Cap. 2 VARIABILE

2.1. Definirea unei variabile

Ca orice alt limbaj de programare, CLIPS-ul are variabile ce pot lua valori. Valoarea pe care o poate lua o variabila poate fi unul din tipurile de date primitive (**float**, **integer**, **symbol**, **string**, **external address**, **fact address, instance name** şi **instance address**). Față de fapte, care sunt statice, conținutul unei variabile este dinamic, astfel încât valoarea asignată acesteia poate fi schimbată. Întotdeauna în CLIPS o variabilă este scrisă în sintaxa unui semn de întrebare "?", urmat de un câmp simbolic, care e numele variabilei. Formatul general este:

?<variable-name>

Numele variabilei poate fi orice simbol, cu restricția că trebuie să înceapă cu o literă (în caz contrar deşi nu se semnalează eroare, variabila este ca şi inexistentă !). Pentru o bună înțelegere a programului este indicat ca numele variabilelor să fie cât mai sugestive. De exemplu:

?vârsta ?loc_nastere ?clasa_apartenenta ?specie

De reținut faptul că spațiile nu trebuiesc puse între semnul de întrebare şi numele variabilei (ulterior se va vedea cum se foloseşte în CLIPS semnul de întrebare, singur, fără nume de variabilă). O variabilă poate fi folosită în LHS-ul regulii pentru a conține valoarea unui slot, apoi se pot face comparații ale acesteia cu alte valori din LHS, sau poate fi accesată din RHS-ul regulii pentru o afişare.

```
(deftemplate automobil
        (slot culoare)  (slot tip)  (slot ani)
        (multislot nume) )          ;numele proprietarului maşinii

(deffacts masini
        (automobil (culoare rosie)(tip Toyota) (nume Radu Vasile) (ani 0))
        (automobil (culoare neagra)(tip Mercedes)(nume Traian Basescu)(ani 2))
        (automobil (culoare rosie)(tip Dacia_1300)(nume Marian Petru)(ani 15))
        (automobil (culoare alba)(tip Trabant)(nume Iulian Petru)(ani 30)) )

(defrule afiseaza_tipul_si_vârsta_masinii
         (automobil (tip ?marca) (ani ?n))
         =>
         (printout  t  "Automobilul "  ?marca " are "  ?n  "ani."   crlf ) )
```

Se vor salva aceste construcții într-un fişier "automobil.clp" şi se va lansa în execuție programul, observându-se mesajele afişate pe monitor.

```
CLIPS> (clear)
CLIPS> (load "C:\\temp\\automobil.clp")
CLIPS> (reset)
CLIPS> (run)
Automobilul Trabant are o vechime de 30 ani.
```

Automobilul Dacia_1300 are o vechime de 15 ani.
Automobilul Mercedes are o vechime de 2 ani.
Automobilul Toyota are o vechime de 0 ani.

La lansarea în execuție, în agendă se pot observa patru aprinderi ale aceleiaşi reguli însă în ipostaze diferite (se poate da comanda *agenda* înainte de *run*). CLIPS-ul găseşte patru fapte diferite pe care le poate potrivi cu pattern-ul regulii. Se poate observa că ordinea apariției celor patru automobile este inversă, decât cea în care ele au fost introduse din construcția *deffacts* în lista de fapte la tastarea comenzii *reset* (se poate verifica cu *facts*). Ne-am putea explica acest lucru imaginându-ne lista de fapte ca o stiva LIFO, astfel încât ultimul fapt intrat este primul comparat cu pattern-ul regulii. Ordinea introducerii faptelor în lista este un aspect de care utilizatorul trebuie să țină seama atunci când doreşte afişarea într-o anumită ordine a faptelor. Inainte ca o variabilă să fie folosită, trebuie să-i asignăm o valoare. In cazul următor variabila neavând asignată nici o valoare, CLIPS-ul va afişa un mesaj de eroare.

```
(defrule exemplu
        =>
        (printout  t  ?x) )
```

[PRCCODE3] Undefined variable x refered in RHS of defrule.

2.2. Utilizarea multiplă a unei variabile

O proprietate foarte utilă si importantă pe care o au variabilele este aceea că pot fi folosite în mai multe locuri în LHS-ul regulii. La prima apariție variabilei îi este asignată o valoare pe care o va purta tot timpul în acea regulă. Alte apariții ale variabilei cu acelaşi nume în cadrul LHS-ului regulii urmăresc legarea tot cu prima valoare asignată variabilei. Continuând exemplul anterior adăugam construcțiile:

```
CLIPS> (undefrule *)              ;se şterg toate regulile
CLIPS> (deftemplate găseşte (slot culoare) )
CLIPS> (defrule cauta_masina
                (găseşte (culoare ?x))
                (automobil (culoare ?x) (tip ?marca))
                =>
                (printout  t  "Automobilul " ?marca " are culoarea " ?x  "." crlf))
```

In acestă regulă nou creată se vor afişa numai maşinile de o anumită culoare. Nu trebuie decât asertat un nou fapt *găseşte* şi CLIPS-ul va încerca să potrivească pattern-urilor regulii, faptele existente în lista de fapte. Dacă vor fi găsite perechi de fapte care să se potrivească atunci regula se va activa şi va apărea în agendă, aşteptând comanda *run* pentru a se aprinde.

```
CLIPS> (assert (găseşte (culoare rosie) ) )
<Fact-5>
```

```
CLIPS> (run)
Automobilul Dacia_1300 are culoarea rosie.
Automobilul Toyota are culoarea rosie.
CLIPS> (assert (găseşte (culoare neagra) ) )
<Fact-6>
CLIPS> (run)
Automobilul Mercedes are culoarea neagra.
CLIPS> (assert (găseşte (culoare verde) ) )
<Fact-7>
CLIPS> (run)
CLIPS>
```

In ultimul caz nu a fost găsit nici un automobil care are culoarea verde. După cum s-a mai spus CLIPS-ul va asigna la prima apariție a variabilei ?x valoarea slotului *culoare* al faptului *găseşte*, iar la a doua apariție va căuta în baza de fapte un fapt *automobil* care să aibă valoarea slotului *culoare* identificată cu valoarea asignată variabilei. Din acest motiv inversând ordinea celor două pattern-uri efectul aprinderii regulii poate fi uneori cu totul diferit.

Important de reținut este şi faptul că o variabilă se comportă în cadrul unei reguli ca orice variabilă locală dintr-un limbaj de programare (existența sa este legată de acea regulă, în momentul terminării execuției regulii variabila moare). Dacă în reguli diferite întâlnim variabile cu acelaşi nume ele nu au nici o legătură una cu alta. Se va vedea mai târziu că există şi posibilitatea definirii unor variabile globale, care există pe toată durata programului.

Din acest motiv nu se pot stabili în nici un caz conexiuni între reguli prin intermediul unor astfel de variabile locale. In concluzie putem întâlni o variabilă cu acelaşi nume în fiecare regulă, fără a fi afectată execuția programului.

2.3. Legarea adresei de fapt de o variabilă

După cum s-a afirmat o variabilă poate lua o valoare de tipul unei date primitive. In cele ce urmează valorile luate de către fapte vor fi de tipul **fact address** (adrese de fapt) astfel încât vom avea posibilitatea retractării, modificării şi duplicării unui fapt în cadrul unei reguli.

Executarea unor astfel de operații în cadrul regulii este mult mai uşoară şi mai utilă decât atunci când se realiza la nivelul prompter-ului (top level). La nivelul prompter-ului aceste operații se realizau cu ajutorul indicelui de fapt, în cadrul unei reguli se leagă adresa faptului ce îndeplineşte condițiile pentru a fi şters (se potriveşte cu pattern-ul regulii) de o variabilă, care urma a fi apelată în RHS-ul regulii de acțiunea ce se execută asupra faptului.

Legarea variabilei de adresa de fapt se face cu ajutorul operatorului de legare din pattern (**pattern binding** operator) care este şirul de caractere "<-" (o săgeata orientata spre stânga, formată din semnul 'mai mic decât' şi semnul 'minus'). La fişierul "automobil.clp" creat anterior construim o regulă care să schimbe culoarea unei maşini.

```
(deftemplate culoare
        (slot old)  (slot new) )

(defrule schimba_culoare
        (culoare (old ?old) (new ?new))
        ?f <- (automobil (culoare ?old) (tip ?marca))
        =>
        (retract ?f)                                        ;??!!
        (modify ?f (culoare ?new))
        (printout  t   ?marca " are acum culoarea " ?new crlf) )
```

```
CLIPS> (clear)
CLIPS> (reset)
CLIPS> (assert (culoare (old neagra) (new albastra)))
<Fact-5>
CLIPS> (watch facts)
CLIPS> (run)
<== f-2 (automobil (culoare neagra) (tip Mercedes) (numeTraianBasescu) (ani 2))
==> f-6 (automobil (culoare albastra) (tip Mercedes) (numeTraianBasescu) (ani 2))
Automobilul are acum culoarea albastra.
```

Primul pattern al regulii *schimba_culoare* determinat dacă există vreo culoare ce se doreşte a fi schimbată, al doilea pattern găseşte în cazul în care există maşina de culoarea ce se doreşte a fi schimbată. Adresa faptului *automobil* ce va fi modificat este legat de variabila ?f, folosită apoi în RHS pentru a retracta faptul şi pentru a-l modifica. Se observă că deşi faptul *automobil* a fost retractat, putem avea acces în continuare la valorile sloturilor *nume, ani, tip* . Variabila *?marca* care păstrează valoarea slotului *tip* este folosită pentru o afişare pe ecran (variabilele locale trăiesc pe durata întregii reguli).

In exemplul anterior se poate observa apariţia câmpului *culoare* în două ipostaze; în prima reprezintă numele unui slot al construcţiei deftemplate *automobil*, în a doua este chiar numele unei construcţii deftemplate. In CLIPS un câmp în funcţie de contextul în care el apare, este identificat în mod unic ca fiind numele unei construcţii, al unui slot, al unei variabile, al unei funcţii etc. Din acelaşi motiv s-au putut folosi câmpurile *old* şi *new* atât ca nume de sloturi, cât şi ca variabile. CLIPS-ul nu permite realizarea a două construcţii deftemplate sau defrule sau deffacts cu acelaşi nume, în acest caz fie returnând un mesaj de eroare, fie realizând o suprascriere. Totuşi poate exista o construcţie deftemplate şi una defrule care au acelaşi nume (orice alte asocieri sunt permise). Am putea de exemplu introduce fără a avea nici o problemă regula:

```
(defrule culoare      =>     )
```

In cazul variabilelor, după cum s-a mai spus doar la prima apariţie se face legarea variabilei de o valoare, în toate celelalte apariţii realizându-se comparaţii cu valoarea asociata variabilei. In schimb la sloturi, putem avea mai multe construcţii deftemplate care să aibă acelaşi nume pentru unul din sloturi. De exemplu pentru:

```
(deftemplate om (slot nume) (slot culoare ))        ;culoarea pielii
```

CLIPS-ul identifică în funcție de context dacă slotul *culoare* aparține construcției deftemplate *om* sau a construcției *automobil*. Unica problema apare doar pentru programator, care poate face confuzii şi poate uşor greşi dacă nu e atent. In mod intenționat în cadrul regulii *schimba_culoare* a fost introdusă şi o comanda inutilă şi anume *(retract ?f)*, deoarece comanda *modify* realizează automat şi retractarea faptului pe care îl modifică. Dacă se doreşte păstrarea comenzii *retract* poate fi foarte bine folosită şi comanda *assert* însă trebuiesc specificate valorile tuturor sloturilor, în caz contrar CLIPS-ul asociindu-le valoarea *nil*. De exemplu pentru asertarea următoare:

```
(assert (culoare))
```

CLIPS-ul va introduce în baza de fapte *(culoare (old nil) (new nil))*. Trebuie reținut faptul că nu putem avea în acelaşi timp un fapt ordonat şi un fapt deftemplate cu acelaşi nume. De exemplu am obține o eroare dacă am încerca să asertăm:

```
(assert (culoare maro))                    ;sau
(assert (culoare (old galben) (new mov)))
```

In unele situații programatorul poate realiza fără să vrea o *buclă infinită*, acest lucru realizându-se atunci când un fapt nou creat de o regulă se potriveşte cu pattern-ul acesteia , regula fiind introdusă în agendă, după care aprinzându-se dă naştere unui nou fapt ,care o reaprinde şi tot aşa la nesfârşit. De exemplu:

```
(defrule schimba_culoarea_Daciei
        (nuanta ?x)
        ?f <- (automobil (tip Dacia_1300))
        =>
        (modify ?f (culoare  ?x)) )
CLIPS> (assert (nuanta gri))
CLIPS> (run 100)
```

Nu s-a putut aserta un fapt *(culoare gri)*, deoarece ar fi fost un fapt ordonat, care nu s-ar fi potrivit cu şablonul cerut de construcția deftemplate *culoare* anterior creată. Aceasta ar putea fi însa ştearsă fie cu *(undeftemplate culoare)* sau cu un *(clear)* urmat de un *(reset)*.

Dacă se vor da înainte de *(run 100)* comenzile *(watch facts)*, *(watch rules)* ,*(watch activations)* se poate observa cum regula se reaprinde de 100 de ori. In acest caz soluția este foarte simplă, nu trebuie decât să ştergem faptul prin care s-a specificat culoarea şi în acest fel pentru primul pattern al regulii nu va mai exista nici un fapt care să se potrivească cu acesta. Se realizează modificarea următoare:

```
(defrule schimba_culoarea_Daciei
        ?f1 <- (nuanta ?x)
        ?f2 <- (automobil (tip Dacia_1300))
        =>
        (retract ?f1)
        (modify ?f2 (culoare  ?x)) )
```

Realizarea unei construcții *defrule* cu un nume identic cu o altă construcție *defrule* anterioară, duce la înlocuirea acesteia (se suprascrie). De acesta dată se folosesc doua variabile ?f1 si ?f2 , fiecare luând valoarea unei adrese a unuia din cele două fapte ce se potrivesc cu pattern-ul regulii. In acest caz nu se mai realizează efectul de *bucla infinita*. Se dau comenzile:

```
CLIPS> (reset)
CLIPS> (watch facts)
CLIPS> (run)
<==  f-6  (nuanta gri)
<==  f-3 (automobil (culoare rosie) (tip Dacia_1300) (nume Marian Petru) (ani 15))
==>  f-7 (automobil (culoare gri) (tip Dacia_1300) (nume Marian Petru) (ani 15))
```

Pentru a întrerupe o bucla infinită datorată comenzii *run* fără argumente se foloseşte Ctrl-C sau orice altă comandă prin care se poate opri execuția calculatorului ce rulează CLIPS.

2.4. Variabila liberă de un singur câmp

Interpretarea pe care o dă CLIPS-ul unei variabile libere de un singur câmp (**single-field wildcard**) este de simbol care înlocuieşte o parte a unei entități pattern. Utilitatea folosirii unui wildcard într-un pattern se observă mai ales atunci când se doreşte testarea existenței unui câmp într-un slot (de obicei un multislot). Variabila liberă se notează cu un semn de întrebare '?' după care însă nu mai urmează nici un nume de variabilă (de aici şi denumirea de liberă). Deoarece nu avem un nume prin care să o putem apela şi diferența de alte variabile (putem avea oricâte variabile libere dorim), un wildcard nu este folosit decât pentru a înlocui un câmp. Să presupunem că avem un număr de înmatriculare la maşină declarat ca multislot într-o construcție deftemplate. Pentru a obține doar oraşul (partea din mijloc) procedam astfel:

```
(nr_inmatr  05  IS  ARG)                    (nr_inmatr  ?  ?oras  ?)
```

Dacă se doreşte doar numărul de serie (ultimul număr) primele două câmpuri vor fi înlocuite cu semne de întrebare, doar ultimului câmp fiindu-i asociată o variabilă obişnuită. După cum se observă valorile unui multislot care nu au nici o importanță pentru noi în cadrul unei reguli pot fi înlocuite cu un wildcard. In construcția deftemplate *automobil* am declarat ca multislot numele proprietarului. S-au introdus numai nume de persoane formate din două câmpuri (nume şi prenume), dar s-ar putea proceda similar şi atunci când am avea trei câmpuri (şi inițiala, de exemplu). Astfel, se va realiza o regulă care afişează tipul maşinii, pentru o persoană de la care nu cunoaştem decât numele mic (prenume):

```
(defrule afiseaza_masina_persoana
        (masina_lui ?prenume)
        (automobil (nume ?prenume  ?) (tip ?marca))
        =>
        (printout  t  ?prenume  " are un automobil " ?marca crlf ))
```

```
CLIPS> (reset)
CLIPS> (assert (masina_lui Iulian))
CLIPS> (run)
Iulian are un automobil Trabant.
```

In CLIPS, în momentul în care programatorul nu specifică toate sloturile unui fapt dintr-un pattern, se foloseşte automat câte un wildcard pentru valoarea fiecărui câmp nespecificat. Astfel în exemplul nostru pentru faptul *automobil* sunt specificate doar sloturile *nume* şi *tip* şi CLIPS-ul mai adaugă pentru celelalte sloturi ale pattern-ului încă două wildcard-uri. Astfel faptul este convertit în :

```
(automobil (culoare ?) (tip ?marca) (nume ?prenume ?) (ani ?))
```

2.4.1. Problema blocurilor

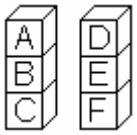
Pentru a demonstra mai bine legarea variabilelor se va realiza un program de mutare a blocurilor. Acest exemplu poate fi asociat cu joaca unui copil care ia dintr-un şir de cuburi puse unul peste altul câte un cub şi îl pune jos pe podea, începând de la cel mai de sus şi terminând cu ultimul. Acest model de inteligență primară, poate fi aplicat şi într-o întreprindere unde un robot cu un singur braț poate executa operații de acest gen. De aici se poate merge mai departe prin diversificarea operațiilor de executat . Forma de aranjare a blocurilor ne duce imediat cu gândul la o stivă de tip LIFO, unde un cub poate fi pus (stivuit) doar deasupra altuia sau eventual dat jos.

Scopul acestei probleme a blocurilor este aranjarea într-o anumită configurație dată, acest lucru fiind realizat într-un număr minim de mutări. Pentru exemplul pe care îl vom realiza sunt date o serie de restricţii cum ar fi:

- este permis doar un scop inițial de mutare a unui bloc peste un alt bloc.
- oricare ar fi scopul propus, acesta nu trebuie să fie deja realizat de aşezarea inițială a pieselor.

Rezolvarea acestei probleme se va realiza într-un număr optim de mutări procedând în felul următor. Dacă scopul inițial este mutarea blocului x peste blocul y, atunci sunt mutate pe podea toate blocurile aflate peste blocul x (dacă acestea există); apoi sunt mutate pe podea toate blocurile aflate peste blocul y (dacă există); în cele din urma mutând blocul x peste blocul y. Prin verificarea datelor inițiale, a doua restricție practic înlătură situația imposibilă de a muta un bloc peste un altul, atunci când acest lucru este deja realizat.

Se va începe prin reprezentarea configurațiilor blocurilor după care se va testa programul. Astfel în figura următoare avem două stive, prima stivă fiind formată din blocul A , aflat peste blocul B, peste blocul C şi a doua stivă e formată din blocul D, aflat peste blocul E, peste blocul F.



Scopul stabilit este de a muta blocul C peste blocul E.

Din acest moment trebuiesc determinate regulile ce vor ajuta la rezolvarea problemei pas cu pas. Pentru a realiza mutarea blocului C peste blocul E, cea mai simplă soluție ar fi dacă am putea face acest lucru direct, fără a mai avea deasupra lor vreun bloc. Pseudocodul pentru o asemenea regulă ar fi următorul:

RULE *MUTA-DIRECT*
IF Scopul este de a muta blocul ?de_sus în vârful blocului ?de_jos şi
blocul ?de_sus este în vârful stivei sale şi
blocul ?de_jos este în vârful stivei sale,
THEN Muta blocul ?de_sus peste blocul ?de_jos.

In cazul nostru regula *muta-direct* nu poate fi folosită deoarece peste blocul C se afla blocurile A şi B; iar peste blocul E se află blocul D. Inainte de a efectua această regulă ar trebui mutate blocurile A, B şi D pe podea. Acest lucru este simplu de realizat, deoarece problema blocurilor nu a cerut o reaşezare a blocurilor într-o stivă şi nu avem nici o restricție privind numărul de blocuri ce pot fi puse pe podea, acesta fiind practic nelimitat. Avem nevoie de două reguli; una de mutare a blocurilor ce se afla în stiva blocului ce trebuie mutat peste; alta de mutare a blocurilor ce se află în stiva blocului peste care trebuie mutat.

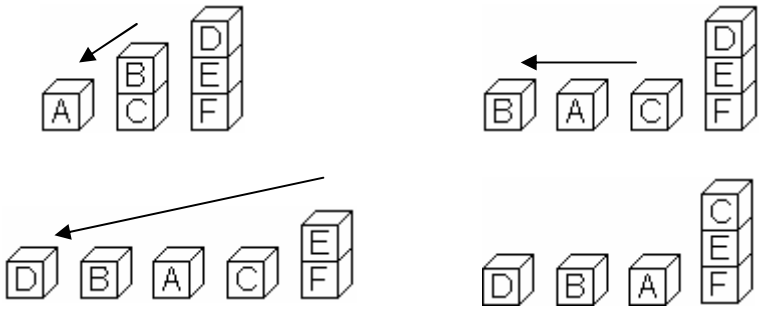
RULE *STERGERE-PT-BLOCUL-DE-DEASUPRA*
IF Scopul este sa mut blocul ?x si
blocul ?x nu este în vârful stivei sale si
blocul ?de_deasupra este în vârful lui ?x,
THEN Un nou scop este de a muta blocul ?de _deasupra pe podea.

RULE *STERGERE-PT-BLOCUL-DE-DESUBT*
IF Scopul este sa mut alt bloc peste blocul ?y si
blocul ?y nu este in vârful stivei sale si
blocul ?de_deasupra este in vârful lui ?y,
THEN Un nou scop este de a muta blocul ?de _deasupra pe podea.

Regula *stergere-pt-blocul-de-deasupra* va lucra pentru a goli blocurile aflate peste blocul C. Prima oară se va stabili că blocul B trebuie mutat pe podea (nu este posibil încă), dar acest nou scop va activa regula din nou care de aceasta data va stabili ca trebuie mutat pe podea şi blocul A. Regula *stergere-pt-blocul-de-desubt* va stabili că trebuie mutat şi blocul D pe podea. Din acest moment avem ca scopuri secundare mutarea blocurilor A, B, D pe podea, lucru ce poate fi realizat direct pentru blocurile A, B şi apoi pentru blocul D. Regula se aseamănă cu *muta-direct* cu deosebirea ca acum nu se va mai muta peste un bloc ci se va muta pe podea. Pseudocodul regulii ar putea fi:

RULE *MUTA-PE-PODEA*
IF Scopul este sa muți blocul ?de_deasupra si
blocul ?de_deasupra este in vârful stivei sale,
THEN Muta blocul de deasupra pe podea.

Ordinea mutării blocurilor este reprezentată în figurile următoare:



Odată mutat pe podea, un alt bloc devine vârf în stiva sa şi de acest lucru va trebui să se țină cont atunci când se vor scrie regulile. După ce am scris regulile în pseudocod, putem trece la faptele ce vor fi folosite de reguli pentru a se putea aprinde. Informațiile în care se va specifica pentru fiecare bloc în parte ce anume are deasupra sa şi ce are sub el vor fi descrise de construcția deftemplate următoare:

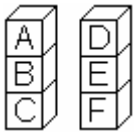
```
(deftemplate deasupra_lui
        (slot  sus)
        (slot  jos))
```

Faptele care sunt descrise de acest template sunt următoare:

```
(deasupra_lui    (sus A) (jos B) )
(deasupra_lui    (sus B) (jos C) )
(deasupra_lui    (sus D) (jos E) )
(deasupra_lui    (sus E) (jos F) )
```

De asemenea este foarte important să ştim dacă un bloc este în vârful stivei sau dacă este la coada ei; se vor adaugă şi faptele:

```
(deasupra_lui    (sus nimic) (jos A) )
(deasupra_lui    (sus C) (jos podea) )
(deasupra_lui    (sus nimic) (jos D) )
(deasupra_lui    (sus F) (jos podea) )
```



Se indică în acest fel foarte clar prin sloturile *(sus nimic)* vârfurile celor două stive şi anume blocurile A si D, iar prin sloturile *(jos podea)* capetele inferioare ale stivelor, blocurile C şi F. Problema care apare este faptul că regulile ar putea confunda simbolurile *nimic* şi *podea* cu blocurile din stive de aceea ar trebui ca acestea să fie clar specificate de fapte.

```
(bloc A)        (bloc B)        (bloc C)
(bloc D)        (bloc E)        (bloc F)
```

In final trebuie specificate scopurile de mutare a blocurilor, pentru a putea fi apoi prelucrate de către reguli. Aceste scopuri pot fi descrise de următorul deftemplate:

```
(deftemplate scop (slot muta) (slot peste) )
```

, iar în acest caz scopul inițial este:

```
(scop (muta C) (peste E) )
```

Odată definite toate faptele şi deftemplate-urile, putem realiza o configurare inițială a datelor din problema blocurilor pe care să o descriem cu următoarea construcție deffacts:

```
(deffacts stare-initiala
        (bloc A)        (bloc B)        (bloc C)
        (bloc D)        (bloc E)        (bloc F)
        (deasupra_lui  (sus nimic) (jos A))
        (deasupra_lui  (sus A) (jos B))
        (deasupra_lui  (sus B) (jos C))
        (deasupra_lui  (sus C) (jos podea))
        (deasupra_lui  (sus nimic) (jos D))
        (deasupra_lui  (sus D) (jos E))
        (deasupra_lui  (sus E) (jos F))
        (deasupra_lui  (sus F) (jos podea))
        (scop (muta C) (peste E)) )
```

Regula *muta-direct* este scrisa după cum urmează:

```
(defrule muta-direct
        ?scop <- (scop (muta ?bloc1) (peste ?bloc2) )
        (bloc ?bloc1)                                   ;??!!
        (bloc ?bloc2)                                   ;??!!
        (deasupra_lui (sus nimic) (jos ?bloc1) )
        ?stiva1 <- (deasupra_lui (sus ?bloc1) (jos ?bloc3) )
        ?stiva2 <- (deasupra_lui (sus nimic) (jos ?bloc2) )
        =>
        (retract ?scop ?stiva1 ?stiva2)
        (assert (deasupra_lui (sus ?bloc1) (jos ?bloc2) )
                (deasupra_lui (sus  nimic) (jos ?bloc3) ) )
        (printout  t  ?bloc1 " mutat peste " ?bloc2 "." crlf ) )
```

Primele trei pattern-uri determinate dacă există un scop de a muta un bloc peste un alt bloc (pattern-urile doi şi trei se asigură că această regulă nu va procesa o mutare la podea). Al patrulea şi al şaselea pattern verifică dacă cele două blocuri sunt în vârful stivelor din care fac parte. Al cincilea şi al şaselea pattern pregătesc informațiile necesare pentru modificarea celor două stive, după ce se vor muta blocurile unul peste celalalt. In RHS se şterge scopul (acesta a fost realizat) şi cele două stive şi se asertează noile stive modificate, afişându-se mutarea realizată.

Regula *muta-pe-podea* este implementată astfel:

```
(defrule muta-pe-podea
        ?scop <- (scop (muta ?bloc1) (peste podea) )
        (bloc ?bloc1)                                          ;??!!
        (deasupra_lui (sus nimic) (jos ?bloc1) )
        ?stiva <- (deasupra_lui (sus ?bloc1) (jos ?bloc2) )
        =>
        (retract ?scop ?stiva)
        (assert (deasupra_lui (sus ?bloc1) (jos nimic) )
                (deasupra_lui (sus nimic) (jos ?bloc2) ) )
        (printout  t  ?bloc1 " mutat pe podea." crlf) )
```

Regula este aproape identică cu *muta-direct* cu excepțiile că acum nu mai trebuie verificat dacă simbolul *podea* este bloc şi că nu se mai modifică decât o stivă. Se poate observa faptul ca regula *muta-pe-podea* are nevoie pentru a se aprinde de faptul *(scop (muta ?bloc) (peste podea))*. Acest fapt va fi furnizat de regulile următoare, care stabilesc care dintre blocuri trebuiesc mutate la podea pentru a putea efectua mutarea finală între cele două blocuri :

```
(defrule stergere-pt-blocul-de-deasupra
        (scop (muta ?bloc1) )
        (bloc ?bloc1)                                          ;??!!
        (deasupra_lui (sus ?bloc2) (jos ?bloc1) )
        (bloc ?bloc2)
        =>
        (assert (scop (muta ?bloc2) (peste podea) ) ) )

(defrule stergere-pt-blocul-de-desubt
        (scop (peste ?bloc1) )
        (bloc ?bloc1)                                          ;??!!
        (deasupra_lui (sus ?bloc2) (jos ?bloc1) )
        (bloc ?bloc2)
        =>
        (assert (scop (muta ?bloc2) (peste podea) ) ) )
```

Ultimele două reguli de ştergere a elementelor din stivă, verifică în cele patru pattern-uri, dacă blocul specificat în *scop* (pe care fie trebuie să-l mutăm peste alt bloc, fie trebuie mutat un alt bloc peste el), mai are deasupra sa alte elemente. Dacă da, atunci se stabileşte scopul secundar de mutare al blocului de deasupra. In locurile unde s-au pus comentarii formate din "??!!" liniile respective pot fi şterse fără a se afecta funcționarea programului. Ele nu sunt cu totul inutile, ajutând doar la înțelegerea programului.

Programul este acum complet şi poate fi rulat. Se includ într-un fişier *"bloc.clp"* construcțiile deftemplate *scop* şi *deasupra_lui*; construcția deffacts *stare-initiala* şi regulile *muta-direct, muta-pe-podea, stergere-pt-blocul-de-deasupra, stergere-pt-blocul-de-desubt*. Programul ar trebui să afişeze următoarele:

```
CLIPS> (clear)
```

```
CLIPS> (load "C:\\temp\\bloc.clp")
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (run)
A mutat pe podea.
B mutat pe podea.
D mutat pe podea.
C mutat peste E.
CLIPS>
```

Se recomandă pentru început o rulare pas cu pas a programului pentru a observa modificările care apar (în varianta pentru DOS se va da comanda *(run 1)* în timp ce *watch facts*, *watch activations* şi *watch rules* vor fi active; în varianta pentru Windows se poate da comanda *Step* din meniul *Execution* sau *Ctrl+T* şi vor fi activate din meniul *Window* ferestrele care ne interesează).

Acest exemplu ilustrează trecerea pas cu pas prin toate etapele de proiectare a unui program în CLIPS, de la înţelegerea cerinţelor problemei şi scrierea pseudo-codului, la stabilirea tipurilor de fapte necesare şi construirea deftemplate-urilor şi a deffacts-urilor, terminând cu realizarea regulilor din translarea pseudo-regulilor în CLIPS.

In general un sistem expert cere mult mai multe prototipuri şi dezvoltări iterative până se ajunge la o formă finală. Nu întotdeauna este posibilă determinarea celei mai bune metode de reprezentare a faptelor şi a tipurilor de reguli de care avem nevoie pentru a construi un sistem expert.

2.5. Variabilele libere multicâmp

Variabilele libere multicâmp (în literatura de specialitate multifield wildcards) sunt variabile care pot înlocui zero sau mai multe câmpuri într-un pattern şi sunt reprezentate în CLIPS prin două caractere $?. Ca şi variabilele libere de un singur câmp ele nu au un nume de variabilă şi deci nu pot fi referite în RHS-ul unei reguli. In CLIPS operatorul $ are o semnificaţie specială în LHS, tocmai el specificând faptul că pot fi înlocuite zero sau mai multe câmpuri. Marele avantaj al folosirii acestor variabile se observă atunci când nu avem un număr fix de câmpuri într-un multislot. Să presupunem că avem un multislot *nume_persoana* unde avem numele, iniţiala tatălui şi unul sau mai multe prenume; şi că dorim să aflăm ultimul câmp. Dacă am fi avut la dispoziţie numai variabilele libere simple nu am fi putut şti de câte anume avem nevoie necunoscând exact numărul câmpurilor şi deci nici cel al câmpului dorit de noi.

```
(deftemplate persoana
    (multislot nume)  (slot ocupatie)  (slot etate) )

(deffacts oameni
    (persoana (nume Traian G. Basescu)(ocupatie ministru)(etate 35))
    (persoana (nume Marian Ionut T. Petru) (ocupatie programator)(etate 23))
    (persoana (nume Ramona L. Doru) (ocupatie student) (etate 19)) )
```

```
(defrule numele_persoanei
      (persoana (nume  $?  ?nume ) )
      =>
      (printout  t  "Gasit persoana cu numele : " ?nume "." crlf ) )

(defrule initiala_tatalui
      (persoana (nume  $?  ?init  ? ) )
      =>
      (printout  t  "Gasit inițiala tatalui :  "  ?init crlf ) )

(defrule numele_persoanei_si_initiala_tatalui
      (persoana (nume  $?  ?init  ?nume ) )
      =>
      (printout  t  "Persoana  " ?nume " are inițiala tatalui " ?init crlf) )
```

In prima regulă variabila liberă multicâmp $? înlocuieşte toate câmpurile până la ultimul câmp; iar în a doua regulă înlocuieşte toate câmpurile până la penultimul câmp (care reprezintă inițiala tatălui), fiind necesară apoi introducerea unei variabile libere simple pentru înlocuirea numelui persoanei. Efectul folosirii într-un multislot în acelaşi timp două variabile libere multicâmp este diferit:

```
(nume  $?  ?camp  $?)
```

Variabila *?câmp* poate lua valoarea oricărui câmp din multislotul *nume*, deoarece primul wildcards poate înlocui zero sau mai multe câmpuri, iar al doilea poate şi el înlocui zero sau mai multe câmpuri. In acest caz regula va fi activată de atâtea ori câte câmpuri are multislotul *nume*. Se produce acelaşi efect ca şi atunci când de exemplu s-ar căuta un fişier care în numele său are şirul *cmp* şi s-ar da de către utilizator în locul numelui complet combinația **cmp**.

```
(defrule afiseaza_campurile_numelui
      (persoana (nume  $?  ?câmp  $? ) )
      =>
      (printout  t  ?câmp crlf ) )
CLIPS> (agenda)
```

Să presupunem că dorim afişarea tuturor persoanelor care au şi automobil (se folosesc construcțiile deffacts şi deftemplate din exemplul *"automobil.clp"*). Putem adauga acum toate construcțiile într-un singur fişier :

```
CLIPS> (load "C:\\temp\\automobil.clp"
CLIPS> (reset)
CLIPS> (undefrule *)              ;se şterg regulile anterioare
CLIPS> (defrule afiseaza_automobilul_unei_persoane
          (automobil (nume  ?prenume  ?nume) (tip ?marca) )
          (persoana (nume  $?  ?prenume  $?  ?  ?nume) )
          =>
          (printout  t  ?prenume " " ?nume " are automobil " ?marca crlf ))
CLIPS> (run)
```

In regula noastră la prima apariție a variabilelor *?nume* şi *?prenume* acestora li se vor asocia valorile găsite în multislotul *nume* al unui template

automobil ,urmând mai apoi să se caute dacă în lista de fapte există vreun template *persoana* care sa aibă în multislotul sau câmpuri cu aceleaşi valori. După cum se observă se caută variabila *prenume* în toate câmpurile de la început (mai puţin ultimele două: initiala_tatalui şi numele persoanei), astfel încât pentru o persoana care are de exemplu trei prenume vor fi verificate toate cele trei prenume.

CLIPS-ul foloseşte automat această variabilă pentru a înlocui într-un pattern un multislot care nu a fost specificat. Astfel dacă o regulă ar folosi template-ul *persoana* specificând doar sloturile *ocupaţie* şi *etate* :

```
(persoana (ocupatie ?loc) (etate ?ani) )
```

va fi realizată automat de către CLIPS o conversie în forma următoare:

```
(persoana (nume  $?) (ocupatie ?loc) (etate ?ani) )
```

2.6. Variabilele multicâmp

Diferenţa între o variabilă multicâmp şi o variabilă liberă muticâmp este aceea că după secvenţa de caractere **$?**, aceasta este precedat de un nume de variabilă cu ajutorul căreia putem să o referim în RHS-ul unei reguli. Dacă în LHS-ul regulii este obligatorie existenţa operatorului **$** care specifică faptul că avem o variabilă muticâmp şi nu una simplă, în RHS-ul regulii se poate plasa variabila doar precedat de operatorul **?**, CLIPS-ul ştiind în acel moment faptul că variabila are asociate mai multe valori şi nu una singură. Dacă utilizatorul însă consideră că acest lucru l-ar putea deruta, el poate amplasa în permanenţă ambii operatori pentru o astfel de variabilă.

Să presupunem că dorim în acest moment afişarea tuturor datelor despre persoanele introduse cu ajutorul construcţiei *deffacts* , cu specificaţia că se doreşte afişarea tuturor câmpurilor din multislotul *nume*. Ţinând cont de faptul că numărul total de câmpuri este variabil (nu se ştiu exact câte prenume are persoana), nu putem folosi variabile simple, de aceea se foloseşte o variabilă multicâmp :

```
(defrule afisare_date_persoana
        (persoana  (nume  $?nume) (ocupatie ?loc) (etate ?ani) )
        =>
        (printout  t  $?nume " are ocupatia " ?loc " si vârsta de " ?ani " ani."  t))
```

Se poate observa faptul că la afişare o variabilă multicâmp este cuprinsă între paranteze rotunde, acesta fiind modul de afişare în CLIPS pentru un multifield. Dacă am dori stocarea prenumelor într-o singură variabilă multifield am putea proceda astfel:

```
(defrule afisare_date_persoana
        (persoana (nume  $?prenume  ?init  ?nume) (ocupatie ?loc) (etate ?ani) )
        =>
        (printout  t   "Nume : " ?nume  crlf  "Initiala_tatalui : " ?init  crlf
                       "Prenume : " ?prenume  crlf  "Ocupatia : " ?loc crlf
                       "Etate : "  ?ani  crlf   "**********************"  crlf  ) )
```

Un alt exemplu în care s-ar putea observa utilitatea folosirii variabilelor multicâmp este acela în care fiind dată o persoana cu mai mulţi copii se doreşte fie afişarea copiilor acesteia, fie căutarea unui anumit copil în baza de fapte pentru a se observa dacă acesta există sau nu.

```
(deftemplate parinte
        (multislot nume)
        (multislot  copii) )

(deffacts parinti
        (parinte (nume Marian Petru) (copii  Stefan  Maria  Ana))
        (parinte (nume Cristina Popa) (copii  Ana  Ionut)) )

(defrule afiseaza_copii
        (afiseaza_copii $?nume)
        (parinte (nume $?nume) (copii $?copii))
        =>
        (printout  t  ?nume " are copii " ?copii  crlf) )

(defrule gaseste_copil
        (gaseste_copil ?copil)
        (parinte (nume $?nume) (copii $?inainte  ?copil  $?dupa))
        =>
        (printout  t  ?nume " are copilul " ?copil ". "
                    "Ceilalti copii sunt  " ?inainte ?dupa crlf) )
```

Se observă folosirea în ultima regulă în cadrul multislotului *copii* a mai multor variabile multicâmp. Scopul este acelaşi ca şi cel din exemplul arătat la variabilele libere multicâmp şi anume acela de a se căuta între toţi copiii (fiecare variabilă poate lua zero sau mai multe valori). Pentru a vedea cum lucrează regulile se vor da comenzile:

```
CLIPS> (reset)
CLIPS> (assert (afiseaza_copii Cristina Popa))
<Fact-3>
CLIPS> (run)
(Cristina Popa) are copii (Ana Ionut)
CLIPS> (assert (gaseste_copil Ana))
<Fact-4>
CLIPS> (run)
(Cristina Popa) are copilul Ana. Ceilalti copii sunt ( ) (Ionut)
(Marian Petru) are copilul Ana. Ceilalti copii sunt (Stefan Maria) ()
CLIPS> (assert (gaseste_copil Cristian))
<Fact-5>
CLIPS> (run)
CLIPS>
```

Se observă modul în care sunt legate variabilele *?înainte* şi *?după* în cazul căutării copilului *Ana*. In ambele situaţii găsite (regula este activată de două template-uri) ambele variabile iau şi valoarea *( )*, adică nu sunt legate de nici o valoare şi numărul de câmpuri pe care îl înlocuiesc este de zero câmpuri. Trebuie

reținut faptul că în LHS o variabila multicâmp trebuie să aibă întotdeauna operatorul $ şi nu numai la prima apariție. CLIPS-ul nu semnalează eroare dacă pentru următoarele apariții se omite operatorul $ şi va avea în lista temporară a regulii o variabilă multicâmp şi o variabilă cu un singur câmp cu acelaşi nume. Se poate observa acest fapt ştergând operatorul $ la a doua apariție a variabilei *nume* din regula *afiseaza_copii*. Când se va da comanda *(run)* nu se va găsi nici o activare a regulii în agendă şi deci nu se va executa nimic.

2.6.1. Implementarea unei stive

După cum se ştie o stivă este o structură de date ordonată în care putem să adăugăm sau să ştergem articole. Un articol nou poate fi adăugat , printr-o operație numită şi push (împingere), sau ultimul articol poate fi scos, operația numindu-se pop. Stiva va fi de tipul LIFO, adică primul articol introdus va fi ultimul şters şi ultimul articol introdus va fi primul şters.

Este relativ uşor de implementat o stivă cu ajutorul variabilelor multiple, care să poată efectua operațiile de push şi de pop. Presupunând existent un fapt ordonat numit *stiva*, care conține o colecție de articole, următoarea regulă va efectua operația de introducere a unei valori pentru acest fapt :

```
CLIPS> (clear)
CLIPS> (assert (stiva  Rosu  Galben)
               (push Albastru))
       <Fact-1>
CLIPS> (defrule push
               ?push <- (push ?valoare)
               ?stiva <- (stiva  $?rest)
               =>
               (retract ?push  ?stiva)
               (assert (stiva  ?valoare  $?rest) )
               (printout  t  "S-a adaugat elementul " ?valoare  "."  crlf) )
CLIPS> (run)
CLIPS> (facts)
f1     (stiva  Rosu  Galben Albastru)
```

Pentru operația de scoatere a unui element sunt necesare două reguli; una care să afişeze un mesaj de eroare în cazul în care în stivă nu mai este nici un element; cealaltă care să efectueze operația de *pop* normal în cazul în care în stivă mai sunt elemente.

```
(defrule pop_valid
        ?pop <- (pop)
        ?stiva <- (stiva ?valoare $?rest)
        =>
        (retract  ?pop  ?stiva)
        (assert (stiva $?rest) )
        (printout  t  "A fost  şters elementul " ?valoare "."  crlf) )
```

```
(defrule pop_invalid
        ?pop <- (pop)
        ?stiva <- (stiva)
        =>
        (retract ?pop ?stiva)
        (printout  t  “Stiva este goala ! ” crlf) )
```

Odată introduse aceste reguli se pot da comenzi de push şi de pop pentru a vedea cum funcționează aceste reguli.

```
CLIPS> (assert (pop))
CLIPS> (run)
A fost şters elementul Albastru.
CLIPS> (set-fact-duplication TRUE)          ;pot fi asertate fapte identice
CLIPS> (assert (pop)   (pop)  (pop))
CLIPS> (run)
A fost şters elementul Galben.
A fost şters elementul Roşu.
Stiva este goala!
```

Dacă la regulă *push* nu s-ar retracta valoarea ce se doreşte să fie introdusă şi s-ar lăsa doar comanda *(retract ?stiva)* am obține o buclă infinită, valoarea respectivă fiind introdusă la infinit. Dacă, de asemenea, pentru regula *pop_valid* nu s-ar retracta faptul *(pop),* s-ar executa operația de ştergere a tuturor elementelor până ce stiva va fi goală şi regula *pop_invalid* ar afişa un mesaj de eroare.

În schimb, dacă şi pentru regula *pop_invalid* s-ar lăsa doar comanda *(retract ?stiva)* în acest caz, nu se poate obține o buclă infinită, deoarece am avea în cadrul LHS-ului aceleaşi două fapte care nu mai pot aprinde regula încă o dată. Toate acestea pot fi verificate realizându-se modificările respective. În cazul în care avem mai multe stive şi se doreşte adăugarea sau ştergerea într-o anumită stivă specificată de utilizator, atunci nu trebuiesc făcute decât mici modificări la regulile de mai sus. Astfel, pentru regula *push* se vor schimba pattern-urile:

```
?push <- (push ?valoare)              |   ?push <- (push ?nume ?valoare)
?stiva <- (stiva  $?rest)             |   ?stiva <- (stiva ?nume  $?rest)
(assert (stiva  ?valoare  $?rest))    |   (assert (stiva  ?nume ?valoare $?rest))
```

Pentru regula *pop_valid* se vor face de asemenea modificările:

```
?pop <- (pop)                         |   ?pop <- (pop ?nume)
?stiva <- (stiva ?valoare $?rest)     |  ?stiva <- (stiva ?nume ?valoare $?rest)
(assert (stiva $?rest) )              |   (assert (stiva ?nume $?rest) )
```

Pentru regula *pop_invalid* se vor modifica pattern-urile:

```
?pop <- (pop )                        |    ?pop <- (pop ?nume)
?stiva <- (stiva)                     |    ?stiva <- (stiva ?nume)
```

Variabila ?nume va specifica numele stivei în care se doreşte efectuarea operațiilor de push sau de pop. Elementele unei stive vor fi introduse începând cu al doilea câmp, deoarece primul câmp va avea introdus numele (numărul) stivei.

2.6.2. Problema blocurilor revizuită

Folosind de această dată variabilele muticâmp şi wildcards-urile multicâmp se poate rescrie problema blocurilor într-o formă mai uşor de înțeles. De această dată fiecare stivă va fi reprezentată de un fapt ordonat după cum se va vedea. Operațiile de mutare ale blocurilor vor fi similare cu operațiile de push/pop într-o stivă. Din toate construcțiile deftemplate anterioare se va păstra doar una singură (deftemplate *scop*). Aceasta va fi introdusă la început deoarece CLIPS-ul trebuie să facă deosebire între sloturile acestei construcții şi funcții, în caz contrar fiind semnalată o eroare.

Pentru cei ce doresc problema se poate modifica, astfel încât blocurile care trebuiesc puse jos pe podea, să fie puse într-o stivă special construită pentru acestea (inițial vidă- fără nici un element).

```
(deftemplate     scop
        (slot muta)
        (slot peste) )

(deffacts stare-initiala
        (stiva  A B C)
        (stiva  D E F)
        (scop (muta C) (peste E) ) )

(defrule muta-direct
        ?scop <- (scop (muta ?bloc1) (peste ?bloc2) )
        ?stiva1 <- (stiva ?bloc1  $?rest1)
        ?stiva2 <- (stiva ?bloc2  $?rest2)
        =>
        (retract ?scop ?stiva1 ?stiva2)
        (assert (stiva  $?rest1)
        (stiva  ?bloc1 ?bloc2  $?rest2) )
        (printout  t  ?bloc1 " mutat peste " ?bloc2 "." crlf ) )

(defrule muta-pe-podea
        ?scop <- (scop (muta ?bloc) (peste podea) )
        ?stiva <- (stiva ?bloc $?rest)
        =>
        (retract ?scop ?stiva)
        (assert  (stiva ?bloc)
                 (stiva $?rest) )
        (printout  t  ?bloc " mutat pe podea." crlf) )

(defrule stergere-pt-blocul-de-deasupra
        (scop (muta ?bloc1) )
        (stiva ?bloc2  $?  ?bloc1  $?)       ;bloc2-vârful stivei lui bloc1
        =>
        (assert (scop (muta ?bloc2) (peste podea) ) ) )
```

```
(defrule stergere-pt-blocul-de-desubt
        (scop (peste ?bloc1) )
        (stiva ?bloc2  $?  ?bloc1 $?)        ;bloc2-vârful stivei lui bloc1
        =>
        (assert (scop (muta ?bloc2) (peste podea) ) ) )
```

2.7. Potrivirea într-un pattern pe mai multe căi

Deşi de cele mai multe ori un fapt activează o regulă o singură dată, există şi cazuri când un fapt poate activa o regulă de mai multe ori, pe mai multe cai. Acest lucru se întâmplă atunci când acelaşi fapt se potriveşte cu pattern-ul regulii în mai multe moduri. Pentru exemplele de mai sus vom introduce o persoană care are mai mulți copii cu acelaşi nume.

```
CLIPS> (reset)
CLIPS> (assert (parinte (nume Leca Andrei) (copii Dan Dan Dan)))
<Fact-3>
CLIPS> (assert (gaseste_copil Dan))
<Fact-4>
CLIPS> (agenda)
0      gaseste_copil:   f-4,f-3
0      gaseste_copil:   f-4,f-3
0      gaseste_copil:   f-4,f-3
For a total of 3 activations.
CLIPS> (run)
(Leca Andrei) are copilul Dan. Ceilalti copii sunt () (Dan Dan)
(Leca Andrei) are copilul Dan. Ceilalti copii sunt (Dan) (Dan)
(Leca Andrei) are copilul Dan. Ceilalti copii sunt (Dan Dan) ()
```

Există trei căi diferite în care regula este aprinsă de variabilele *?copil*, *$?înainte* şi *$?dup*, acestea în funcție de modul în care sunt legate variabilele de noul fapt introdus f-3. Se pot observa foarte bine, după cele trei mesaje afişate de regulă, la fiecare aprindere a sa, valorile asociate celor două variabile multicâmp.

2.8. Întrebări

1. Între variabilele: ?a ?A , CLIPS-ul face deosebire ?
2. În urma ştergerii unui fapt, un alt fapt poate avea ca adresă pe cea a faptului care a fost retractat? De ce?
3. Cu ajutorul cărui operator putem lega adresa unui fapt de o variabilă, pentru ca apoi să putem realiza operații de ştergere sau de modificare asupra acestuia?
4. Care este diferența între o variabilă obişnuită şi o variabilă liberă? Când se recomandă folosirea unei variabile libere?

5. Care este diferența între o variabilă multicâmp şi una de un singur câmp? Când este recomandată folosirea unei variabile de tip multicâmp în locul unei variabile obişnuite (de un singur câmp)?
6. In exemplul următor extras din LHS-ul unei reguli, presupunând că *nume* este un fapt sau un slot multicâmp, indicați care sunt valorile pe care le poate lua variabila ?val. Această variabilă poate lua valoarea primului şi ultimului câmp? Argumentați afirmațiile făcute.

   (nume $? ?val $?)

7. Dacă în exemplul anterior am avea faptul nume de un singur câmp, CLIPS-ul ar semnala o eroare la încărcare ? De ce?

2.9. Aplicații

1. Modificați problema blocurilor astfel încât mutarea blocurilor să nu mai fie realizată pe podea ci pe o altă stivă.
2. Puneți în stive şi elemente care se repetă. Observați ce se întâmplă atunci când avem un bloc în ambele stive, sau de două ori în aceeaşi stivă.
3. Realizați la sfârşitul problemei afişarea stivelor, dacă este posibil cu ajutorul unei singure reguli.
4. Implementați o coadă (FIFO), folosind aceleaşi operații de push şi pop la care se pot face o serie de verificări suplimentare (de exemplu se poate limita numărul maxim de elemente din stivă).
5. Incercați implementarea unei cozi circulare, în care la momentul în care s-a ajuns la numărul maxim de elemente se afişează un mesaj de eroare, sau se suprascrie peste primul element din coadă (incrementând poziția primului şi a ultimului element).
6. Pentru exemplul în care se realiza potrivirea unui patern pe mai multe căi (ca efect al faptului că un părinte avea mai mulți copii cu acelaşi nume), realizați o regulă care să afişeze numele părintelui dacă găseşte un astfel de caz.
7. Realizați un arbore genealogic şi dezvoltați mai multe reguli care să acopere mai multe tipuri şi grade de rudenie dintre persoanele introduse în acel arbore.

2.10. Probleme rezolvate

1. **Arbore genealogic -** Fie un grup de persoane, în care, pentru fiecare persoană, este specificat numele şi copiii acesteia. Se cere realizarea unei reguli care să afişeze numele persoanelor care sunt verii unei anumite persoane, specificate. Se pot realiza, de asemenea, reguli care să afişeze bunicii, mătuşile (etc.) pentru o persoană anume. În acest fel putem realiza pentru un arbore genealogic mai mare, o serie de reguli care să specifice gradul şi tipul de rudenie dintre două persoane.

```
(deftemplate persoana
        (slot nume)
        (multislot copii))

(deffacts date_initiale
        (persoana (nume Liviu) (copii Vlad Maria Nelu))
        (persoana (nume Vlad) (copii Oana Mircea))
        (persoana (nume Maria) (copii Dan))
        (persoana (nume Nelu) (copii Paul Radu Sorin))
        (cauta_verisori Dan))

(defrule verisori
        (cauta_verisori ?n)
        (persoana (nume ?p1) (copii $? ?n $?))
        (persoana (nume ?b) (copii $? ?p1 $?))
        (persoana (nume ?b) (copii $? ?p2 $?))
        (persoana (nume ?p2) (copii $?v))
        (test (neq ?p1 ?p2))
        =>
        (printout t "Verii lui “  ?n  “ sunt: "  $?v crlf))
```

După cum se observă din LHS-ul regulii, părinții ?p1 şi ?p2 sunt fraţi între ei deoarece au acelaşi părinte, ?b (bunicul persoanei specificate). Testul făcut la final este pus pentru a nu se ajunge în situaţia în care ?p1 şi ?p2 sunt una şi aceiaşi persoană şi în care persoana ?n este văr cu el însuşi. Facilitatea de a face teste în cadrul LHS-ului unei reguli asupra diferitelor variabile va fi prezentată în capitolul următor. Acelaşi efect s-ar obţine prin realizarea a două reguli aproape identice, diferite între ele doar printr-un singur pattern. Se acoperă astfel cele două situaţii în care numele părintelui ?p1 este situat înaintea lui ?p2 în cadrul unui fapt şi invers.

```
        (persoana (nume  ?b) (copii  $?  ?p1  $?  ?p2  $?) )
        (persoana (nume  ?b) (copii  $?  ?p2  $?  ?p1  $?) )

(defrule veri_1
        (cauta_veri ?n)
        (persoana (nume  ?p1) (copii  $?  ?n  $?) )
        (persoana (nume  ?b) (copii  $?  ?p1  $?  ?p2  $? ) )
        (persoana (nume  ?p2) (copii  $?v) )
        (test (neq  ?p1  ?p2) )
        =>
        (printout t "Verii lui “  ?n  “ sunt: "  $?v crlf) )

(defrule veri_2
        (cauta_veri  ?n)
        (persoana (nume  ?p1) (copii  $?  ?n  $?))
        (persoana (nume  ?b) (copii  $?  ?p2  $?  ?p1  $? ) )
        (persoana (nume  ?p2) (copii  $?v) )
        (test (neq  ?p1  ?p2) )
        =>
        (printout t "Verii lui “  ?n  “ sunt: "  $?v crlf) )
```

Dezavantajul constă în faptul că nu s-a realizat o scriere a programului într-o formă cât mai compactă. In cazul în care una sau mai multe reguli sunt aproape identice şi diferă doar printr-un singur pattern se poate folosi şi elementul condiţional **or**. Se subînţelege că ?p1 este diferit de ?p2 (variabile diferite) şi nu mai este necesar testul de inegalitate. Pentru o persoană care nu are copii ($?v mulţimea vidă) regula ar afişa pe ?n văr cu "nimeni".

```
(defrule veri
        (cauta_veri  ?n)
        (persoana (nume ?p1) (copii $? ?n $?))
        (or       (persoana (nume ?b) (copii $? ?p2 $? ?p1 $?))
                  (persoana (nume ?b) (copii $? ?p2 $? ?p1 $?)) )
        (persoana (nume ?p2) (copii $?v))
        (test (neq ?p1 ?p2))
        =>
        (printout t "Verii lui "  ?n  " sunt: "  $?v crlf))
```

Presupunem că pentru problema noastră avem la un moment dat un fapt care specifică tatăl şi un altul care specifică mama unor copii. Va funcţiona bine sau nu problema, pentru acest caz? Vor fi sau nu vor fi fraţii, în mod eronat, veri? Cei ce vor dori să afle răspunsul pot încerca să aserteze două astfel de fapte. Pentru alte relaţii de rudenie putem scrie regulile următoare:

```
(defrule bunic_sau_bunica
        (este_bunic  ?b)
        (persoana (nume  ?b) (copii  $?  ?p  $?))
        (persoana (nume  ?p) (copii  ?  $?))
        =>
        (printout  t  "Persoana  "  ?b  " este bunic." t) )

(defrule unchi_sau_matusă
        (este_unchi  ?u)
        (or  (persoana (nume  ?b) (copii  $?  ?p  $?  ?u  $?))
             (persoana (nume  ?b) (copii  $?  ?p  $?  ?u  $?)) )
        (persoana (nume  ?p) (copii  ?  $?))
        =>
        (printout  t  "Persoana  "  ?b  " este unchi." t) )
```

In slotul *(copii ? $?)* se subînţelege că există cel puţin un copil ($? ia zero sau mai multe valori, iar ? ia obligatoriu o valoare. O altă soluţie ar fi realizarea unui test **(test (> (length$ $?c) 0))**, unde funcţia length$ returnează lungimea unui multicâmp.Dacă s-ar adăuga la construcţia deftemplate un slot care specifică sexul persoanei, s-ar putea modifica regulile de mai sus, astfel încât să se ştie exact dacă persoana este unchi sau mătuşă, bunic sau bunică.

```
(deftemplate persoana
        (slot nume)
        (slot sex)
        (multislot copii))
```

```
(defrule veri_de_pe_tata
        (cauta_veri  ?n)
        (or (persoana (copii  $?  ?p1  $?  ?p2  $?))
            (persoana (copii  $?  ?p2  $?  ?p1  $?))
        (persoana (nume  ?p1) (sex M) (copii  $?  ?n  $?))
        (persoana (nume  ?p2) (copii  $?  ?v  $?))
        =>
        (printout  t  ?n  " este verisor cu "   ?v  t) )

(defrule strabunica
        (cauta_strabunica  ?n)
        (persoana (nume  ?sb) (sex F) (copii  $?  ?b  $?))
        (persoana (nume  ?b) (copii  $?  ?p  $?))
        (persoana (nume  ?p) (copii  $?  ?n  $?))
        =>
        (printout  t  "Persoana  "  ?sb  " este strabunica." t) )
```

2. Lanțul slăbiciunilor – Fiind date un număr de persoane şi fiind definite relațiile de prietenie între acestea, se cere să se specifice dacă persoana X poate face cunoştință cu persoana Y, ştiindu-se faptul că un prieten al unei persoane îi poate face cunoştință acesteia cu unul din prietenii ei şi tot aşa până probabil cele două persoane ar putea ajunge să se cunoască. Se mai cere eventual drumul minim între cele două persoane.

```
(deftemplate persoana
        (slot nume)
        (multislot prieteni) )

(deffacts prieteni
        (persoana (nume Marius) (prieteni  X  Y  Z))
        (persoana (nume X) (prieteni  A  B  Marius))
        (persoana (nume Y) (prieteni B  D  E))
        (persoana (nume E) (prieteni D Maria))
        (persoana (nume Maria) (prieteni Y  X  D))
        (cunoaste  Marius  Maria)
        (lant  Marius))

(defrule cunoaste_persoana
        (lant  $?first  ?x)
        (persoana (nume ?x) (prieteni  $?  ?z  $?))
        (test (not (member$ ?z $?first)))     ;persoana ?z nu este in lista
        =>
        (assert (lant  $?first ?x  ?z)) )

(defrule lant_slabiciuni
        (cunoaste  ?x  ?y)
        (lant  ?x  $?lant  ?y)
        =>
        (printout  t  "Lantul slabiciunilor:  "  ?x  $?lant  ?y  crlf) )
```

Până la acest moment programul se execută corect, dar soluțiile pe care acesta le poate furniza sunt legate de persoanele pe care le poate cunoaşte prin intermediul prietenilor săi, doar una dintre cele două persoane (în cazul nostru Marius). Pentru a remedia această problemă să încercăm găsirea de soluții şi din sens invers: este suficient să introducem în faptele inițiale şi faptul *(lant Maria)*. Pentru ca să poată fi afişate şi celelalte soluții regula *lant_slabiciuni* se va modifica astfel:

```
(defrule lant_slabiciuni
        (cunoaste  ?x  ?y)
        (or  (lant  ?x  $?lant  ?y)
             (lant  ?y  $?lant  ?x) )
        =>
        (printout  t  "Lant slabiciuni:  "  ?x  $?lant  ?y  crlf) )
```

In acest moment, la execuție se afişează toate soluțiile, dar pornind întotdeauna de la persoana ?x şi niciodată de la persoana ?y. Dacă se doreşte remedierea acestei probleme de afişare se rescrie regula astfel:

```
(defrule lant_slabiciuni
        (or  (cunoaste  ?x  ?y)
             (cunoaste  ?y  ?x) )
        (lant  ?x  $?lant  ?y)
        =>
        (printout  t  "Lantul slabiciunilor:  "  ?x  $?lant  ?y  crlf) )
```

Pentru afişarea lanțului de dimensiune minimă, se poate folosi regula următoare, în care sunt folosite elementele condiționale **not**, **or** şi **exists** pentru a verifica o condiție ce s-ar exprima literar "Să nu existe un şir de dimensiune mai mică". Deoarece avem pentru cele două şiruri de comparat patru cazuri posibile (fiecare şir poate începe cu ?x sau cu ?y) se vor folosi două elemente condiționale **or**, eliminându-se astfel toate situațiile posibile.

```
(defrule lant_minim
        (cunoaste  ?x  ?y)
        (or (lant ?x $?l1 ?y)
            (lant ?y $?l1 ?x) )
        (not(exists (or (lant ?x $?l2&:(< (length$ $?l2) (length$ $?l1)) ?y)
                        (lant ?y $?l2&:(< (length$ $?l2) (length$ $?l1)) ?x) )))
        =>
        (printout  t "Lantul minim dintre: " ?x " şi " ?y  " este " $?l1 crlf) )
```

Regula *lant_minim* conține multe elemente care vor fi studiate în capitolele următoare şi a fost folosită doar pentru a ilustra modul în care s-ar putea realiza afişarea lanțului minim. Putem considera problema anterioară prin generalizare, ca o *problema de drum minim între două noduri ale unui graf orientat*.

3. **Drum minim** - Date fiind un număr de segmente şi două puncte din plan pe care dorim să le unim printr-o linie de dimensiune cât mai mică, se cer toate drumurile posibile (eventual drumul minim). Diferența între problema

anterioară şi cea actuală constau în faptul că de această dată graful nu mai este orientat (pe un arc între două noduri putem merge în orice direcție). Puterea algoritmului constă în faptul că nu se foloseşte o metodă de **back-traking**, nu se merge pe un drum unic (nu se şterge nici un fapt *drum*), ci se caută toate soluțiile simultan (algoritm tipic programării logice).

```
(deffacts segmente
        (segm A B) (segm A C) (segm A D) (segm B C) (segm B D) (segm D E)
        (uneste  A E)
        (drum  A))

(defrule adauga_punct
        (drum  $?first  ?x)
        (or   (segm  ?x  ?y)
              (segm  ?y  ?x))
        (test (not (member$  ?y  $?first)))              ;punctul ?y nu este pe linie
        =>
        (assert (drum  $?first ?x  ?y)) )

(defrule drum
        (uneste  ?x  ?y)
        (drum  ?x  $?dr  ?y)
        =>
        (printout  t  “Drum:  ”  ?x  $?dr  ?y  crlf) )

(defrule lant_minim
        (uneste  ?x  ?y)
        (drum  ?x  $?d1  ?y)
        (not(exists (drum  ?x  $?d2&:(< (length$ $?d2) (length$ $?d1))  ?y)  ))
        =>
        (printout  t "Drumul minim dintre: " ?x “ şi “ ?y  “ este “ $?d1 crlf) )
```