Cap.3 Câmpuri şi elemente condiţionale

3.1 Constrângerea literală

Cea mai simplă constrângere pe care o putem întâlni în cadrul pattern-ului element condiţional (CE) al unei reguli este aceea în care se precizează valoarea exactă pe care trebuie să o ia un câmp pentru a se potrivi în cadrul acelui pattern. O astfel de constrângere se numeşte constrângere **literală** şi constă în precizarea unei constante de tipul *float*, *integer*, *symbol*, *string* şi *instance name*. Ea nu conţine în nici un caz variabile sau wildcard-uri. Un exemplu simplu de regulă care are o constrângere literală în LHS ar fi:

```
(defrule par_negru
        (persoana (culoare_par negru) (nume ?n))
        =>
        (printout  t  "Persoana  " ?n " are par negru."  crlf) )
```

Constrângerea literală este legată de constanta de tip simbol *negru*. După cum s-a mai spus în cazul în care avem mai multe elemente condiţionale în LHS-ul unei reguli, între acestea există un **and** implicit (în sensul că dacă este îndeplinită şi condiţia_1 şi condiţia_2 şi…condiţia_n regula se activează).

CLIPS-ul pune la dispoziţia utilizatorului şi elemente condiţionale de tipul **not**, **or**, **and** (explicit), **test**, **exists**, **forall**, **logical**, dar şi constrângeri **conective: ~** (not) , **|** (or) , **&** (and), în acest fel realizându-se reguli care să aibă o putere mai mare şi un pattern mai complex. Mai există şi constrângerile **predicative** care vor fi discutate mai târziu. Pentru început, vor fi analizate constrângerile conective, pentru ca mai apoi să fie studiate şi elementele condiţionale.

3.2. Constrângerile conective

3.2.1. Câmpul condiţional NOT

Constrângerile conective sunt utilizate pentru a 'conecta' constrângerile individuale sau variabilele, unele cu altele. Prioritatea cea mai mare o are **~** (not), urmată apoi de **&** (and) şi în cele din urmă de **|** (or). Pentru câmpul condiţional *not* se foloseşte semnul *tilda* ~, iar această constrângere va fi satisfăcută dacă expresia următoare are valoarea de adevăr egală cu simbolul FALSE.

Să ne imaginăm un set de figuri geometrice destinat preşcolarilor şi caracterizat prin formă, culoare şi grosime. Dacă, de exemplu, dorim eliminarea dintre aceste figuri, a acelora ce îndeplinesc anumite caracteristici, folosim în cadrul regulii câmpul condiţional not.

```
(deftemplate figura_geometrica
        (slot forma) (slot culoare) (slot grosime) )
```

```
(deffacts figuri
        (figura_geometrica (forma patrat) (culoare rosie) (grosime mica))
        (figura_geometrica (forma patrat) (culoare galbena) (grosime medie))
        (figura_geometrica (forma dreptunghi) (culoare rosie) (grosime mica))
        (figura_geometrica (forma cerc) (culoare albastra) (grosime mare))
        (figura_geometrica (forma romb) (culoare neagra) (grosime medie)) )

(defrule nu_este_patrat
        (figura_geometrica (forma ~patrat))
        =>
        )
```

În regula noastră nu avem nici o variabilă pe care să o putem folosi la afişarea figurii geometrice ce îndeplineşte condiția de a nu fi *pătrat* şi deşi regula are trei activări în agendă (se poate vedea după ce se dă comanda *reset,* cu ajutorul comenzii *agenda*), nu putem vizualiza nimic pe monitor. Față de regula de mai sus, regula următoare foloseşte şi o variabilă *?forma* pentru afişare.

```
(defrule nu_este_patrat
        (figura_geometrica (forma ?forma &~patrat))
        =>
        (printout t " S-a gasit un " ?forma "." crlf) )
```

O altă cale ar fi fost realizarea unei reguli pentru afişarea fiecărei figuri în parte, dar această metodă este neavantajoasă mai ales atunci când baza de cunoştințe este foarte vastă. Este mult mai uşor de realizat o restricție pentru valoarea unui câmp, limitându-se astfel valorile pe care le poate lua un câmp în LHS-ul regulii. In cazul nostru constrângerea *not* realizează negarea constrângerilor următoare, respectiv condiția de a nu fi *pătrat*. Pentru a realiza legarea unei variabile de o constrângere conectivă trebuie folosit în mod obligatoriu ampersand-ul '&', în caz contrar CLIPS-ul semnalând eroare.

3.2. Câmpul condițional OR

Condiția **or** reprezentată prin bara verticală '|', este folosită pentru a permite realizarea corespondenței unui câmp al unui pattern cu una sau mai multe valori posibile. Dacă se doreşte selectarea figuri geometrice ce au grosime mică sau medie, regula va folosi un astfel de câmp condițional *or*.

```
(defrule grosime_mica_sau_medie
        (figura_geometrica (forma ?f) (grosime mica | medie))
        =>
        (printout t "Gasit un " ?f " care nu are grosimea mare." crlf) )
```

Nici în acest caz nu putem şti care figură geometrică a avut grosimea mică şi care a avut grosimea mare, deoarece nu avem o variabilă care să poată fi folosită în RHS-ul regulii la afişare. Următoarea regulă va realiza acest lucru, cu restricția de a folosi ampersand-ul pentru legarea variabilei de constrângerile conective:

```
(defrule grosime_mica_sau_medie
        (figura_geometrica (forma ?f) (grosime ?grosime &mica | medie))
        =>
        (printout  t  "Gasit un " ?f  " care are grosimea "  ?grosime "." t))
```

Observaţie: CLIPS-ul nu va semnala nici o eroare atunci când se introduce greşit într-o constrângere valoarea pentru care se va face căutarea în baza de fapte, (sintaxa fiind corectă). De exemplu, dacă se va folosi în loc de *medie* termenul de *mijlocie,* CLIPS-ul va căuta un fapt care să se potrivească cu pattern-ul regulii şi deoarece nu va găsi nici un fapt care să aibă valoarea slotului *grosime,* identică cu simbolul *mijlocie* va afişa doar figurile geometrice de grosime *mica*.

```
(defrule grosime_mica_sau_medie
        (figura_geometrica (forma ?f) (grosime ?grosime &mica | mijlocie))
        =>
       (printout  t  "Gasit un " ?f  " care are grosimea "  ?grosime " ." crlf) )
```

3.2.3. Câmpul condiţional AND

Al treilea şi ultimul tip de constrângere conectivă este constrângerea **and**. Este reprezentată prin simbolul **&** (ampersand) şi este de obicei folosită pentru a lega alte constrângeri, în rest utilizarea ei nefiind practică (unui câmp îi este asociată o valoare şi este inutil să verifici dacă acel câmp are două valori în acelaşi timp). După cum s-a văzut este foarte utilă atunci când se realizează legarea unei variabile de una sau mai multe constrângeri (condiţii) adiţionale, dacă se doreşte, de exemplu, eliminarea acelor figuri care au culoarea roşie sau culoarea albastră:

```
(defrule nu_are_culoarea_rosie_sau_albastra
        (figura_geometrica (forma ?f) (culoare ?c &~rosie&~albastra))
        =>
        (printout  t  "Gasit un " ?f " de culoare " ?c "." crlf))
```

Se observă folosirea constrângerii conective *and* pentru legarea unei variabile de una sau mai multe condiţii dar şi pentru conectarea mai multor constrângeri între ele. În exemplul următor se poate vedea inutilitatea folosirii lui *and* :

```
(defrule inutil
        (figura_geometrica (forma ?f) (culoare rosie &~neagra))
        =>
        (printout  t  "Gasit un " ?f  " de culoare rosie." crlf))
```

Evident, dacă figura are culoarea roşie, nu are simultan culoarea neagră. Se observă acest lucru şi din cele două activări ale regulii, CLIPS-ul negăsind decât un dreptunghi şi un pătrat care se potrivesc cu pattern-ul regulii (culoarea roşie). Când se formulează restricţiile se pot realiza erori de programare, dacă condiţiile nu sunt bine enunţate.

3.2.4. Combinarea câmpurilor condiţionale

Pentru realizarea unei reguli cu un pattern foarte puternic se pot mixa câmpurile condiționale cu variabile şi valori literale. De exemplu dacă se doreşte realizarea unor perechi de figuri în care prima să fie obligatoriu un pătrat, de culoare roşie sau galbenă, iar a doua figură să nu fie pătrat, să aibă culoarea albastră sau o culoare diferită de cea a perechii sale, iar grosimea celor două să fie identică; se poate realiza o regulă de forma:

```
(defrule pereche_de_figuri
        (figura_geometrica (forma patrat)(culoare ?c&rosie | galbena)(grosime ?g))
        (figura_geometrica (forma ~patrat) (culoare ~?c | albastra) (grosime ?g))
        =>
        (printout  t  "S-a gasit o astfel de pereche."  crlf))
```

Pentru această regulă s-a găsit o pereche de fapte care satisfac toate condițiile din pattern. Pentru a putea afişa datele celor două figuri folosim o serie de variabile:

```
(defrule pereche_de_figuri
        (figura_geometrica(forma patrat)(culoare ?c1&rosie |galbena)(grosime ?g))
        (figura_geometrica(forma ?f2 &~patrat)
                                (culoare ?c2&~?c1 | albastra)(grosime ?g))
        =>
        (printout  t  "(Patrat de culoare " ?c1 " - " ?f2  " de culoare " ?c2 ")" t)
        (printout  t  "Ambele figuri au grosime " ?g "." crlf) )
```

Trebuie reținut faptul că o variabilă va fi legată de o valoare numai dacă apare ca primă condiție într-un câmp, fiind legată apoi de celelalte condiții prin constrângerile conective. De exemplu, în regula următoare se va produce o eroare deoarece variabila *?c* nu este legată de nici o valoare.

```
(defrule utilizare_gresita_a_unei_variabile
        (figura_geometrica (forma ?f) (culoare rosie | ?c))
        =>
        (printout  t  "Gasit " ?f  " de culoare " ?c "."crlf) )
```

De asemenea trebuie reținut şi faptul că nu toate combinațiile de constrângeri sunt utile. Utilizând constrângerea *and* între două constante literale (ex.: roşie&albastră) se va obține întotdeauna o condiție nesatisfăcută dacă cele două constante literale nu sunt identice. De asemenea utilizând constrângerea *or* între două constante literale negate (ex: ~roşie | ~albastră) se obține întotdeauna o constrângere satisfăcută. Pentru a vedea utilitatea folosirii acestor constrângeri se va realiza permutarea a trei obiecte (pot fi şi cifre), cu ajutorul unei singure reguli.

```
(deffacts obiecte_de_permutat
        (obiect  rosu)
        (obiect  galben)                ;valorile pot fi şi cifre
        (obiect  albastru) )
```

```
(defrule permutare_de_trei_obiecte
        (obiect ?o1)
        (obiect ?o2 &~?o1)
        (obiect ?o3 &~?o2 &~?o1)
        =>
        (printout  t  "(" ?o1 " " ?o2 " " ?o3 ")"  crlf) )
```

Se vor observa după ce au fost date comenzile *reset* şi *run* toate cele şase permutări posibile. In acest exemplu se poate vedea modul în care programatorul poate utiliza mai multe fapte cu acelaşi nume, restricționându-le ca acestea să apară în pattern-ul regulii doar o singură dată. Dacă în construcția deffacts am avea două obiecte din cele trei identice regula nu s-ar mai activa deloc (se poate încerca). Dacă am avea mai mult de trei obiecte am obține aranjamente de trei luate câte patru (adică 24 de activări). Modificând regula prin păstrarea primelor două pattern-uri se obțin toate perechile posibile de realizat din obiectele existente în lista de fapte.

Dacă am introduce în aceste fapte ordonate cifre în locul simbolurilor am obține permutări de cifre mai uşor de urmărit şi verificat (se ştie că într-un fapt ordonat putem pune orice fel de tipuri de date, iar regula noastră poate permuta şiruri, întregi sau simboluri, simultan). Pentru construcția deffacts:

```
(deffacts cifre_de_permutat
        (obiect 1)
        (obiect 2)
        (obiect 3))
```

observăm că prima din cele şase permutări este (3 2 1). CLIPS-ul vede lista de fapte ca o stivă LIFO şi deci ultimul fapt introdus va fi primul care va fi scos şi se va încerca potrivirea sa cu pattern-ul unei reguli. Dacă dorim obținerea de permutări începând cu permutarea (1 2 3) trebuie ca ultimul obiect introdus în listă să aibă valoarea 1, iar primul obiect introdus să aibă valoarea 3. Construcția deftemplate ar arăta în acest caz astfel:

```
(deffacts cifre_de_permutat
        (obiect 3)
        (obiect 2)
        (obiect 1))
```

3.3 Elementele condiționale

3.3.1. Elementul condițional OR

După cum s-a mai spus între pattern-urile unei reguli există un *and* implicit. Din acest motiv o regulă nu se activează decât dacă toate pattern-urile sunt adevărate. CLIPS-ul oferă posibilitatea folosirii în LHS şi a unui *and* explicit, precum şi a unui *or* explicit. Pentru a vedea utilitatea folosirii unui element

condiţional, fie următoarele reguli (utilizăm construcţia deftemplate *figura_geometrică* şi construcţia deffacts *figuri* definite mai sus):

```
(defrule este_patrat_rosu
        (figura_geometrica (forma patrat) (culoare rosie))
        =>
        (printout  t  "S-a gasit figura cautata."  crlf))

(defrule este_cerc_albastru
        (figura_geometrica (forma cerc) (culoare albastra))
        =>
        (printout  t  "S-a gasit figura cautata."  crlf))
```

Cele două reguli au un efect aproape identic, prima regulă căutând un pătrat de culoare roşie şi afişând în caz de succes un mesaj, a doua regulă căutând însă un cerc albastru. Dacă s-ar încerca compactizarea celor două reguli cu ajutorul constrângerilor conective acest lucru nu ar fi posibil. O astfel de regulă:

```
(defrule cauta_figura
        (figura_geometrica (forma patrat | cerc) (culoare rosie | albastra))
        =>
        (printout  t  "S-a gasit figura cautata." crlf ))
```

s-ar activa după cum se observă şi pentru un pătrat albastru sau un cerc roşu (se poate verifica asertând un astfel de fapt nou). Se poate realiza o singură regulă care să înglobeze în ea ambele reguli prin folosirea *elementului condiţional* **or** care să lege pattern-urile celor două reguli. Regula se va activa fie dacă primul pattern va fi adevărat, fie dacă al doilea pattern va fi adevărat.

```
(defrule cauta_figura
        (or     (figura_geometrica (forma cerc) (culoare albastra))
                (figura_geometrica (forma patrat) (culoare rosie)) )
        =>
        (printout  t  "S-a gasit figura cautata."  crlf))
```

Dacă s-ar dori găsirea unei singure figuri din mai multe posibile, după care căutarea să înceteze, s-ar putea utiliza un fapt ca flag *(cautare (valoare on))*, care să se transforme automat în *(cautare (valoare off))* sau am putea retracta acel fapt-flag astfel încât unul din pattern-urile regulii nu va mai fi adevărat şi deci regula nu s-ar mai activa încă o dată. Un astfel de exemplu ar arăta că pot exista în afara elementului condiţional *or* şi alte condiţii. Cele două reguli vor fi modificate astfel încât să nu se activeze decât o singură dată (se caută o singură figură):

```
(defrule cauta_patrat_rosu
        ?f <- (cautare (valoare on))
        (figura_geometrica (forma patrat) (culoare rosie))
        =>
        (modify ?f  (valoare off))
        (printout  t  "S-a gasit figura cautata."  crlf))
```

```
(defrule cauta_cerc_albastru
        ?f <- (cautare (valoare on))
        (figura_geometrica (forma cerc) (culoare albastra))
        =>
        (modify ?f  (valoare off))
        (printout  t  "S-a gasit figura cautata."  crlf))
```

Această soluție oferă posibilitatea modificării în cadrul altei reguli a valorii faptului-flag din nou în *on* şi deci operația de căutare se poate relua (după ce s-a realizat o interpretare a datelor curente). Pentru a funcționa trebuie realizată o construcție deftemplate *cautare* cu un singur slot *valoare* şi trebuie asertat un nou fapt cu valoarea *on*. Cele două reguli se pot reuni într-una singură, astfel:

```
(defrule cauta_figura
        ?f <- (cautare (valoare on))
        (or     (figura_geometrica (forma cerc) (culoare albastra))
                (figura_geometrica (forma patrat) (culoare rosie)) )
        =>
        (modify ?f  (valoare off))
        (printout  t  "S-a gasit figura cautata."  crlf))
```

Cealaltă soluție care presupunea ştergerea faptului-flag (şi nu modificarea sa) este la fel de eficientă, operația de căutare reluându-se imediat ce flag-ul a fost reasertat în cadrul unei alte reguli. Pentru ştergerea figurii care a fost analizată trebuie reținut faptul că o regulă cu un element condițional *or* este echivalentă cu mai multe reguli şi deci se va lega fiecare pattern de o variabilă cu acelaşi nume.

```
(defrule cauta_figura
        ?f <- (cautare (valoare on))
        (or     ?fig <- (figura_geometrica (forma cerc) (culoare albastra))
                ?fig <- (figura_geometrica (forma patrat) (culoare rosie)) )
        =>
        (retract ?fig)
        (modify ?f  (valoare off))
        (printout  t  "S-a gasit figura cautata."  crlf))
```

Dacă nu s-ar fi legat fiecare pattern al elementului condițional *or (*putem avea mai mult de două pattern-uri) de variabila folosită apoi de funcția *retract*, atunci în anumite situații faptul nu ar fi fost şters (este posibil ca utilizatorul să nu dorească ştergerea figurii în anumite cazuri).

3.3.2. Elementul condițional AND

Elementul condițional **and** (CE *and*) este opusul elementului condițional **or**. O regulă se va aprinde numai dacă toate pattern-urile vor fi satisfăcute. Deoarece CLIPS-ul plasează în mod implicit un CE *and* în LHS-ul regulii, orice regulă poate fi rescrisă folosindu-se un *and* explicit.

```
(regula nume_regulă                    |    (regula nume_regulă
        pattern_1                      |            (and    pattern_1
        pattern_2                      |                    pattern_2
        ……                             |                    ……
        pattern_n                      |                    pattern_n)
        =>                             |            =>
        acţiuni )                      |            acţiuni )
```

Nu există nici un avantaj în scrierea unei reguli care să conţină întreg LHS-ul cuprins într-un CE *and* explicit. Utilitatea folosirii unui element condiţional se observă când împreună cu alte elemente condiţionale realizează o regulă cu un pattern mai complex. De exemplu dacă s-ar specifica ce figură anume se caută:

```
(defrule cauta_figura
        (or  (and  ?f <- (cautare_cerc_albastru)                    ;flag1
                         (figura_geometrica (forma cerc) (culoare albastra)) )
             (and  ?f <- (cautare_patrat_rosu)                      ;flag2
                         (figura_geometrica (forma patrat) (culoare rosie)) ) )
        =>
        (retract ?f)
        (printout  t  "S-a gasit figura cautata."  crlf))
```

După cum se observă, această regulă se activează în două situaţii, fie atunci când se caută şi este găsit un *cerc albastru*, fie atunci când se caută şi este găsit un *pătrat roşu*. Ea poate fi descompusă în două reguli mai mici în care nu se va mai folosi CE *and* (deoarece vom avea un *and* implicit între pattern-uri) :

```
(defrule cauta_cerc_albastru
        ?f <- (cautare_cerc_albastru)
        (figura_geometrica (forma cerc) (culoare albastra)) )
        =>
        (retract ?f)
        (printout  t  "S-a gasit figura cautata."  crlf) )

(defrule cauta_patrat_rosu
        ?f <- (cautare_patrat_rosu)
        (figura_geometrica (forma patrat) (culoare rosie)) ) )
        =>
        (retract ?f)
        (printout  t  "S-a gasit figura cautata."  crlf) )
```

3.3.3. Elementul condiţional NOT

Uneori este utilă activarea unei reguli în condiţiile în care un fapt particular nu există în lista de fapte. CLIPS-ul asigură această facilitate programatorului prin elementul condiţional **not**. Cu ajutorul CE *not* poate fi negat doar un singur pattern. Dacă se doreşte negarea mai multor pattern-uri trebuiesc folosite mai multe CE *not*.

<CE not> ::= (not <element-conditional>)

De exemplu regula următoare se activează atât timp cât nu există în lista de fapte, un fapt cu numele *stop*:

```
(defrule afiseaza_informatii_figuri
        (not    (stop))
        (figura_geometrica (forma ?f) (culoare ?c) (grosime ?g))
        =>
        (printout t "Gasit "?f " de culoare " ?c " si grosime " ?g "." crlf) )
```

```
CLIPS> (run 1)          ; Ctrl+T în varianta CLIPS pentru Windows
Gasit romb de culoare neagră şi grosime medie.
CLIPS> (assert (stop))
<Fact-6>                        ;toate cele patru activari ale regulii se pierd
CLIPS> (run)
CLIPS>
```

Există reguli care sunt reciproc exclusive, adică nu vor putea fi niciodată activate amândouă în acelaşi timp în agendă şi acest lucru deoarece unul dintre pattern-urile fiecărei reguli nu va fi niciodată satisfăcut simultan (celelalte pattern-uri pot fi identice). Acest lucru se poate uşor realiza prin negarea în cadrul unei reguli a pattern-ului celeilalte reguli. De exemplu :

```
(defrule este_figura_rosie
        (cautare)                                       ;flag
        (figura_geometrica (forma ?f) (culoare rosie))
        =>
        (printout t "Gasit " ?f " de culoare rosie." crlf) )

(defrule nu_este_figura_rosie
        (cautare)                                       ;flag
        (not    (figura_geometrica (culoare rosie)) )
        =>
        (printout t "Nu avem figuri de culoare rosie." crlf) )
```

```
CLIPS> (assert (cautare))
CLIPS> (agenda)
0       este_figura_rosie: f-7,f-3
0       este_figura_rosie: f-7,f-1
For a total of 2 activations.
CLIPS> (retract 3 1)
CLIPS> (agenda)
0       nu_este_figura_rosie: f-7,
For a total of 1 activations.
```

După cum se observă prima regulă este activată cât timp avem în lista de fapte faptele f-1 şi f-3 (fapte care au valoarea slotului *culoare* identică cu simbolul *rosie*). La retractarea acestora se activează cealaltă regulă. Explicația este legată de faptul cel de-al doilea pattern al fiecărei reguli este negatul celuilalt.

O variabilă legată de o valoare în cadrul unui CE *not* poate fi utilizată doar în acel pattern. Variabilele legate în afara unui CE *not* pot fi folosite fără a avea probleme în cadrul acestuia. In exemplul următor CLIPS-ul va semnala eroare, deoarece variabila *?f* a fost iniţializată într-un element condiţional *not* şi a fost folosită mai apoi în RHS-ul regulii când s-a dorit afişarea valorii acesteia.

```
(defrule eroare_de_variabila
        (not    (figura_geometrica  (forma ?f) (culoare rosie)) )
        =>
        (printout  t "Nu exista " ?f  "de culoare rosie." crlf))
```

[PRCCODE3] Undefined variable f referenced in RHS of defrule.

Elementul condiţional *not* poate fi folosit şi atunci când se verifică existenţa unor fapte cu valorile sloturilor identice. Regula următoare realizează afişarea acelor figuri care nu au alte figuri de formă diferită şi aceiaşi culoare. Se vor găsi trei activări ale regulii în agendă:

```
(defrule verifica_forma
        (figura_geometrica (forma ?f) (culoare ?c))
        (not    (figura_geometrica  (forma ~?f) (culoare ?c)) )
        =>
        (printout  t  "Culoarea " ?c " este intalnita doar la figurile "  ?f   t))
```

Inversarea celor două pattern-uri duce la o funcţionare incorectă a regulii. Valorile legate de variabilele ?f şi ?c în primul pattern al regulii (cel cu CE *not*) nu au nici un efect asupra valorilor permise pentru aceleaşi variabile în al doilea pattern. Trebuie reţinut deci faptul că plasarea incorectă a CE *not* poate afecta activarea unei reguli.

```
(defrule verifica_forma
        (not    (figura_geometrica  (forma ~?f) (culoare ?c)) )
        (figura_geometrica (forma ?f) (culoare ?c))
        =>
        (printout  t  "Culoarea " ?c " este intalnita doar la figurile "  ?f   t))
```

Similar, dată fiind o bază de cunoştinţe cu mai multe persoane, putem afla dacă nu există două persoane care au ziua de naştere pe aceiaşi dată calendaristică :

```
(defrule nu_exista_zile_de_nastere_identice
        (not    (and    (persoana (nume ?n) (zi_nastere ?z))
                        (persoana (nume ~?n) (zi_nastere ?z)) ))
        =>
        (printout  t  "Nu avem doua persoane cu aceiasi zi de nastere."  t ) )
```

Pentru fiecare regulă care are primul element condiţional al oricărui CE *and* (implicit sau explicit) un CE *not* sau CE *test* CLIPS-ul va adăuga automat pattern-ul *initial-fact*. Din acest motiv regula:

```
(defrule nici_o_cautare
        (not    (cautare))
```

```
        =>
        (printout  t  "Nu se efectueaza nici o cautare." crlf))
```

este convertită în următoarea regulă:

```
(defrule nici_o_cautare
        (initial-fact)
        (not      (cautare))
        =>
        (printout  t  "Nu se efectueaza nici o cautare." crlf))
```

Această convenție poate fi înțeleasă dacă se studiază ieşirea comenzii *matches*. Pentru indicele de fapt pentru CE *not* nu sunt afişate match-urile parțiale sau în activările regulilor din agenda. Activarea "f-7,, f-3" arată potrivirea primului CE cu faptul care are index-ul 7, al doilea CE este un CE *not* care nu se potriveşte cu nici unul din faptele existente (fiind satisfăcut) şi în sfârşit, cel de-al treilea CE care se potriveşte cu faptul cu index-ul 3.

3.3.4. Elementul condițional EXISTS

Elementul condițional **exists** asigură potrivirea cu un pattern în condițiile în care există cel puțin un fapt dintr-un număr total de fapte care se potrivesc cu acel pattern. Se permite astfel o *singură potrivire* parțială sau o singură activare pentru o regulă care a fost realizată pe baza existenței unui fapt dintr-o clasă de fapte. Date fiind regulile:

```
        <CE exists> ::= ( exists  <element-conditional>+ )

(defrule figura_rosie
        (figura_geometrica (culoare rosie))
        =>
        (printout  t  "Gasit figura de culoare rosie."  crlf))

(defrule exista_figura_rosie
        (exists    (figura_geometrica (culoare rosie)))
        =>
        (printout  t  "Gasit figura de culoare rosie. "  crlf))

CLIPS> (agenda)
0       exista_figura_rosie: f-0,
0       figura_rosie:     f-3
0       figura_rosie: f-1
For a total of 3 activations.
```

Se poate observa foarte clar cum o regulă care are mai multe activări în agendă, în momentul în care este modificată prin introducerea CE *exists* nu mai are decât o singură activare. Se observă cum în mod automat primul pattern al regulii se potriveşte cu *initial-fact* (indexul 0), al doilea fapt potrivindu-se ca şi în cazul CE *not* cu mai multe fapte, este lăsat fără vreun index de fapt. Explicația apariției

pattern-ului *initial-fact* este legată de modul în care un CE *exists* este implementat din două CE *and* între care se află două CE *not*. Astfel LHS-ul regulii de mai sus poate fi convertit prin înlocuirea elementului condiţional *exists* rezultând:

```
(defrule exista_figura_rosie
        (and  (not  (not  (and      (figura_geometrica (culoare rosie)) ))))
                =>
                (printout  t  "Gasit figura de culoare rosie. "  crlf))
```

Deoarece CE *and* care încadrează tot LHS-ul regulii are primul element condiţional un CE *not*, CLIPS-ul va introduce, după cum s-a mai spus, faptul iniţial. Realizând această adăugare şi ştergând CE *and* ce încadrează pattern-ul *figura_geometrica* se obţine:

```
(defrule exista_figura_rosie
        (and     (initial-fact)
                 (not     (not      (figura_geometrica (culoare rosie)) )))
        =>
        (printout  t  "Gasit figura de culoare rosie. "  crlf))
```

Deoarece un CE *exists* este realizat din CE *not* toate restricţiile privitoare la legarea unei variabile de o valoare şi de modul de amplasare al elementului condiţional în LHS-ul regulii sunt şi aici valabile.

3.3.5. Elementul condiţional FORALL

CLIPS-ul permite cu ajutorul acestui element condiţional potrivirea cu un pattern realizat din mai multe elemente condiţionale (un grup) care trebuiesc să fie satisfăcute fiecare în parte. Este suficient ca doar un singur CE din acest grup să nu fie satisfăcut şi regula nu se mai activează. De exemplu:

```
CLIPS> (deftemplate figura_geometrica (slot forma) (slot culoare))
CLIPS> (defrule exemplu_pentru_forall
                (forall   (figura_geometrica (forma ?f) (culoare ?c))
                          (cauta_forma ?f)
                          (cauta_culoare ?c) )
                =>
                (printout  t  "Elementul conditional forall este satisacut." t))

CLIPS> (reset)
==> Activations 0        exemplu_pentru_forall: f-0,
CLIPS> (assert (figura_geometrica (forma patrat) (culoare alba)))
<== Activations 0        exemplu_pentru_forall: f-0,
<Fact-1>
CLIPS> (assert (cauta_culoare alba)   (cauta_forma patrat))
==> Activations 0        exemplu_pentru_forall: f-0,
<Fact-3>
CLIPS> (assert (cauta_forma cerc))
```

```
<Fact-4>
CLIPS> (assert (figura_geometrica (forma cerc) (culoare verde)))
<== Activations 0       exemplu_pentru_forall: f-0,
<Fact-5>
CLIPS> (assert (cauta_culoare verde))
==> Activations 0       exemplu_pentru_forall: f-0,
<Fact-6>
CLIPS> (retract  3)
<== Activations 0       exemplu_pentru_forall: f-0,
CLIPS> (retract  2)
CLIPS> (retract  1)
==> Activations 0       exemplu_pentru_forall: f-0,
```

Se poate observa din acest exemplu că primul CE din grupul elementului condițional *forall* are cea mai mare importanță (în cazul nostru deftemplate-ul figura_geometrică). În momentul asertării unui asemenea fapt regula își pierde activarea, fiind necesară asertarea celorlalte fapte din grup pentru reactivarea regulii (valorile sloturilor trebuia să fie aceleaşi). Un efect similar se produce şi în cazul retractării unui CE din grup. Formatul general al unui CE *forall* este:

<CE forall> ::= (forall <primul-CE> <restul-CEs>+)

In concluzie un CE *forall* este satisfăcut dacă pentru fiecare fapt care se potriveşte cu <primul-CE> avem şi fapte care să se potrivească cu <rest-CEs>+. Acest element condițional poate fi realizat şi el de asemenea din CE *and* şi CE *or*.

```
(not    (and    <primul-CE>
                (not    (and    <restul-CEs>+) ) ) )
```

Din aceleaşi motive explicate mai sus este introdus pattern-ul *initial-fact* şi se realizează o serie de restricții care trebuiesc respectate.

3.3.6. Elementul condițional LOGICAL

Elementul condițional **logical** permite specificarea existenței unui fapt care este dependent de existența altui fapt sau grup de fapte. O entitate "pattern" creată în RHS-ul regulii (sau ca rezultat al acțiunilor efectuate în RHS) poate fi dependentă logic de entitățile "pattern" (model) din LHS-ul regulii care s-au potrivit cu pattern-urile încadrate de CE *logical* . O astfel de entitate poate avea ca suport logic un grup de entități pattern, fiecare din diferite reguli. În cazul în care vreo entitate din această grupare ce formează suportul logic este şterasă, atunci şi entitatea dependentă respectivă este şterasă.

```
(defrule cauta_patrat_galben
        (cauta_figura)
        (figura_geometrica (forma patrat) (culoare galben))
        =>
        (assert (gasit_figura)) )
```

```
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (watch facts)
CLIPS> (assert (figura_geometrica (forma patrat) (culoare galben))
                    (cauta_figura))
==>     f-1     (figura_geometrica (forma patrat) (culoare galben))
==>     f-2     (cauta_figura)
<Fact-2>
CLIPS> (run)
==>     f-3     (gasit_figura)
CLIPS> (retract 1 2)
<==     f-1     (figura_geometrica (forma patrat) (culoare galben))
<==     f-2     (cauta_figura)
CLIPS> (facts)
f-0     (initial-fact)
f-3     (gasit_figura)
For a total of 2 facts.
```

Se poate observa modul în care ştergerea faptelor din LHS-ul unei reguli ce conține CE *logical*, afectează un nou fapt dependent de acestea, introdus în RHS.

```
(defrule cauta_patrat_galben
        (logical(cauta_figura)
                (figura_geometrica (forma patrat) (culoare galben)))
        =>
        (assert (gasit_figura)) )
```

```
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (watch facts)
CLIPS> (assert (figura_geometrica (forma patrat) (culoare galben))
                    (cauta_figura))
==>     f-1     (figura_geometrica (forma patrat) (culoare galben))
==>     f-2     (cauta_figura)
<Fact-2>
CLIPS> (run)
==>     f-3     (gasit_figura)
CLIPS> (retract 1)
<==     f-1     (figura_geometrica (forma patrat) (culoare galben))
<==     f-3      (gasit_figura)
```

In acest moment la execuția regulii se stabileşte o legătură între faptul asertat în RHS-ul regulii şi faptele ce se potrivesc cu pattern-urile cuprinse de CE *logical* în LHS-ul regulii, astfel încât în cadrul acestei reguli retractarea faptului *(cauta_figura)* sau a faptului *(figura_geometrica (forma patrat) (culoare galben))* duce automat, după cum se vede din exemplu, la retractarea faptului *(gasit_figura)*. Suportul logic pentru noul fapt asertat în RHS este format dintr-o grupare de două fapte, ambele făcând parte din aceeaşi regulă. Spunem că avem între faptele din

suportul logic şi faptul dependent de acestea, o stare de dependență. CE *logical* nu trebuie să includă toate pattern-urile din LHS-ul unei reguli, el trebuie să conțină doar primul CE al unei reguli. In unele cazuri poate fi utilă realizarea unei dependențe legate de non-existența unui fapt, acest lucru fiind posibil în momentul în care se foloseşte un CE *not* împreună cu un CE *logical*. Se pot realiza condiții mai complexe prin utilizarea elementelor condiționale *exists* şi *forall*.

Dacă suportul logic era, până în acest moment, realizat din fapte care se potriveau cu pattern-urile unei singure reguli, se va încerca acum realizarea unui suport logic din fapte ce se potrivesc cu pattern-urile unor reguli diferite, elementul condițional *logical* conținând de această dată doar primul CE din LHS :

```
(defrule cauta_patrat_rosu
        (logical (figura_geometrica (forma patrat) (culoare rosu)) )
        (cauta_patrat)
        =>
        (assert (gasit_figura)) )

(defrule cauta_patrat_galben
        (logical (figura_geometrica (forma patrat) (culoare galben)) )
        (cauta_patrat)
        =>
        (assert (gasit_figura)) )

CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (assert (figura_geometrica (forma patrat) (culoare rosu))
                 (figura_geometrica (forma patrat) (culoare galben))
                 (cauta_patrat))
==>     f-1     (figura_geometrica (forma patrat) (culoare rosu))
==>    f-2      (figura_geometrica (forma patrat) (culoare galben))
==>     f-3     (cauta_patrat)
<Fact-3>
CLIPS> (run)
FIRE    1       cauta_patrat_galben: f-3,f-2
==>     f-4     (gasit_figura)
FIRE    2       cauta_patrat_rosu: f-1,f-3
CLIPS> (retract 2)
<==     f-2     (figura_geometrica (forma patrat) (culoare galben))
CLIPS> (retract 1)
<==     f-2     (figura_geometrica (forma patrat) (culoare rosu))
<==     f-4      (gasit_figura)
```

Se observă faptul că în fiecare din cele două reguli datorită CE *logical* se va realiza un suport logic pentru noul fapt creat *(gasit_figura)*, format din două fapte care se potrivesc fiecare cu pattern-ul altei reguli. Ştergerea unui fapt din suportul logic nu duce în acest caz la ştergerea automată a faptului dependent de acel suport, fiind necesară şi ştergerea celui de-al doilea fapt. Un fapt care a fost

asertat la nivelul prompter-ului sau în RHS-ul unei reguli, care nu conţine un CE *logical* are un suport necondiţionat, acest lucru făcând imposibilă retractarea sa automată, în momentul retractării altui fapt. CLIPS-ul oferă pentru vizualizarea stărilor de dependenţă dintre fapte două comenzi, a căror sintaxă este următoarea:

(dependets <fact-index-or-address>)
(dependencies <fact-index-or-address>)

Prima comandă **dependents** oferă o listă cu toate faptele sau instanţele care primesc suport logic, a doua comandă **dependencies** o listă a faptelor de la care faptul specificat primeşte suport logic. Presupunând că pentru exemplul anterior nu a fost realizată nici o retractare, avem în lista de fapte toate cele şase fapte: f-0 *(initial - fact),* f-1 *(figura_geometrica (forma patrat)(culoare galben)),* f-2 *(figura_geometrica (forma dreptunghi)(culoare galben)),* f-3 *(cauta_patrat),* f-4 *(cauta_ dreptunghi),* f-5 *(gasit_figura).*

In afară de următoarele cazuri, cele două comenzi vor returna simbolul *None* :

```
CLIPS> (dependents 3)
f-5
CLIPS> (dependents 4)
f-5
CLIPS> (dependencies 5)
f-3
f-4
```

3.3.7. Elementul condiţional TEST

Elementul condiţional **test** oferă o modalitate foarte puternică de evaluare a expresiilor din cadrul părţii stângi a unei reguli (LHS). În unele cazuri este foarte utilă repetarea unor calcule sau procesarea altor informaţii. O modalitate pentru a realiza acest lucru este prin realizarea unei bucle. Spre exemplu, în cazul în care un program cere utilizatorului introducerea unor date, iar aceste date sunt restricţionate pentru o bună funcţionare a programului, să ia anumite valori, se poate intra într-o buclă până în momentul în care sunt furnizate la intrare date corecte. Sintaxa unui CE *test* este următoarea:

<test CE> ::= (test <function-call>)

În locul realizării corespondenţei (potrivirii) unui fapt cu un pattern, CE *test* realizează evaluarea unei expresii. Un element condiţional *test* este satisfăcut atunci când funcţia apelată sau expresia evaluată returnează o valoare care este diferită de simbolul FALSE. În cazul în care valoarea returnată este FALSE, atunci CE *test* este nesatisfăcut. Se poate realiza o comparare a variabilelor cu ajutorul constrângerilor predicative (se vor discuta ulterior). Pentru compararea variabilelor pot fi apelate de asemenea funcţii externe, care să realizeze această operaţie, în

orice mod dorit de utilizator. Poate fi legată de un CE *test* orice funcție externă, condiția fiind ca modul de introducere a argumentelor şi valorile returnate să nu fie acceptate de mediul CLIPS. De obicei cele mai utilizate sunt funcțiile predicative pe care le oferă CLIPS-ul (mai ales cele de comparare). De exemplu :

```
(test (>= ?nr 0))          ; număr pozitiv
```

Se pot însă realiza teste mult mai complexe, ca de exemplu verificarea dacă un număr este un întreg cuprins între două valori şi eventual mai mic decât valoarea unei variabile din cadrul aceleiaşi reguli. Verificarea se realizează automat prin funcția predicativă *integerp* .

```
(test    (and    (integerp ?nr)
                 (>=  ?nr  1)
                 (<=  ?nr  30)
                 (<   ?nr  ?max) ) )
```

Se poate realiza simplu o negare a expresiei de mai sus. Această expresie negată poate fi folosită de o regulă pentru a atenționa utilizatorul că valoarea variabilei ?nr nu îndeplineşte condițiile cerute.

```
(test    (or     (not (integerp ?nr))
                 (<  ?nr  1)
                 (>  ?nr  30)
                 (>=  ?nr  ?max) ) )
```

Pentru exemplificarea elementului condițional *test* se va relua exemplul care realiza permutările, de această dată nefolosind însă nici un câmp condițional ~ & | :

```
(deffacts obiecte_de_permutat
        (obiect  3)      (obiect  2)      (obiect  1))

(defrule permutare_de_trei_obiecte
        (obiect ?o1)
        (obiect ?o2)
        (obiect ?o3)
        (test (and (<> ?o2 ?o1)
                   (<> ?o3 ?o2 ?o1) ) )
        =>
        (printout  t  "(" ?o1 " " ?o2 " " ?o3 ")"  crlf) )
```

Funcția predicativă < > cere argumente de tip numeric şi din acest motiv dacă dorim să realizăm permutări de simboluri sau şiruri va trebui să folosim în locul acestei funcții predicative o altă funcție şi anume **neq**.

```
(deffacts obiecte_de_permutat
        (obiect  aer)
        (obiect  foc)
        (obiect  apa))
```

```
(defrule permutare_de_trei_obiecte
        (obiect ?o1)
        (obiect ?o2)
        (obiect ?o3)
        (test  (and  (neq ?o2 ?o1)
                     (neq ?o3 ?o2 ?o1) ) )
        =>
        (printout  t  "(" ?o1 " " ?o2 " " ?o3 ")"  crlf) )
```

Se remarcă folosirea formatului prefix (în locul celui infix, folosit în mod uzual) operatorul fiind plasat în fața operanzilor. Acest aspect, împreună cu detalierea funcțiilor predicative va fi discutat în cele ce urmează.

3.4. Funcții predicative

Denumire	**Specificații**
(**numberp**<expr>)	Returnează simbolul TRUE dacă argumentul funcției este de tip **float** sau **integer**.
(**floatp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este de tip **float**.
(**integerp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este de tip **integer**.
(**lexemep** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este de tip **string** sau **symbol**.
(**strigp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este de tip **string**.
(**symbolp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este de tip **symbol**.
(**evenp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este un **număr par**.
(**oddp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este un **număr impar**.
(**multifieldp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este o **valoare multicâmp**.
(**pointerp** <expr>)	Returnează simbolul TRUE dacă argumentul funcției este o **adresă externă**.
(**eq** <expr> <expr>+)	Returnează simbolul TRUE dacă primul argument al funcției **este egal** în valoare cu toate argumentele din subsecvență.
(**neq** <expr> <expr>+)	Returnează simbolul TRUE dacă primul argument al funcției **nu este egal** în valoare cu toate argumentele din subsecvență.

(= <num-expr> <num-expr>+) conversie integer în float	Returnează simbolul TRUE dacă primul argument al funcției **este egal** în valoarea cu toate argumentele din subsecvență.
(<> <num-expr> <num-expr>+) conversie integer în float	Returnează simbolul TRUE dacă primul argument al funcției **nu este egal** în valoare cu toate argumentele din subsecvență.
(> <num-expr> <num-expr>+) conversie integer în float	Returnează simbolul TRUE dacă fiecare argument A_i este **mai mare decât** argumentul A_{i+1} de după el (ordine strict descrescătoare).
(>= <num-expr> <num-expr>+) conversie integer în float	Returnează simbolul TRUE dacă fiecare argument A_i este **mai mare sau egal** decât arg. A_{i+1} de după el (ordine descrescătoare).
(< <num-expr> <num-expr>+) conversie integer în float	Returnează simbolul TRUE dacă fiecare argument A_i este **mai mic decât** argumentul A_{i+1} de după el (ordine strict crescătoare).
(<= <num-expr> <num-expr>+) conversie integer în float	Returnează simbolul TRUE dacă fiecare argument A_i este **mai mic sau egal** decât argumentul A_{i+1} de după el (ordine crescătoare).
(**and** <expr>+)	Returnează simbolul TRUE dacă fiecare argument este adevărat. Fiecare argument al funcției este evaluat de la stânga la dreapta.
(**or** <expr>+)	Returnează simbolul TRUE dacă orice argument este adevărat. Fiecare argument al funcției este evaluat de la stânga la dreapta.
(**not** <expr>)	Returnează simbolul TRUE dacă argumentul este evaluat ca fiind fals, altfel returnează FALSE.

O **funcție predicativă** este prin definiție orice funcție care returnează fie simbolul TRUE, fie simbolul FALSE. CLIPS-ul tratează, în logica predicativă, de fapt, orice valoare diferită de simbolul FALSE, ca fiind simbolul TRUE. O funcție predicativă poate fi de asemenea gândită ca returnând o valoare booleană. Funcțiile predicative pot fi de două tipuri: *funcție predefinită* şi *funcție definită de utilizator*. Funcțiile predefinite sunt deja oferite de CLIPS, pe când cele definite de utilizator sau *funcțiile externe* sunt scrise în C sau alte limbaje şi link-editate de mediul CLIPS. Au fost enumerate funcțiile predicative (de realizare a operațiilor logice boolene, de comparare a valorilor şi de testare a diferitelor tipuri de date). Se poate lua fiecare funcție în parte şi se poate verifica funcționarea ei după cum urmează:

```
CLIPS> (integerp 0.5)
FALSE
CLIPS> (eq 3 3.0)          ; eq  lucrează doar cu acelaşi tip de date
FALSE
CLIPS> (neq 3 3.0)         ;un tip de dată este integer, iar celălalt float
TRUE                       ;neq vede cele două valori ca fiind diferite
```

```
CLIPS> (= 3 3.0)
TRUE                        ; = face conversie integer la float
CLIPS> (<> 3 3.0)
FALSE                       ; <> face conversie integer la float
CLIPS> (or (> 4 3) (> 4 5))
TRUE
CLIPS> (and (>4 3) (>4 5))
FALSE
```

3.5. Intrebări

1. Ce este o constrângere literală ? Poate conține variabile sau wildcard-uri ?
2. Enumerați toate elementele condiționale şi constrângerile conective din CLIPS. Care sunt diferențele între un element condițional **and** şi o constrângere conectivă **&** (câmp condițional) ?
3. Elementele condiționale şi câmpurile condiționale se utilizează numai în RHS-ul regulii, numai în LHS-ul regulii sau în toată regula ?
4. Cu ajutorul cărui simbol se leagă de o variabilă una sau mai multe constrângeri conective?
5. In expresia următoare care este diferența între primul **&** (ampersand) şi cel de-al doilea ?

   (figura_geometrica (culoare ?c &~rosie &~albastra))
6. In regula următoare apare o eroare de programare. Care este aceea ?

```
(defrule exemplu_1
        (figura_geometrica (forma ?f) (culoare rosie &~neagra))
        =>
        (printout  t  "Gasit un " ?f  " de culoare rosie." crlf) )
```

7. La încărcarea regulii următoare CLIPS-ul va semnala o eroare ? Din ce cauză ?

```
(defrule exemplu_2
        (figura_geometrica (forma ?f) (culoare rosie | ?c))
        =>
        (printout  t  "Gasit " ?f  " de culoare " ?c "."crlf) )
```

8. Care este diferența între elementul condițional **and** explicit şi **and** implicit ?
9. O regulă în care s-a folosit elementul condițional **or** poate fi descompusă în două reguli mai mici ? Argumentați răspunsul.
10. Cum putem realiza două reguli care sunt reciproc exclusive ? Dați un exemplu.
11. Putem utiliza o variabilă legată de o valoare în cadrul unui CE **not** doar în interiorul acelui pattern sau şi în afara acestuia?
12. Care este eroarea care apare la încărcarea următoarei reguli ?

```
(defrule exemplu_3
        (not    (figura_geometrica  (forma ?f) (culoare rosie)) )
        =>
        (printout  t "Nu exista " ?f  "de culoare rosie." crlf))
```

13. În ce condiţii va fi adăugat automat unei reguli pattern-ul *initial-fact* ?
14. Care sunt diferenţele dintre elementele condiţionale **exists** şi **forall** dacă acestora li se aplica aceeaşi listă de fapte ?
15. Care sunt diferenţele între comenzile **dependets** şi **dependencies** ?
16. Ce returnează o funcţie predicativă ?

3.6. Aplicaţii

1. Realizaţi o regulă care foloseşte trei elemente condiţionale diferite şi trei constrângeri conective diferite.
2. Care este negata expresiei de mai jos ?

```
(test  (and  (or (integerp ?nr)
                 (floatp ?nr) )
             (>  ?nr  1)
             (<  ?nr  30) ) )
```

3. Compactaţi regulile care urmează într-o singură regulă folosind CE **and** şi **or**.

```
(defrule r1       (defrule r2       (defrule r3       (defrule r4
      (a)               (b)               (a)               (b)
      (d)               (c)               (e)               (c)
      => )              (e)               (f)               (d)
                        (f)               => )              => )
                        => )
```

4. Scrieţi o problemă care să determine valoarea literelor din următorul joc de cuvinte, astfel încât înlocuind literele A, D, E, G, J, M, R, S, T, U, cu o cifră (0-9) care le corespunde în mod unic, să se obţină o operaţie aritmetică corectă :

```
  ARAD+              SEND+
  SATU               MORE
  MARE               MONEY
 ARGES
JUDETE
```

5. Scrieţi un program care să stabilească pornind de la coordonatele a trei puncte dintr-un plan, tipul triunghiului pe care îl formează. Se vor lua în consideraţie cazurile de triunghi isoscel, echilateral, dreptunghic, oarecare. Se va face suplimentar şi verificarea dacă cele trei puncte nu sunt aflate toate pe o dreaptă. Programul poate fi testat cu următoarele tipuri de date:
   a) Puncte (0,0), (2,4) şi (6,0).
   b) Puncte (1,2), (4,5) şi (7,2).
   c) Puncte (0,0), (3,5.2) şi (6,0).

6. Considerăm în plan o mulțime de segmente ale căror extremități au coordonate întregi. Se cere să decideți dacă desenând aceste segmente se obține un contur poligonal (o linie poligonală închisă care nu se autointersectează) şi în caz afirmativ, să se determine numărul de vârfuri ale acestuia.
7. Pentru un şir de caractere dat, găsiți un subşir de lungime maximă care începe şi se sfârşeşte cu acelaşi caracter. Folosiți, dacă se poate o singură regulă.
8. Pentru problema drumului minim într-un graf neorientat (sau orientat) adăugați pentru fiecare arc o caracteristică numită cost şi astfel distanța minimă la două noduri să fie aceea care are un cost minim.
9. Modificați problema damelor (exemplificată mai jos) astfel încât aceasta să afişeze şi numărul curent al soluției. De asemenea schimbați dimensiunile tablei din 8x8, în dimensiuni mai mici (de ex: 5x5, 6x6).
10. Se dă o listă de elemente şi se cere să se găsească numărul de apariții în lista respectivă unui element specificat, sau să se afişeze elementul (elementele) care apare de cele mai multe ori.

3.7. Probleme rezolvate

1. **Problema culorilor**. Se dă o listă de culori primare. Ştiindu-se faptul că două culori primare se pot amesteca generând o altă culoare, vom realiza aceste modificări asupra listei inițiale, astfel încât în final vom obține o listă total modificată. Se va folosi pentru stocarea culorilor un fapt ordonat (listă) şi se vor genera regulile care combină culorile primare alăturate între ele.

```
(deffacts date_init
        (culori albastru rosu galben albastru galben rosu) )

(defrule portocaliu
        (or ?f <- (culori $?first ?c1&rosu ?c2&galben $?end)
            ?f <- (culori $?first ?c1&galben ?c2&rosu $?end))
        =>
        (retract ?f)
        (assert (culori $?first portocaliu ?c2 $?end)) )

(defrule verde
        (or ?f <- (culori $?first ?c1&albastru ?c2&galben $?end)
            ?f <- (culori $?first ?c1&galben ?c2&albastru $?end))
        =>
        (retract ?f)
        (assert (culori $?first verde ?c2 $?end)) )

(defrule violet
        (or ?f <- (culori $?first ?c1&rosu ?c2&albastru $?end)
            ?f <- (culori $?first ?c1&albastru ?c2&rosu $?end))
        =>
        (retract ?f)
```

```
        (assert (culori $?first violet ?c2 $?end)) )
```

După cum se observă, fiecare regulă de amestecare a două culori primare este compactizată cu ajutorul CE **or**, nefiind necesară scrierea a două reguli mai mici corespunzătoare celor două situații posibile de formare a culorii intermediare. De asemenea se poate observa inutilitatea variabilei ?c1, care poate fi eliminată, obținându-se astfel o simplă constrângere literală. După rularea programului se observă faptul că din toate culorile primare inițiale se păstrează ultima dintre acestea, pentru ştergerea acesteia fiind necesară realizarea unei alte reguli speciale. Deoarece ştergerea acestei culori trebuie realizată numai după ce au fost efectuate toate combinațiile, regula noastră ar trebui să se aprindă ultima. Acest lucru se poate realiza foarte simplu dacă îi schimbăm prioritatea (implicit ea este 0).

```
(defrule sterge_ultima_culoare
        (declare (salience -10))
        ?f1 <- (final)
        ?f2 <- (culori $?first ?ultim)
        =>
        (retract ?f1 ?f2)
        (assert (culori $?first)) )
```

Schimbarea priorității va fi exemplificată mai pe larg în capitolele următoare. Folosindu-ne de cele învățate până acum, putem aprinde în cele din urmă regula de ştergere procedând şi astfel:

```
(defrule final_combinari
        (not (exists (culori $?  rosu | galben | albastru  $? ?)))
        =>
        (assert (final)) )

(defrule sterge_ultima_culoare
        ?f1 <- (final)
        ?f2 <- (culori $?first ?ultim)
        =>
        (retract ?f1 ?f2)
        (assert (culori $?first)) )
```

Regula *final_combinări* verifică, dacă în afară de ultimul element (păstrat în variabila wildcard de un singur câmp) nu mai există culori primare (roşu | galben | albastru). Dacă acest lucru este adevărat este asertat un fapt *final*, care aprinde regula de ştergere a ultimului element (funcționează ca un flag).

2. **Problema damelor.** Problema damelor constă in aşezarea pe tabla de şah a unui număr de 8 dame, fără ca acestea să intre în conflict. Evident, nu vom putea avea două dame pe aceeaşi linie, coloană sau diagonală. Problema poate fi generalizată prin găsirea soluțiilor pentru table de dimensiuni mai mici. Problema nu este

rezolvată recursiv, fiind mai apropiată de modul de programare logic, decât de programarea procedurală:

```
(deftemplate dama
        (slot lin) (slot col))              ;coloana şi coloana pe care se poate afla dama

(deffacts initial
        (afisare . . . . . .)                   ;pt.afisare 8 puncte
        (numar 1 2 3 4 5 6 7 8) )               ;numarul pe care îl poate lua o linie
                                                ;sau o coloană(dimensiune tabla 8x8)
(defrule generare-combinatii
        (numar $? ?n1  $?)
        (numar $? ?n2  $?)
         =>
        (assert (dama (lin ?n1) (col ?n2)) ) )

(defrule cautare_solutie
      (dama (lin 1) (col ?c1))
      (dama (lin 2) (col ?c2&~?c1))
      (test (neq (abs (- ?c2 ?c1)) 1))
      (dama (lin 3) (col ?c3&~?c2&~?c1))
      (test  (and  (neq (abs (- ?c3 ?c1)) 2)
                   (neq (abs (- ?c3 ?c2)) 1)))
      (dama (lin 4) (col ?c4&~?c3&~?c2&~?c1))
      (test  (and  (neq (abs (- ?c4 ?c1)) 3)
                   (neq (abs (- ?c4 ?c2)) 2)
                   (neq (abs (- ?c4 ?c3)) 1)))
      (dama (lin 5) (col ?c5&~?c4&~?c3&~?c2&~?c1))
      (test  (and  (neq (abs (- ?c5 ?c1)) 4)
                   (neq (abs (- ?c5 ?c2)) 3)
                   (neq (abs (- ?c5 ?c3)) 2)
                   (neq (abs (- ?c5 ?c4)) 1)))
      (dama (lin 6) (col ?c6&~?c5&~?c4&~?c3&~?c2&~?c1))
      (test  (and  (neq (abs (- ?c6 ?c1)) 5)
                   (neq (abs (- ?c6 ?c2)) 4)
                   (neq (abs (- ?c6 ?c3)) 3)
                   (neq (abs (- ?c6 ?c4)) 2)
                   (neq (abs (- ?c6 ?c5)) 1)))
      (dama (lin 7) (col ?c7&~?c6&~?c5&~?c4&~?c3&~?c2&~?c1))
      (test  (and  (neq (abs (- ?c7 ?c1)) 6)
                   (neq (abs (- ?c7 ?c2)) 5)
                   (neq (abs (- ?c7 ?c3)) 4)
                   (neq (abs (- ?c7 ?c4)) 3)
                   (neq (abs (- ?c7 ?c5)) 2)
                   (neq (abs (- ?c7 ?c6)) 1)))
      (dama (lin 8)(col ?c8&~?c7&~?c6&~?c5&~?c4&~?c3&~?c2&~?c1))
      (test  (and  (neq (abs (- ?c8 ?c1)) 7)
                   (neq (abs (- ?c8 ?c2)) 6)
                   (neq (abs (- ?c8 ?c3)) 5)
                   (neq (abs (- ?c8 ?c4)) 4)
```

```
                (neq (abs (- ?c8 ?c5)) 3)
                (neq (abs (- ?c8 ?c6)) 2)
                (neq (abs (- ?c8 ?c7)) 1)))
    =>
    (printout t t "Apasa <Enter>...")
    (readline)
    (printout t 1- ?c1 ,2- ?c2 ,3- ?c3 ,4- ?c4 ,5- ?c5 ,6- ?c6 ,7- ?c7 ,8- ?c8
                crlf crlf)
    (assert (solutie ?c1 ?c2 ?c3 ?c4 ?c5 ?c6 ?c7 ?c8 )
            (linie 0)) )

(defrule afisare_linie
    (solutie $?first ?c $?)
    ?f <- (linie ?l)
    (test (eq (length$ $?first) ?l))
    (afisare $?a)
    =>
    (retract ?f)
    (printout t (implode$ (replace$  $?a  ?c  ?c  o)) t)
    (assert (linie (+ ?l 1)) ) )
```

În regula *generare-combinații* se generează cele 64 de câmpuri ale tablei pe care se poate situa dama la un moment dat. Regula este echivalentă cu execuția a două bucle **for** imbricate (una pe linie şi una pe coloană). În regula *căutare_ soluție* se fac teste dacă damele alese se află pe linii, coloane, sau diagonale diferite. CLIPS-ul este cel care va găsi toate soluțiile posibile. Ultima regulă *afişare_linie* înlocuieşte în linia de afişare punctul cu un caracter "o" (**replace$**) şi afişează şirul rezultat din transformarea multifield-ului în string (**implode$**).

3. **Găsirea minimului şi maximului** – Dat fiind un şir de numere, să se găsească minimul şi maximul acelui şir. Problemei va fi rezolvată în mai multe feluri.

```
(deffacts date_initiale
        (data  1  3  –5  7  2.3) )

(defrule min1
        (data  $?  ?v1  $?)
        (not (exists (data  $?  ?v2&:(<  ?v2 ?v1)  $?)))
        =>
        (printout  t  "Minim: " ?v1 t) )

(defrule min2
        (data $? ?v1 $?)
        (forall  (data  $?  ?v2  $?)
                 (test (>=  ?v2  ?v1)))
        =>
        (printout  t  "Minim: " ?v1 t) )
```

In regula *min2* şi în regula *min1* testul putea fi introdus legat direct de variabila ?v2 prin câmpul condițional predicativ **&:** (se va vedea în capitolul

următor) sau folosindu-se elementul condițional **test**. Exprimarea logică a condițiilor puse ar suna astfel : "Să nu existe un element mai mic" pentru prima regulă şi "Toate elementele să fie mai mari sau egale decât elementul ales minim".

```
(defrule min3
        (data $?d)
        =>
        (printout  t  "Minim: " (min (expand$ $?d))  t) )
```

În ultima regulă funcția **min** (funcție matematică a CLIPS-ului) acceptă doar parametri numerici de tip mono-câmp. O listă (o variabilă multicâmp) ar trebui "ruptă" (separată) în valori independente (de tip mono-câmp) pentru a putea fi acceptată ca parametru de funcția min, de acest lucru ocupându-se funcția **expand$**. Unica problemă care mai apare este faptul că funcția doreşte minim doi parametri şi în cazul în care lista ar avea un singur element sau ar fi vidă, programul se va întrerupe şi va genera o eroare. Funcția modificată ar arăta astfel:

```
(defrule min3
        (data $?d)
        (test (> (length$  $?d)  1))
        =>
        (printout  t  "Minim: " (min (expand$ $?d))  t) )
```

Pentru a realiza maximul nu trebuie să facem altceva decât să negăm toate condițiile puse din primele două reguli şi să folosim funcția **max** în ultima regulă.

```
(defrule max1
        (data  $?  ?v1  $?)
        (not (exists (data  $?  ?v2&:(>  ?v2 ?v1)  $?)))
        =>
        (printout  t  "Maxim: " ?v1 t) )

(defrule max2
        (data $? ?v1 $?)
        (forall  (data  $?  ?v2  $?)
                 (test (<= ?v2  ?v1)))
        =>
        (printout  t  "Maxim: " ?v1 t) )

(defrule max3
        (data $?d)
        (test (> (length$  $?d)  1))
        =>
        (printout  t  "Maxim: " (max (expand$ $?d))  t) )
```

Regulile realizate până în prezent au fost mai apropiate de programarea logică decât de cea procedurală. Vom realiza şi o variantă procedurală în care se inițializează minimul cu valoarea 0 şi apoi se ia din şir fiecare element pornind de la primul şi terminând cu ultimul (ca într-o buclă **for**) şi se compară cu minimul actual. Dacă acesta este mai mic decât minimul actual, se modifică valoarea

minimului, altfel încât valoarea acestuia rămâne aceeaşi. După ce s-au făcut toate modificările, o altă regulă va afişa minimul. În mod evident, pentru a afişa maximul, se neagă testul făcut pentru minim. Dacă se încearcă afişarea minimului şi a maximului în aceeaşi regulă se vor face două afişări: într-una s-a modificat doar o valoare dintre minim şi maxim; în a doua, valorile ambilor s-a modificat.

```
(deffacts date_initiale
        (min 0)
        (max 0)
        (data  1  3  –5  7  2.3) )

(defrule min
        (data  $?  ?x  $?)
        ?f <- (min  ?min&:(<  ?x  ?min))
        =>
        (retract ?f)
        (assert (min ?x) ) )

(defrule print_min
        (min ?min)
        =>
        (printout  t  “Minimul este:  “  ?min  t) )

(defrule max
        (data  $?  ?x  $?)
        ?f <- (max  ?max&:(>  ?x  ?max))
        =>
        (retract ?f)
        (assert (max ?x) ) )

(defrule print_max
        (max ?max)
        =>
        (printout  t “Maximul este: “  ?max  t) )

(defrule print_min_max
        (min ?min)
        (max ?max)
        =>
        (printout  t  "Minimul este:  "  ?min  "Maximul este: "  ?max  t) )
```

Explicația pentru faptul că se fac două afişări dintre care una la o fază intermediară şi nu se face doar o afişare finală constă în faptul că la fiecare schimbare a contextului regulii (asertarea sau ştergerea unor fapte) CLIPS-ul reacționează în mod dinamic modificând (reactualizând) toate regulile din agenda.

Pentru a remedia această problemă se poate schimba prioritatea regulii (regula cu prioritatea cea mai mică se aprinde ultima) şi în felul acesta s-ar putea elimina toate fazele intermediare. Soluțiile oferite pentru găsirea minimului şi maximului dintr-un şir, nu sunt unice, orice programator poate găsi oricând alte reguli care să rezolve aceeaşi problemă.

O altă problemă care poate duce la un rezultat incorect este inițializarea minimului şi maximului cu 0 (dacă toate elementele din şir sunt strict pozitive sau strict negative). Teoretic, minimul şi maximul ar trebui să fie egale cu primul element din şir. Din acest motiv faptele *min* şi *max* ar trebui să fie asertate într-o regulă şi nu inițializate cu o valoare într-o construcție *deffacts*.

```
(defrule  init_min_si_max
          (data  ?f  $?)
          =>
          (assert  (min ?f)
                   (max ?f))
```

4. **Operații cu mulțimi**. Se dau două mulțimi de elemente A şi B: efectuați intersecția, reuniunea şi diferența între cele două mulțimi şi afişați rezultatele.

```
(deffacts date_initiale
          (A   0  2  4  6  8  9)
          (B   0  5  9  11)
          (intersectie )    (reuniune )    (difA-B ) )  ;liste vide

(defrule intersectia
          (A  $?  ?n  $?)                    ; elementul ?n si in A si in B
          (B  $?  ?n  $?)
          ?f  <- (intersectie $?i)
          (test (eq  (member$  ?n  $?i)  FALSE))
          =>
          (retract ?f)
          (assert (intersectie  $?i  ?n)) )

(defrule reuniune
          (or  (A  $?  ?n  $?)                ; elementul ?n sau în A sau în B
               (B  $?  ?n  $?) )
          ?f <- (reuniune  $?r)
          (test (not (member$  ?n  $?r)))
          =>
          (retract ?f)
          (assert (reuniune  $?r  ?n)) )

(defrule  diferentaA-B
          (A  $?  ?n  $?)                    ; elementul ?n în A şi nu în B
          (B  $?b)
          ?f <- (difA-B  $?d)
          (test (and (not (member$  ?n  $?b))
                     (not (member$  ?n  $?d)) ))
          =>
          (retract  ?f)
          (assert (difA-B  $?d  ?n)) )
```

În testele regulilor condiția *(eq (member$?n $?i) FALSE)* este echivalentă cu *(not (member$?n $?b))*, deoarece funcția member$ returnează

poziția unui element într-un câmp, sau simbolul FALSE dacă elementul respectiv nu se află în listă. În toate regulile testul de unicitate (elementul să nu existe deja în listă) rezolvă problema buclei infinite (care ar fi apărut deoarece ştergem şi reasertăm acelaşi fapt în aceeaşi regulă). Acest lucru se poate uşor verifica comentând linia pe care se află testul. Regula *diferentaA-B* poate fi scrisă în mai multe moduri şi la fel ca şi la funcțiile *min* şi *max,* forma finală a unei reguli depinde de imaginația programatorului.

```
(defrule diferentaA-B
        (A  $?  ?n  $?)                    ; elementul ?n în A şi nu în B
        (not (exists (B  $?  ?n  $?)))
        ?f <- (difA-B  $?d )
        (test (not (member$  ?n  $?d)) )
        =>
        (retract  ?f)
        (assert (difA-B  $?d  ?n)) )

(defrule diferentaA-B
        (A  $?  ?n  $?)                    ; elementul ?n în A şi nu în B
        (B  $?b)
        ?f <- (difA-B  $?d)
        (test (and (neq  ?n  (expand$  $?b))
                   (not (member$  ?n  $?d)) ))
        =>
        (retract  ?f)
        (assert (difA-B  $?d  ?n)) )
```

Altă metodă de calculare a acestora ține de relațiile matematice. Diferența dintre două mulțimi (A-B) este mulțimea elementelor din A, care nu se află în intersecția acestora: (A-B) = A - (A&B)

Reuniunea a două mulțimi este mulțimea elementelor din diferențele (A-B) şi (B-A), precum şi din intersecție: (A+B) = (A-B) + (B-A) + (A&B)
sau: (A+B) = A + B – (A-B)

```
(defrule diferentaB-A
        (B  $?  ?n  $?)
        (not (exists (intersectie $?  ?n  $?)))
        ?f <- (difB-A  $?d&~:(member$  ?n  $?d))
        =>
        (retract ?f)
        (assert (difB-A  $?d ?n)) )
```

In regula *diferentaB-A* s-a legat un test de o variabilă folosindu-se operatorul '&' şi apoi cu '~' s-a negat un test predicativ (test care returnează TRUE sau FALSE şi care este specificat cu ':'). Acest test legat de o variabilă *$?d&~:(member$?n $?d)*, poate fi introdus în regulă şi prin elementul condițional **test**, avantajele folosirii testului legat de o variabilă constând într-un număr de pattern-uri mai mic pentru regulă.

```
(defrule reuniune
        (or  (A  $?  ?n  $?)
             (B  $?  ?n  $?) )
        (inters  $?i&~:(member$  ?n  $?i))
        ?f <- (reuniune $?r&~:(member$  ?n  $?r))
        =>
        (retract ?f)
        (assert (reuniune  $?r  ?n)) )

(defrule reuniune
        (difA-B  $?d1)
        (difB-A  $?d2)
        (inters  $?i)
        =>
        (assert (reuniune  $?d1  $?i  $?d2)) )
```