ?x 3)) (val (nth$ (+ ?x 3) $?d)) )
                 (m (lin ?n) (col 4) (poz (+ ?x 4)) (val (nth$ (+ ?x 4) $?d)) )
                 (nr_linie (+ ?n 1)) ) )

(defrule rotate_0
        (r (lin ?l1) (col ?c1))
        (r (lin ?l2) (col ?c2))
        (r (lin ?l3) (col ?c3))
        (r (lin ?l4) (col ?c4))
        (m (lin ?l1) (col ?c1) (val ?v1) (poz ?p1))
        (m (lin ?l2) (col ?c2) (val ?v2) (poz ?p2))
        (m (lin ?l3) (col ?c3) (val ?v3) (poz ?p3))
        (m (lin ?l4) (col ?c4) (val ?v4) (poz ?p4))
        (test (and (< ?p1 ?p2 ?p3 ?p4)
                   (< ?p2 ?p3 ?p4)
                   (< ?p3 ?p4) ) )
        ?f <- (rotate 0)
        =>
        (assert (cript ?v1 ?v2 ?v3 ?v4))
        (retract ?f) )

(defrule rotate_90
        (r (lin ?l1) (col ?c1))
        (r (lin ?l2) (col ?c2))
        (r (lin ?l3) (col ?c3))
        (r (lin ?l4) (col ?c4))
        (m (lin ?c1) (col  =(- 5 ?l1)) (val ?v1) (poz ?p1))
        (m (lin ?c2) (col  =(- 5 ?l2)) (val ?v2) (poz ?p2))
        (m (lin ?c3) (col  =(- 5 ?l3)) (val ?v3) (poz ?p3))
        (m (lin ?c4) (col  =(- 5 ?l4)) (val ?v4) (poz ?p4))
        (test (and (< ?p1 ?p2 ?p3 ?p4)
                   (< ?p2 ?p3 ?p4)
                   (< ?p3 ?p4) ) )
        ?f1 <- (cript $?v)
        ?f2 <- (rotate 90)
        =>
        (retract ?f1 ?f2)
        (assert (cript $?v ?v1 ?v2 ?v3 ?v4)) )
```

```
(defrule rotate_180
        (r (lin ?l1) (col ?c1))
        (r (lin ?l2) (col ?c2))
        (r (lin ?l3) (col ?c3))
        (r (lin ?l4) (col ?c4))
        (m (lin =(- 5 ?l1)) (col =(- 5 ?c1)) (val ?v1) (poz ?p1))
        (m (lin =(- 5 ?l2)) (col =(- 5 ?c2)) (val ?v2) (poz ?p2))
        (m (lin =(- 5 ?l3)) (col =(- 5 ?c3)) (val ?v3) (poz ?p3))
        (m (lin =(- 5 ?l4)) (col =(- 5 ?c4)) (val ?v4) (poz ?p4))
        (test (and (< ?p1 ?p2 ?p3 ?p4)
                   (< ?p2 ?p3 ?p4)
                   (< ?p3 ?p4) ) )
        ?f1 <- (cript $?v)
        ?f2 <- (rotate 180)
        =>
        (retract ?f1 ?f2)
        (assert (cript $?v ?v1 ?v2 ?v3 ?v4)) )
(defrule rotate_270
        (r (lin ?l1) (col ?c1))
        (r (lin ?l2) (col ?c2))
        (r (lin ?l3) (col ?c3))
        (r (lin ?l4) (col ?c4))
        (m (lin =(- 5 ?c1)) (col ?l1) (val ?v1) (poz ?p1))
        (m (lin =(- 5 ?c2)) (col ?l2) (val ?v2) (poz ?p2))
        (m (lin =(- 5 ?c3)) (col ?l3) (val ?v3) (poz ?p3))
        (m (lin =(- 5 ?c4)) (col ?l4) (val ?v4) (poz ?p4))
        (test (and (< ?p1 ?p2 ?p3 ?p4)
                   (< ?p2 ?p3 ?p4)
                   (< ?p3 ?p4) ) )
        ?f1 <- (cript $?v)
        ?f2 <- (rotate 270)
        =>
        (retract ?f1 ?f2)
        (assert (cript $?v ?v1 ?v2 ?v3 ?v4)) )
```