CAP 2. Funcţii definite de utilizator

2.1. Construcţia DEFFUNCTION

Construcţiile *deffunction* permit definirea de noi funcţii direct în CLIPS. În versiunile anterioare ale CLIPS-ului, singura cale pentru a avea funcţii definite de utilizator (**user-defined-functions**) era scrierea acestora într-un limbaj extern, cum ar fi C sau Ada, şi recompilarea şi relinkeditarea surselor CLIPS-ului împreună cu noile surse. Corpul unei construcţii *deffunction* este format dintr-o serie de expresii similare cu cele din RHS-ul unei reguli şi sunt executate atunci când funcţia este apelată. Valoarea returnată în mod normal de o construcţie *deffunction* este valoarea ultimei expresii evaluate din corpul acesteia. Apelarea unei funcţii definite de utilizator este identică cu apelarea oricărei funcţii din CLIPS. Sintaxa acestei construcţii este următoarea:

```
(deffunction   <name>  [<comment>]
        (<regular-parameter>* [<wildcard-parameter>])
        <action>*)
        <regular-parameter>      ::= <single-field-variable>
        <wildcard-parameter>   ::= <multifield-variable>
```

O construcţie *deffunction* este formată din 5 elemente: un nume, un comentariu opţional, o listă de zero sau mai mulţi parametri, un parametru multiplu opţional (necesar în cazul în care se doreşte un număr variabil de argumente) şi o secvenţă de acţiuni sau expresii, care vor fi executate la apelarea funcţiei.

O funcţie definită de utilizator trebuie să aibă un nume unic, diferit de numele tuturor funcţiilor de sistem sau a funcţiilor generice (se va vedea mai jos). In particular o construcţie *deffunction* nu poate fi supraîncărcată ca o funcţie de sistem. O astfel de construcţie trebuie declarată înainte de a fi apelată de o altă funcţie, obiect, regulă sau executată la top-level. Singura excepţie o reprezintă funcţiile care se autoapelează sau altfel spus funcţiile recursive.

După cum s-a afirmat mai sus o construcţie deffunction poate accepta şi zero parametri la definirea acesteia. Unica problemă în acest caz este faptul că deşi nu se declară nici un parametru nu se pot introduce direct secvenţa de expresii sau acţiuni după numele funcţiei (eventual după comentariu). Parantezele între care sunt introduşi parametri funcţiei trebuiesc obligatoriu introduse (CLIPS-ul va identifica în acest caz o listă vidă). Astfel pentru exemplul următor în care sunt omise aceste paranteze CLIPS-ul va furniza o eroare de sintaxa.

```
CLIPS>  (deffunction a  (printout  t Mesaj t))
[PRNUTIL2] Syntax Error: Check appropiate syntax for parameter list.
```

O funcţie definită poate accepta exact sau cel puţin un număr specificat de argumente, aceasta depinzând de utilizarea sau nu a parametrului multiplu

(*wildcard parameter*). Parametrii obişnuiţi specifică numărul minim de argumente care trebuiesc transmişi funcţiei. Fiecare din aceşti parametri poate fi referit ca o variabilă simplă (*single-field variable*) în acţiunile din corpul funcţiei. Dacă este prezent un parametru multiplu, atunci pot fi transmise funcţiei orice număr de argumente mai mare sau egal cu minimul. Dacă nu este prezent un astfel de parametru multiplu, atunci numărul de argumente transmise trebuie să fie identic cu numărul parametrilor obişnuiţi (*regular parameters*). Toate argumentele care nu corespund parametrilor obişnuiţi sunt grupate într-o valoare multicâmp referită de parametrul multiplu. De exemplu funcţiilor standard multicâmp, cum ar fi **length$** şi **nth$** li se poate aplica un parametru multiplu.

```
CLIPS> (deffunction  count  ($?arg)
                (length$   $?arg))
CLIPS> (deffunction print-args (?a ?b $?c)
                (printout  t   ?a  “  “  ?b  ” and “  (length ?c)  “  extras:  “  ?c  crlf) )
CLIPS> (print-args 1 2)
1 2 and 0 extras:  ( )
CLIPS> (print-args a b c d)
a b and 2 extras:  (c d)
CLIPS> (count 1 a “b”)
3
```

După cum se poate observa dacă în lista argumentelor variabila $?c este declarată de tip multicâmp, în corpul funcţiei aceasta poate fi folosită ca şi o variabilă simplă, fără a se mai adăuga caracterul ‘$’. După cum se ştie şi în LHS-ul unei reguli dacă la prima apariţie o variabilă este declarată de tip multicâmp, ea va fi astfel considerată în toată regula.

Trebuie menţionat faptul că în lista de parametri nu putem folosi decât o singură variabilă multicâmp şi aceasta trebuie să fie obligatoriu ultima în lista. Parametrii obişnuiţi care după cum s-a mai spus, specifică numărul minim de argumente care trebuiesc transmişi funcţiei trebuiesc introduşi în faţa acestui parametru multiplu, deoarece acesta va îngloba toţi ceilalţi parametri introduşi pe lângă parametrii obligatorii.

```
CLIPS> (deffunction f (?a  ?b  $?c  ?d)
                (printout  t  ?a  “->”  ?b  “-->”  ?d (first$ $?d)))
```

[PRCCODE8] No parameters allowed after wildcard parameter.

```
CLIPS> (deffunction f  ($?a  $?b)
                (printout  t  (create$  $?a  $?b)))
```

[PRCCODE8] No parameters allowed after wildcard parameter.

Dacă în interiorul LHS-ului unei reguli se puteau folosi şi variabile libere, în corpul unei funcţii existenţa acestora nu este permisă. Dacă în interiorul unei reguli ele erau utilizate doar pentru a înlocui valori ce nu erau importante în execuţia regulii respective (doar pentru a se realiza potrivirea pattern-ului regulii cu datele

din lista de fapte), în lista de argumente a unei funcţii prezenţa unei astfel de variabile nu ar fi utilă decât în cazul în care s-ar dori eliminarea erorilor ce pot apare in urma introducerii unui număr incorect de parametri. De exemplu pentru o funcţia ce realizează suma a două numere, dacă se doreşte eliminarea parametrilor suplimentari se poate folosi o variabilă multiplă explicită.

```
CLIPS> (deffunction suma (?a  ?b  $?)  (+ ?a  ?b))
[PRNUTIL2] Syntax Error: Check appropiate syntax for parameter list.

CLIPS> (deffunction suma (?a  ?b  $?arg)  (+ ?a  ?b))
CLIPS> (suma 3 4 5)
7
```

Când este apelată o funcţie definită de utilizator, acţiunile sale sunt executate pe rând. Valoarea returnată de funcţie este după cum s-a mai spus evaluarea ultimei sale acţiuni. Dacă nu avem nici o acţiune în corpul funcţiei atunci valoarea returnată de aceasta este simbolul FALSE. Acest lucru trebuie reţinut deoarece este unul din cele mai uşoare metode obţinere a simbolului FALSE în interiorul unui program. Dacă pe parcursul execuţiei unei funcţii apare o eroare, atunci acţiunile care nu au fost executate încă sunt abandonate şi funcţia returnează simbolul FALSE. De exemplu dacă pentru funcţia de mai sus unul dintre parametri nu este un număr, deoarece în interiorul acesteia nu se fac verificări asupra tipurilor parametrilor primiţi la intrare, funcţia va returna simbolul FALSE.

```
CLIPS> (eq  (suma 3 a) FALSE)
TRUE
[ARGACCES5] Function + expected argument #2 to be of type integer or float
[PRCCODE4] Execution halted during the actions of deffunction suma.

CLIPS> (deffunction f())
CLIPS> (deffunction egal (?a ?b)
                         (if (eq ?a ?b) then ))
CLIPS> (eq (f) FALSE)
TRUE
CLIPS> (while (eq (str-index (read) "Da da DA Yes yes YES D d Y y")  ( f ) )
              (printout  t  "Doriti sa iesiti (Da/Nu) ?" ))
CLIPS> (assert  (simbol (f) )
                        (sunt_egale  (egal a "a") )
                        (rez_suma  (suma 3 a) ))
<Fact-2>
CLIPS> (facts)
f-1     (simbol FALSE)
f-2     (sunt_egale FALSE)
f-3     (rez-suma)
For a total of 3 facts.
```

Din exemplul de mai sus se poate observa foarte uşor diferenţa între simbolurile FALSE returnate de funcţia *f* în mod automat (deoarece în corpul

funcţiei nu avem nici o expresie sau funcţie), de funcţia *egal* ca urmare a evaluării ultimei expresii din corpul funcţiei *(eq ?a ?b)*, şi de funcţia *suma* în urma apariţiei unei erori datorită parametrilor de intrare. Se observă cum în ultimul caz s-a prin întreruperea execuţiei oricărei acţiuni în faptul *rez_suma* CLIPS-ul nu mai are cum, să aserteze şi simbolul FALSE returnat de funcţia *suma*.

Funcţiile definite de utilizator pot fi şi recursive prin simpla introducere printre acţiunile din corpul funcţiei a unui apel al acesteia. Programatorul trebuie să fie atent în acest caz pentru a nu intra într-o buclă infinită. Un exemplu simplu de funcţie recursivă este calculul factorialului.

```
CLIPS>(deffunction  factorial  (?a)
                (if (or (not (integerp ?a)) (< ?a 0))
                 then
                        (printout t "Factorial Error!" crlf)
                 else
                        (if (= ?a 0)
                         then
                                1
                         else
                                (*  ?a  (factorial  (-  ?a  1) ) )
                        )
                )
        )
CLIPS> (factorial 5)
120
```

Mai putem realiza o recursivitate mutuală între două construcţii d*effunction*, însă aceasta presupune o **declarare dinainte** a uneia dintre cele două funcţii. O astfel de declarare este o simplă declarare a funcţiei fără nici o acţiune. În exemplul următor funcţia **f** este declarată înainte şi astfel poate fi apelată de funcţia **g**. Apoi funcţia **f** este redefinită având acţiunile care apelează funcţia **g**. O astfel de recursivitate mutuală poate fi realizată şi folosind mai mult de două funcţii definite. Prea multe nivele de recursivitate pot însă conduce la o depăşire a stivei de memorie şi acest aspect nu trebuie neglijat de programator. Acesta trebuie să încerce să nu folosească recursivitatea în mod excesiv, pentru a nu se ajunge în astfel de situaţii critice.

```
CLIPS> (deffunction  f  ())              ;declarare dinainte
CLIPS> (deffunction  g ()
                (f) )
CLIPS> (deffunction  f  ()
                (g) )
```

În exemplul de mai sus funcţiile **f** şi **g** se apelează una pe cealaltă într-o buclă infinită. Din acest motiv CLIPS-ul se blochează, fiind necesară o reîncărcare a acestuia. Pentru a se evita o astfel de situaţie programatorul trebuie să definească funcţiile în aşa fel încât acestea să se oprească după un număr finit de execuţii.

```
CLIPS> (deffunction f (?n) )
CLIPS> (deffunction g (?n)
                (printout t ?n " ")
                (if (> ?n 0)
                 then
                        (f (- ?n 1) )
                else
                        (halt) )
        )
CLIPS> (deffunction f (?n)
                (printout t ?n " ")
                (if (> ?n 0)
                        then
                        (g (- ?n 1) )
                else
                        (halt) )
        )
CLIPS> (f 10)
10 9 8 7 6 5 4 3 2 1 0
```

```
CLIPS> (deffunction f (?n) )
CLIPS> (deffunction g (?n)
                (if (> ?n 0)
                 then
                        (f (- ?n 1) )
                 else
                        (halt) )
                (printout t ?n " ")
        )
CLIPS> (deffunction f (?n)
                (if (> ?n 0)
                 then
                        (g (- ?n 1) )
                else
                        (halt) )
                (printout t ?n " ")
        )
CLIPS> (f 10)
0 1 2 3 4 5 6 7 8 9 10
```

Un alt aspect care trebuie luat în considerare este locul în care este amplasat apelul funcţiei recursive. În cazul de mai sus dacă este schimbată poziţia funcţiei ***if*** (în care este inclus apelul mutual recursiv al celeilalte funcţii) cu cea a funcţiei ***printout*** se va obţine după cum era de aşteptat o afişare a numerelor în ordine crescătoare. Amplasări diferite a acţiunilor din corpurile celor două funcţii pot duce şi la o afişare a numerelor pare şi eventual a numerelor impare (se pot face şi aceste modificări).

```
CLIPS> (deffunction f (?n) )
CLIPS> (deffunction g (?n)
                (printout t ?n " ")
                (if (> ?n 0)
                 then
                        (f (- ?n 1) )
                else
                        (halt) )
        )
CLIPS> (deffunction f (?n)
                (if (> ?n 0)
                 then
                        (g (- ?n 1) )
                else
                        (halt) )
                (printout t ?n " ")
        )
CLIPS> (f 10)
9 7 5 3 1 0 2 4 6 8 10
```

```
CLIPS> (deffunction f (?n) )
CLIPS> (deffunction g (?n)
                (if (> ?n 0)
                 then
                        (f (- ?n 1) )
                 else
                        (halt) )
                (printout t ?n " ")
        )
CLIPS> (deffunction f (?n)
                (printout t ?n " ") )
                (if (> ?n 0)
                 then
                        (g (- ?n 1) )
                else
                        (halt) )
        )
CLIPS> (f 10)
10 8 6 4 2 0 1 3 5 7 9
```

Se afişează în continuare funcţiile şi comenzile din CLIPS utilizate pentru funcţiile definite de utilizator.

(**list-deffunctions** [<module-name>])	Afişează o listă cu toate construcţiile *deffunction* din modulul specificat (sau din modulul curent dacă nu este specificat).
(**deffunction-module** <deffunction-name>)	Returnează modulul în care este definită construcţia *deffunction* specificată.
(**ppdeffunction** <deffunction-name>)	Afişează textul cerut la definirea construcţiei *deffunction* specificate.
(**get-deffunction-list** [<module-name>])	Afişează o listă cu toate construcţiile *deffunction* din modulul specificat (sau din modulul curent dacă nu este specificat).
(**undeffunction** <deffunction-name>)	Şterge construcţia *deffunction* specificată. Simbolul ***** poate fi folosit pentru a şterge toate construcţiile *deffunction*. Nu putem efectua ştergerea unei funcţii care este în execuţie sau este referită de o altă construcţie încărcată (ex. RHS-ul unei reguli).

În cazul în care se doreşte ştergerea unei funcţii definite de utilizator referită şi de alte construcţii, atunci trebuiesc şterse mai întâi toate aceste construcţii şi apoi dată comanda **undeffunction**. O situaţie cu adevărat specială este cea în care avem o recursivitate mutuală. Astfel pentru exemplul de mai sus dacă am dori ştergerea funcţiilor *f* şi *g*, nu am putea realiza ştergerea nici uneia dintre aceste două funcţii definite de utilizator, deoarece fiecare dintre acestea o apelează pe cealaltă. În acest caz unica modalitate de a şterge cele două funcţii este de a şterge toate funcţiile folosind *(undeffunction *)*, sau ştergerea tuturor construcţiilor (defrule, defglobal, defgeneric, defmethod etc.), folosind funcţia *(clear*). În unele situaţii această variantă nu este avantajoasă, deoarece trebuiesc redeclarate funcţiile, care au fost şterse.

```
CLIPS> (undeffunction f)
[PRNUTIL4] Unable to delete deffunction f.
CLIPS> (undeffunction g)
[PRNUTIL4] Unable to delete deffunction g.
```

Se menţionează faptul că definirea unei funcţii cu un nume identic cu numele unei funcţii declarate anterior, duce la suprascriere (cea mai nouă funcţie ia locul celei anterioare).

```
CLIPS> (deffunction g (?a ?b)                    ;se suprascrie functia g
                (+ ?a ?b) )
CLIPS> (deffunction f (?a ?b $?arg)              ;se suprascrie functia f
                (printout t ?a "+" ?b "=" (g ?a ?b) t) )
CLIPS> (f 2 3 4)
2+3=5
```

```
CLIPS> (f  2)
[ARGACCES4] Function f expected at least 2 agument(s).
CLIPS> (undeffunction g)                ;functia g este referita de functia f
[PRNUTIL4] Unable to delete deffunction g.
CLIPS> (undeffunction f)
CLIPS> (undeffunction g)                ;functia g a fost stearsa
```

Funcţiile definite mai sus realizează suma a două numere şi afişarea operaţiei de sumare în forma *inordine* (în CLIPS o expresie este scrisă în forma *preordine*, mai întâi operatorul şi apoi operanzii). Deoarece funcţia *g* este apelată de funcţia *f*, pentru ştergerea acesteia este necesară mai întâi ştergerea funcţiei *f*, în care este referită. După cum se observă funcţia *f* acceptă un număr de minim 2 argumente, şi în cazul în care se introduc un număr mai mare de argumente acestea vor fi ignorate. Să încercăm sumarea tuturor argumentelor şi afişarea acestora în forma *inordine*. Suplimentar se va verifica şi corectitudinea datelor de la intrare.

```
CLIPS> (deffunction suma (?a  $?arg)
                  (if (> (length$ $?arg) 0)
                   then  (printout t  ?a "+")
                         (return (+ ?a (suma (expand$ $?arg))))
                   else   (printout t ?a "=")
                         (return ?a)) )
CLIPS> (suma 1 2 3 4)
1+2+3+4=10
CLIPS> (suma 1)
1=1
CLIPS> (suma 1 –1 2 –2)
1+-1+2+-2=0
```

Funcţia *suma* definită mai sus este recursivă. Pentru cazul în care au rămas adunat mai mult două numere, noii parametrii ai funcţiei trebuiesc expandaţi, diferenţa între următoarele două apeluri ale funcţiei

(suma $?arg)) *(suma (expand$ $?arg))*

constând în faptul că în primul caz este transmis un singur parametru (o valoare multicâmp) şi am obţine o eroare în cazul în care am încerca să o adunăm; iar în al doilea caz sunt transmişi toţi parametrii, deoarece aceştia nu mai sunt incluşi într-o variabilă multiplă de tip listă. Funcţia următoare verifică într-o buclă de tip *for* dacă parametrii de intrare sunt numere (întregi sau flotanţi). În cazul în care nu avem parametri eronaţi este apelată funcţia *suma*.

```
CLIPS> (deffunction s ($?arg)
                  (loop-for-count  (?i 1 (length$ $?arg))
                           (if  (not (numberp (nth$ ?i $?arg) ))
                            then (printout  t  "Parametri eronaţi " t ) (return)) )
                  (suma (expand$ $?arg)) )
CLIPS> (suma 1 2 3 a 4)
Parametri eronati.
```

2.2. Construcţia DEFGLOBAL

Cu ajutorul unei construcţii **defglobal** pot fi definite şi accesate de mediul CLIPS-ului variabilele globale. Acestea pot fi accesate ca parte a procesului pattern matching, fără a invoca acest proces în momentul în care este schimbată valoarea acestora. Pentru setarea valorilor variabilelor globale este folosită funcţia *bind*. Variabilele globale sunt resetate la valoarea lor originală în momentul în care este efectuată comanda **reset**, sau când funcţia **bind** apelează variabila globală fără nici o valoare. Variabile globale pot fi schimbate şi folosind funcţia **set-reset-globals**, ştergerea acestora se realizează cu ajutorul a comenzii **undefglobal** sau a comenzii **clear**. Se pot urmări schimbările valorilor variabilelor globale vizualizând fereastra "Globals" în varianta de CLIPS sun Windows, sau folosind comanda **watch globals** (în acest va fi afişat caz un mesaj informaţional de fiecare dată când este schimbată o valoare a unei variabile globale). Sintaxa acestei construcţii este:

(defglobal [<defmodule-name>] <global-assignement>*)

<global-assignement> ::= ?*<symbol>* = <expression>

Într-un program pot exista mai multe construcţii *defglobal* şi în fiecare construcţie pot fi definite oricâte variabile globale. Opţionalul <defmodule-name> indică modulul în care va fi definită construcţia *defglobal*. Dacă acesta nu este specificat atunci construcţia *defglobal* va fi plasată în modulul curent. Dacă avem o variabilă deja definită într-o construcţie *defglobal* anterioară, atunci valoarea acesteia va fi înlocuită cu valoarea găsită în noua construcţie *defglobal* (va fi suprascrisă). Dacă în momentul în care se încearcă definirea unei construcţii *defglobal* se obţine o eroare, atunci orice variabilă globală definită înaintea producerii acestei erori rămâne valabilă.

Comenzile care operează cu *defglobals* cum ar fi **ppdefglobal** sau **undefglobal** aşteaptă intoducerea numelui simbolic al variabilei globale, ce este încadrat de asteriscuri (de exemplu se foloseşte simbolul *max* când ne referim la variabila globală *?*max**). Exemplu de construcţie *defglobal*:

```
(defglobal  ?*x*  =  3
            ?*y*  =  ?*x* )

CLIPS> (bind ?*x* 5)                    ;  ?*x* =5        ?*y* 3
5
CLIPS> (reset)                          ;  ?*x* =3        ?*y* 3
CLIPS> (set-reset-globals FALSE)        ;resetul nu mai modifica valorile
CLIPS> (bind ?*x* 5)                    ;  ?*x* =5        ?*y* 3
5
CLIPS> (reset)                          ;  ?*x* =5        ?*y* 3
CLIPS> (ppdefglobal  x)             ;afiseaza textul de la definirea variabilei
(defglobal MAIN ?*x* = 3)
CLIPS> (bind ?*x*)                   ;variabila globala ia valoarea initiala
3
```

```
CLIPS> (undefglobal  x)                ;?*y* = 3
```

După cum se poate observa variabilele globale ?*x* şi ?*y* nu pointează la aceiaşi locaţie de memorie. Dacă la definirea variabilei ?*y* aceasta va lua aceiaşi valoare ca şi variabila ?*x*, valoarea acesteia va rămâne nemodificată în momentul în care variabila ?*x* îşi va modifica valoarea sau va fi ştearsă. Se observă de asemenea faptul că funcţia *reset* modifică valorile variabilelor globale la cele iniţiale (din construcţia *defglobal*), doar dacă *get-reset-globals* întoarce valoarea TRUE. Spre deosebire de aceasta, funcţia *(bind ?*x*)* apelată fără o valoare explicită, "resetează" întotdeauna valoarea unei variabile globale la valoarea iniţială. Pentru o variabilă obişnuită *(bind ?v)* returnează în mod implicit simbolul FALSE (o variabilă locală nu are rezervat un spaţiu în memorie), fiind necesară pentru definirea acesteia specificarea unei valoarea explicită.

Variabilele globale pot fi folosite în orice loc la fel ca o variabilă locală, cu două excepţii. Nu pot fi utilizate ca variabilă parametru pentru o *deffunction, defmethod* sau *message-handler*. De asemenea nu putem folosi o variabilă globală în acelaşi fel cu o variabilă locală în LHS-ul unei reguli pentru a o lega de o valoare. Din acest motiv regula următoare este ilegală:

```
(defrule illegal-rule
        (fact ?*x*)
        =>)
```

[PRNTUTIL2] Syntax error: Check appropiate syntax for defrule.

În schimb o regulă cum ar fi cea care urmează este legală:

```
(defrule legal-rule
        (fact ?y&:(=  ?y  ?*x*))
        =>)
```

Nu este necesară modificarea regulii, atunci când valoarea lui ?*x* este schimbată. De exemplu dacă *?*x** este 4 şi este asertat faptul *(fact 3)*, atunci regula nu este satisfăcută. Schimbând valoarea lui *?*x** în orice altă valoare diferită de 3, regula nu va fi activată.

De reţinut faptul că atunci când sunt definite variabilele globale este necesară introducerea de spaţii înaintea şi după semnul '=' ce leagă o variabilă globală de o valoare sau expresie, în caz contrar obţinându-se o eroare de sintaxa. O variabilă globală poate fi legată de o valoare explicită, de valoarea altei variabile globale anterior declarată, de o funcţie a căror parametri pot fi de asemenea valori explicite sau variabile globale.

```
(defglobal
        ?*x*  =  3
        ?*y*  =  ?*x*
        ?*z*  = (+  ?*x*  ?*y*)
        ?*q* =  (create$  a  b  c) )
```

În condiţiile în care o variabilă globală este de tip multicâmp acest lucru nu trebuie semnalat ca în cazul variabilelor simple cu ajutorul caracterului '$'. Trebuie reţinut şi faptul că pentru o variabilă globală nu există o valoare implicită, pe care aceasta să o ia în cazul în care nu este specificată nici o valoare sau expresie, şi de aceea în cazul în care acestea nu vor precizate CLIPS-ul va genera tot o eroare de sintaxă. Este prezentat în continuare tabelul cu funcţiile şi comenzile din CLIPS utilizate pentru a afla informaţii despre variabilele globale.

(**get-reset-global)**	Returnează comportarea variabilelor globale la resetare.
(**set-reset-global** <boolean-expr>)	Setează comportarea variabilelor globale la resetare (implicit TRUE). Returnează valoarea anterioară.
(**list-defglobals** [<module-name>])	Afişează o listă cu variabilele globale din modulul specificat (sau din modulul curent dacă nu este specificat).
(**ppdefglobal** <global-variable-name>)	Afişează textul cerut la definirea variabilei globale specificate.
(**show-defglobals** [<module-name>])	Afişează numele şi valorile curente pentru toate construcţiile *defglobal* din modulul specificat (sau din modulul curent dacă nu este specificat). Se poate utiliza simbolul * pentru a afişa variabilele globale din toate modulele.
(**undefglobal** <global-variable-name>)	Şterge o variabilă globală. Se poate utiliza simbolul * pentru a şterge toate variabilele globale. Dacă o variabilă este folosită de o altă construcţie, atunci ea nu poate fi ştearsă.

Un caz mai neobişnuit este acela în care avem o variabilă globală cu numele *****. In acest caz când se va da comanda **(undefglobal *)** vom obţine ca efect ştergerea doar a variabilei globale cu numele *. Este singurul caz în care nu putem utiliza simbolul * pentru ştergerea tuturor variabilelor globale.

```
CLIPS> (defglobal ?*** = 0
                  ?*n* = A)
CLIPS> (list-defglobals)
n
*
CLIPS> (undefglobal *)
CLIPS> (list-defglobals)
n
CLIPS> (deffunction f ()
           (printout  t  "Tija initiala " ?*n* t) )
CLIPS> (undefglobal n)
[PRNUTIL4] Unable to delete defglobal n.
CLIPS> (undeffunction f)
CLIPS> (undefglobal n)
```

Dacă se va încerca introducerea unei variabile globale ca parametru pentru o construcţie *defglobal*, atunci CLIPS-ul va semnala această eroare de programare.

```
CLIPS> (deffunction f (?*x*)
                      (printout  t  ?*x*) )
[PRNTUTIL2] Syntax error: Check appropiate syntax for parameter list.
```

Pentru a obţine o funcţie asemănătoare, dar într-o formă legală putem folosi în cadrul funcţiei nou create un **switch** sau un **if** pentru a verifica dacă valoarea variabilei transmise ca parametru al funcţiei este identică cu valoarea variabilei globale.

```
CLIPS> (defglobal ?*x* = 9
                        ?*y* = 1)
CLIPS> (deffunction f (?val)
                 (switch ?val
                       (case ?*x* then *x*)
                       (case ?*y* then *y*)
                       (default none)))
CLIPS> (deffunction g(?x)
                      (if (eq  ?x  ?*x*)
                       then     (printout  t  ?x  crlf ) ) )
CLIPS> ( f  9 )
*x*
CLIPS> ( f  1 )
*y*
CLIPS> ( f  2 )
None
CLIPS> ( g  9 )
9
CLIPS> ( g  2 )
FALSE
```

Se observă în funcţia **g** faptul că se face distincţie între variabila globală ?*x* şi variabila locală ?x şi de asemenea faptul că această funcţie returnează simbolul FALSE în cazul în care valoarea variabilei transmise ca parametru funcţiei nu este identică cu cea a variabilei globale (în cazul în care nu este executată nici o acţiune şi nu este evaluată nici o expresie). Acest fapt nu este valabil şi pentru o funcţie care tratează toate cazurile posibile. De exemplu:

```
CLIPS> (deffunction is_number (?n)
                      (if  (numberp ?n)
                       then  (return ?n)
                       else  (printout  t  ?n " nu este un numar…" crlf) ) )
CLIPS> (is_number 102)
102
CLIPS> (is_number a9)
a9 nu este un numar…
```

Se poate întâmpla ca la definirea unei funcții numele acesteia să coincidă cu numele unei funcții system. In acest caz ne va fi semnalată această eroare, fiind necesară modificarea numelui funcției pentru ca programul să funcționeze.

```
CLIPS> (deffunction +(?x  ?y)
                (return (+ ?x ?y) ) )
[DFFNXPSR2] Deffunctions are not allowed to replace external function.
CLIPS> (deffunction ad (?x  ?y)
                (return (+ ?x ?y) ) )
CLIPS> (ad 2 4)
6
```

Special pentru a putea realiza o 'supraîncărcare' a operatorilor sau mai exact pentru a putea folosi numele unei funcții sistem a fost realizată construcția **defgeneric**.

2.3. Recursivitatea. Problema turnurilor din HANOI

O problemă clasică, care este rezolvată cu ajutorul recursivității este problema turnurilor din Hanoi.

Având trei suporturi (tije) şi pornind cu un număr de N discuri de diferite mărimi pe prima tijă se cere transferarea tuturor discurilor pe ultima tijă, ştiindu-se faptul că sunt legale doar mutările unui disc pe o tijă goală sau pe un disc mai mare.Prin urmare este considerată a fi o mutare ilegală mutarea unui disc mai mare peste unul mai mic decât acesta.

Numărul minim de mutări prin care se poate realiza transferul este $2^N - 1$. Vom reprezenta grafic modul cel mai scurt de rezolvare al problemei pentru cazul în care avem N=3 discuri.

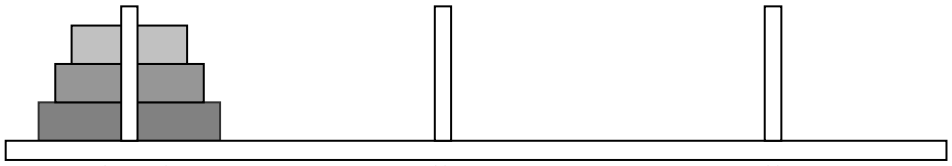


Fig.1 A(3,2,1) B( ) C( )

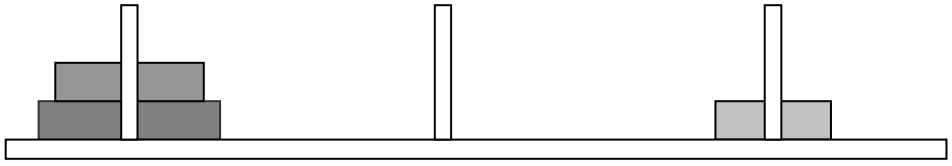


Fig.2 A(3,2) B( ) C(1)

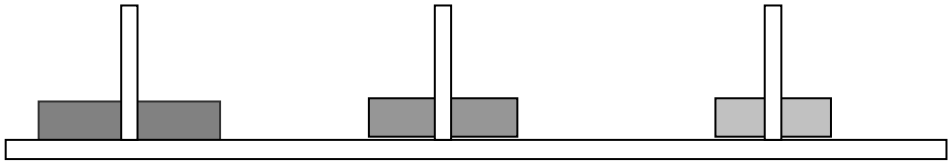


Fig.3 A(3) B(2) C(1)

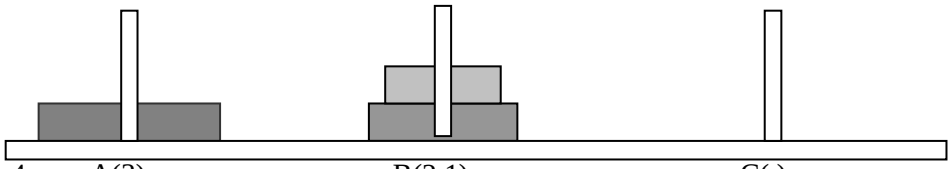


Fig.4 A(3) B(2,1) C( )

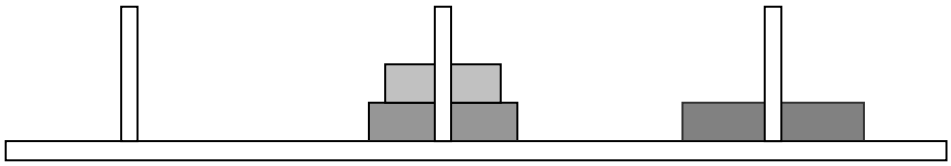


Fig.5 A( ) B(2,1) C(3)

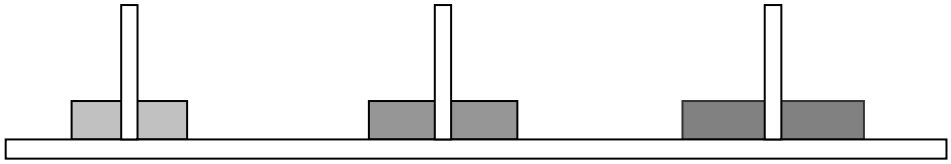


Fig.6 A(1) B(2) C(3)

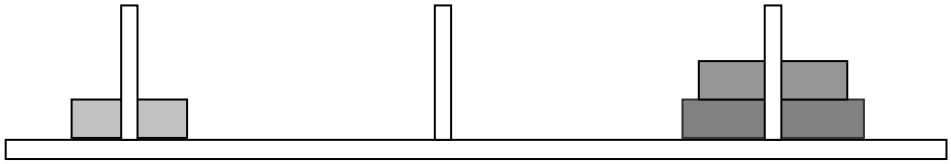


Fig.7 A(1) B( ) C(3,2)

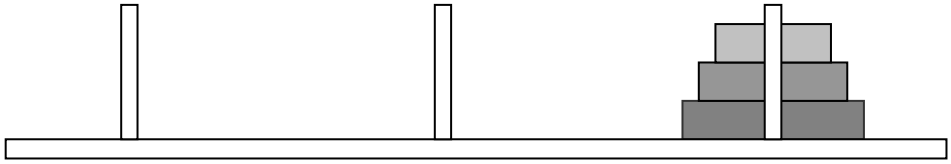


Fig.8 A( ) B( ) C(3,2,1)

După cum s-a observat pentru N=3 discuri s-au efectuat $2^3 - 1$, adică 7 mutări în ordinea care urmează : Disc1 A→C ; Disc2 A→B ; Disc1 C→B ; Disc3 A→C ; Disc1 B→A ; Disc2 B→C ; Disc1 A →C.

Pseudocodul funcţiei recursive este următorul:

```
funcţie Hanoi (n, start, end)                    // n –numărul de discuri
    dacă disc n>1
        atunci Hanoi(n-1, start , interm)
    sf-dacă
    afişează mesaj
    dacă disc n>1
        atunci Hanoi(n-1, interm, end)
    sf-dacă
sf-funcţie
```

Pe lângă funcţia recursivă *Hanoi* care va afişa mutările succesive de discuri vom introduce şi o funcţie *letter* care va face o conversie între litere şi cifre (transformând cifra 1 în litera A, cifra 2 în litera B, etc...). Această ultimă funcţie nu este absolut necesară fiind folosită doar pentru a înlocui în mesaje simbolurile numerice asociate tijelor cu litere (tija1<-> A , tija2 <-> B , tija3 <->C).

```
(deffunction letter (?no)
        (sub-string ?no ?no "ABCDEFGHIJKLMNOPQRSTUVXZ") )

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

(deffunction  hanoi (?n  ?tija_start  ?tija_end)
        (if (> ?n 1)
         then (hanoi  (- ?n 1)  ?tija_start   (- 6 ?tija_start ?tija_end) ) )
        (printout  t  "Discul "  ?n " "  (letter ?tija_start)  " -> " (letter ?tija_end) t)
        (if (> ?n 1)
         then (hanoi  (- ?n 1)  (- 6 ?tija_start ?tija_end)  ?tija_end) )
)
```

După cum se observă pentru obţinerea tijei intermediare s-a folosit o operaţie de scădere. Ştiind că avem în total 3 tije şi că suma tuturor simbolurilor numerice dă 6 (1+2+3), este lesne de înţeles de ce se scad din total tija_start şi tija_end pentru a o obţine pe cea intermediară. Funcţia recursivă folosită nu realizează nici o modificare asupra vreunei date, ea afişând doar mesaje.

Dacă se doreşte o implementare concretă a unor stive din care se scot şi se pun discuri este necesară o altă abordare a problemei. Exemplu de rulare:

```
CLIPS> (letter 1)
A
CLIPS> (letter 3)
C
CLIPS> (hanoi 3 1 3)              // 3-nr. discuri   1-tija_start   3-tija_end
Discul 1   A->C
Discul 2   A->B
Discul 1   C->B
Discul 3   A->C
```

Discul 1 B->A
Discul 2 B->C
Discul 1 A->C

Avantajele unei astfel de abordări a problemei sunt legate în special de viteza de rezolvare a problemei şi de libertatea oferită în alegerea tijei pe care sunt amplasate iniţial discurile şi a tijei destinaţie. Astfel dacă se doreşte se pot muta discurile de pe cea de-a doua tijă pe prima tijă, fiind permisă orice combinaţie posibilă. Apelul funcţiei va fi în acest caz **(hanoi N 2 1)** . Suplimentar s-ar putea realiza o verificare dacă tija de start nu coincide cu cea de final şi eventual s-ar putea stabili pentru N numărul de discuri o valoare minimă şi una maximă.

Se observă că ultimul mesaj afişat este simbolul FALSE, datorită faptului că în funcţia recursivă apelată nu s-a făcut nici o evaluare (s-a afişat doar un mesaj). Dacă se doreşte eliminarea simbolului FALSE se poate introduce la sfârşitul funcţiei un **(return)** fără a se specifica însă nici o valoare de returnat.

2.4. Întrebări

1. Care sunt facilităţile oferite de folosirea construcţiilor deffunction într-un program CLIPS? Din ce este format corpul unei astfel de construcţii şi care este valoarea pe care o returnează?
2. Numele unei funcţii definite de utilizator poate fi identic cu numele unei funcţii de sistem sau a unei funcţii generice (se poate realiza supraîncărcarea)? Se face diferenţa între o funcţie definită de utilizator şi una de sistem?
3. O construcţie *deffunction* trebuie întotdeauna declarată înainte de a fi apelată de o altă funcţie, obiect, regulă sau executată la top-level?
4. Care este numărul minim de parametri pe care îi acceptă? Numărul de parametri la apelul funcţiei trebuie să fie exact cât numărul de parametri specificat la definirea construcţiei? Argumentaţi.
5. Care dintre construcţiile *deffunction* următoare sunt corecte? Pot fi corecte toate sau nici una dintre acestea. Se introduc şi formele de apel ale funcţiilor.

```
(deffunction a  (printout t Mesaj))          CLIPS>(a)
(deffunction b  (printout t "Mesaj" t))      CLIPS>(b ())
(deffunction c ($?arg) (printout  t  $?arg))  CLIPS>(c  1 a "b" 2)
(deffunction d ($?arg) (printout  t  $?arg t))  CLIPS>(c  (create$ 1 a "b" 2))
```

6. Pentru construcţia deffunction următoare care este numărul minim de parametri pe care acesta trebuie să-i primească la apel? Argumentaţi acest răspuns.

```
(deffunction f (?a ?b ?c $?d)  (printout  t  ?a  "->"  ?b  "-->"  ?c   (first$ $?d)))
```

7. Pot sunt folosite mai multe variabile multicâmp ca parametri? Există o restricţie ca aceştia să nu fie alăturaţi? Care din exemplele următoare este de funcţii definite de utilizator sunt corecte?

```
(deffunction f (?a ?b $?c $?d)  (printout  t  ?a  "->"  ?b  "-->"  ?c   (first$ $?d)))
(deffunction f (?a $?b ?c $?d)  (printout  t  ?a  "->"  ?b  "-->"  ?c   (first$ $?d)))
```

8. Pentru a elimina erorile ce pot apare in urma introducerii unui număr incorect de parametri se pot folosi şi variabile libere în lista de argumente? (de exemplu o variabilă multiplă liberă)

   (deffunction f (?a ?b $?) (+ ?a ?b))

9. Care sunt cazurile în care este utilă folosirea unei funcţii fără acţiuni?
10. Cum putem realiza o recursivitate mutuală între două sau mai multe construcţii d*effunction*? Daţi un exemplu.
11. Ce se întâmplă atunci când se defineşte o construcţie deffunction al cărei nume este identic cu cel al unei funcţii declarate anterior?
12. Cum poate fi ştearsă o funcţie care este referită de o altă funcţie sau este apelată în RHS-ul unei reguli?
13. Care sunt funcţiile cu ajutorul cărora pot fi schimbate valorile unei variabile globale? Dar cele prin care putem şterge o variabilă globală?
14. Ce se întâmplă atunci când o variabilă dintr-o construcţie *defglobal* are acelaşi nume cu o variabilă deja definită într-o construcţie *defglobal* anterioară?
15. Care sunt diferenţele de notaţie între o variabilă globală şi o variabilă obişnuită?
16. Variabilele globale pot fi folosite în orice loc la fel ca o variabilă locală?
17. Poate fi utilizată o variabilă globală ca parametru pentru o construcţie *deffunction, defmethod* sau *message-handler* ?
18. Putem folosi o variabilă globală în acelaşi fel cu o variabilă locală în LHS-ul unei reguli pentru a o lega de o valoare? Argumentaţi.
19. Ce valoare au variabilele globale ?*x* şi ?*y* după execuţia fiecărei instrucţiuni în parte?

```
CLIPS>(defglobal ?*x* = 7
                 ?*y* = ?*x* )

CLIPS> (bind ?*x* 5)
CLIPS> (reset)
CLIPS> (set-reset-globals FALSE)
CLIPS> (bind ?*x* 5)
CLIPS> (reset)
CLIPS> (bind ?*x*)
```

20. Se poate defini o variabilă globală fără a explicita o valoare sau expresie? Care este în acest caz valoarea implicită pe care o dă CLIPS-ul unei variabile globale ce nu a fost definită explit ?

    (defglobal ?*x*)

21. Cum sunt diferenţiate variabilele globale de tip multicâmp (listă de valori) de cele care iau o singură valoare?
22. Cu ajutorul comenzii *(undefglobals *)* sunt şterse întotdeauna toate variabilelor globale? Dacă nu daţi un exemplu.

23. Ce face construcţia defglobals următoare?

```
(deffunction f (?*x*)
        (printout  t  ?*x*) )
```

Probleme rezolvate

1. Forma poloneză – Într-un exemplu anterior se încerca scrierea unei operaţii de adunare în forma infix, dificultatea constând în faptul că în CLIPS orice expresie se scrie în forma prefix. Acest program va încerca scrierea unei expresii care conţine operaţii de adunare, scădere, înmulţire şi împărţire, şi care conţine eventual şi paranteze din forma infix cea mai utilizată în forma prefix şi să calculeze valoarea acelei expresii. Deoarece CLIPS-ul acceptă doar scrierea expresiilor în forma prefix, evaluarea expresiei presupune scrierea acesteia în această formă respectând parantezele. Probleme principale constau în faptul că ordinea ode efectuare a operaţiilor este dată de priorităţile operatorilor şi parantezele ce apar în expresie. Eventualele simplificări ce s-ar putea realiza (de exemplu 1+2+3+4 => + 1 2 3 4) nu sunt realizate în această variantă. Ele pot fi uşor realizate verificându-se dacă avem doi sau mai mulţi operatori consecutivi de acelaşi tip. Se pot eventual introduce şi alte tipuri de operaţii, specificându-se pentru faptul *operator* corespunzător semnul şi prioritatea acesteia.

```
(deffacts date-initiale
        (prefix) (infix) (postfix) (rest)
        (operator "+" 0) (operator "-" 0)                ; operatii de prioritate 0
        (operator "*" 1) (operator "/" 1) )              ; operatii de prioritate 1

(defglobal ?*op* = (create$ "+" "-" "*" "/")
           ?*dig* = (create$ "0" "1" "2" "3" "4" "5" "6" "7" "8" "9")
           ?*sep* = (create$ " " "(" ")")
           ?*prefix* =""
           ?*postfix* ="" )

(deffunction verify (?str)
        (bind ?n 0)
        (loop-for-count (?i 1 (length ?str))
                (bind ?l (sub-string ?i ?i ?str))
                (if (not(member$ ?l (create$ ?*op* ?*dig* ?*sep*) ))
                 then (printout t "Caractere invalide !!!" t) (return FALSE))
                (if (eq ?l "(") then (bind ?n (+ ?n 1)) )
                (if (eq ?l ")") then (bind ?n (- ?n 1)) )
        )
        (if (= ?n 0)
         then (printout t "Expresie valida." t)
         else (printout t "Numar paranteze incorect !!!" t) (return FALSE) )
)
```

```
(deffunction simbol (?j ?str)
        (loop-for-count (?i ?j (length ?str))
                (if (neq (sub-string ?i ?i ?str) " ")
                 then (return ?i) ))
)
(deffunction number (?j ?str)
        (loop-for-count (?i ?j (length ?str))
                (if (not(member$ (sub-string ?i ?i ?str) ?*dig*))
                 then (return (- ?i 1)) ))
        (return (length ?str))
)
(deffunction expr (?j ?str)
        (bind ?n 0)
        (loop-for-count (?i ?j (length ?str))
                (bind ?l (sub-string ?i ?i ?str))
                (if (eq ?l "(") then (bind ?n (+ ?n 1)) )
                (if (eq ?l ")") then (bind ?n (- ?n 1)) )
                (if (= ?n 0) then (return ?i) ))
)
(deffunction token (?str)
        (bind $?op (create$ )) (bind ?j 0)
        (while (bind ?i (simbol (+ ?j 1) ?str))
                (bind ?s (sub-string ?i ?i ?str))
                (if (member$ ?s ?*dig*) then (bind ?j (number ?i ?str)) )
                (if (eq ?s "(") then (bind ?j (expr ?i ?str)) )
                (if (member$ ?s ?*op*) then (bind ?j ?i) )
                (bind $?op (create$ $?op (sub-string ?i ?j ?str))) )
        (return $?op)
)
```

Pentru început au fost stocate în variabilele globale *?*sep*, ?*op*, ?*dig** caracterele valide (respectiv separatorii spaţiu şi paranteza închisă şi deschisă, operatorii '+' , '-' , '/' , '*' şi cifrele de la 0 – 9). Funcţia *verify* va lua fiecare caracter introdus la intrare şi va verifica dacă acesta face parte din caracterele valide. De asemenea ea va număra parantezele deschise şi cel al parantezelor închise şi dacă acesta va fi egal va considera expresia ca fiind validă. Funcţia *simbol* caută în şirul primit ca argument *?str* pornind de la o poziţie *?j* primul caracter diferit de spaţiu. Funcţia *number* va extrage, pornind de la o poziţie specificată din şirul datelor se intrare un număr, returnând poziţia primului caracter care nu este o cifră. Funcţia *expr* va returna poziţia de sfârşit a unei expresii (adică paranteza închisă care corespunde primei paranteze deschise). În această funcţie se va ţine cont de faptul că în interiorul unei expresii pot exista şi alte expresii, astfel într-o buclă for se vor număra parantezele şi se va opri execuţia buclei (cu ajutorul comenzii *return*) în momentul în care s-a realizat o stare de echilibru. Funcţia *token*

administrează funcţiile enumerate anterior. În urma execuţiei acesteia şirul datelor de intrare este fragmentat în tipurile elementare menţionate mai sus (numere, expresii sau operatori). Se va returna o valoare multiplă de astfel de entităţi.

```
(defrule date_initiale
        =>
        (printout t "Introduceti expresia: ")
        (if (verify (bind ?str (readline)))
         then (assert (expr (token ?str)) ))
)
(defrule prioritate_operatori
        (declare (salience 10))
        ?f1 <- (expr ?o1 ?op1 ?o2 ?op2 ?o3 $?rest)
        (operator ?op1 ?p1)
        (operator ?op2 ?p2)
        ?f2 <- (prefix $?in)
        =>
        (retract ?f1 ?f2)
        (if (<= ?p1 ?p2)
         then (assert (prefix $?in ?op1 " " ?o1 " ")
                      (expr ?o2 ?op2 ?o3 $?rest)) )
        (if (> ?p1 ?p2)
         then (assert (prefix ?op2 " " $?in ?op1 " " ?o1 " " ?o2)
                      (expr ?o3 $?rest)) )
)
(defrule conversie_1
        ?f1 <- (expr ?o1 ?op ?o2)
        (operator ?op ?)
        ?f2 <- (prefix $?in)
        =>
        (retract ?f1 ?f2)
        (assert (prefix $?in " " ?op " " ?o1 " " ?o2))
)
(defrule conversie_2
        ?f1 <- (expr ?o)
        ?f2 <- (prefix $?in)
        =>
        (retract ?f1 ?f2)
        (assert (prefix $?in " " ?o))
)
```

Până în acest moment după ce în regula *date_initiale* datele de intrare sunt introduse, verificate, separate în subşiruri mai mici ce vor fi asertate într-un fapt *expr*, se studiază prioritatea operatorilor găsiţi. Astfel se iau doi operatori consecutivi şi în funcţie de prioritatea acestora se va forma forma prefix corespunzătoare faptului *expr* curent. Sunt tratate şi două cazuri ‘speciale’, primul în care expresia curentă de evaluat nu conţine decât un singur operator, caz în care

nu mai este necesară studierea priorităţii, introducându-se în faptul prefix mai întâi operatorul şi apoi operanzii; şi cel de-al doilea caz în care avem doar un operand (expresie sau număr) şi care este introdus în forma prefixată aşa cum este el.

```
(defrule expresii_interne
        (declare (salience -5))
        ?f1 <- (prefix $?first " " ?exp  $?last)
        ?f2 <- (rest $?r)
        (test (and (eq "(" (sub-string 1 1 ?exp))
                   (eq ")" (sub-string (str-length ?exp) (str-length ?exp) ?exp)) ))
        =>
        (retract ?f1 ?f2)
        (bind ?*prefix* (str-cat ?*prefix* (expand$ $?first)))
        (assert  (rest  $?last  $?r)   (prefix)
                 (expr (token (sub-string 2 (- (str-length ?exp)1) ?exp))) )
)

(defrule analizeaza_restul
        (declare (salience -5))
        ?f <- (rest  $?last &:(> (length$  $?last) 1) )
        (not (prefix  ?  $?))                    ; dacă expresia prefix e vidă
        =>
        (retract ?f)
        (assert (prefix  $?last)  (rest) )
)

(defrule adauga_la_forma_finala
        (declare (salience -10))
        ?f <- (prefix $?pr)
        =>
        (retract ?f)
        (bind ?*prefix* (str-cat ?*prefix* (expand$ $?pr) ))
)
```

Dacă în forma prefixată mai sunt expresii care nu au fost analizate (şiruri al căror prim caracter este o paranteză deschisă şi care se termină cu o paranteză închisă). Se poate considera că verificarea ultimului caracter (paranteza închisă) este inutilă, dat fiind faptul că funcţia *expr* returnează numai expresii valide ale căror număr de paranteze deschise coincide cu numărul parantezelor închise. Dacă mai există astfel de expresii, atunci în variabila globală *?*prefix** se păstrează datele deja analizate ce sunt situate înaintea expresiei găsite, în faptul *expr* se încarcă noua expresie ce trebuie analizată, iar în faptul *rest* se păstrează toate celelalte date. După ce această expresie a fost analizată, sau altfel spus dacă faptul prefix nu are nici măcar un singur element *(not (prefix ? $?))*, în regula *analizeaza_restul*, se asertează faptul prefix cu datele rămase, pentru ca acestea să poată fi şi ele analizate (dacă mai conţin vreo expresie cuprinsă între paranteze). Regula *adauga_la_forma_finala* are prioritate minimă şi se aprinde ultima în cazul

în care nu au mai rămas expresii de analizat şi datele conținute de faptul *prefix* pot fi introduse în variabila globală *?*prefix** fără a mai necesară efectuarea vre-unei modificări (se concatenează valorile deja existente cu simbolurile rezultate în urma expandării variabilei multiple).

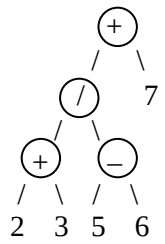
```
(defrule start_evaluare_expresie
        (declare (salience -20))
        =>
        (assert (eval (token ?*prefix*)))
)

(defrule adaugare_paranteze
        ?f <- (eval $?first ?op ?o1 ?o2 $?last)
        (operator ?op ?)
        (test (and (not(member$ ?o1 ?*op*)) (not(member$ ?o2 ?*op*)) ))
        =>
        (retract ?f)
        (assert (eval $?first (str-cat "(" ?op " " ?o1 " " ?o2 ")") $?last))
)

(defrule afisare_rezultate
        (eval ?str)
        =>
        (printout t "Forma prefix: " ?*prefix* t)
        (printout t "Forma de evaluare: " ?str t)
        (printout t "Valoarea expresiei: " (eval ?str) t)
)
```

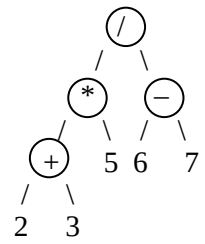
După ce în variabila globală *?*prefix** au fost introduse toate datele, în regula *start_evaluare_expresie* este asertat faptul *eval* în care pentru forma prefixată obținută trebuiesc introduse paranteze pentru a ajunge într-o formă ce va fi acceptată de CLIPS pentru evaluare. În cazul de față lucrurile sunt foarte simple, deoarece s-a folosit pentru reprezentarea expresiei un arbore binar, fiecare operator urmat de doi operanzi este încadrat între două paranteze.

Introduceti expresia: (2+3) / (5-6) +7



Forma prefix: + / + 2 3 – 5 6 7
Forma de evaluare: (+(/ (+ 2 3) (- 5 6))7)
Valoarea expresie: 2.0

Introduceti expresia: ((2+3)*5) / (6-7)



Forma prefix: / * + 2 3 5 - 6 7
Forma de evaluare: (/ (*(+ 2 3)5)(- 6 7))
Valoarea expresie: -25.0

În ultima regulă *afisare_rezultate* nu trebuie făcută confuzie între faptul *eval* din LHS-ul regulii şi funcţia **eval** care returnează valoarea la care a fost evaluată expresia dată de şirul de caractere primit ca argument. Această funcţie evaluează doar funcţii, indiferent dacă acestea sunt funcţii de system sau sunt definite de utilizator.

```
CLIPS> (eval "(+ 2 (* 3 4))" )
14
CLIPS> (deffunction pow (?a ?b)
                  (** ?a ?b))
CLIPS> (eval "(pow 2 5)" )
32.0
```

2. <u>Text scris în "pasăreşte"</u> – Regula de conversie a unui text normal în 'păsărească' şi invers este foarte simplă. Acestă regulă constă în înlocuirea fiecărei vocale *v* cu grupul *vPv*. De exemplu:

"Paparepe apalapandapalapa"	=>	"Pare alandala"
"Cola este rece"	=>	"Copolapa epestepe repecepe"

Programul va executa şi operaţia de scriere şi de citire a unui text scris în 'păsăreşte'. Textul va fi citit dintr-un fişier şi eventual se vor salva rezultatele într-un alt fişier. Elementul de noutate este bucla în care se stă atât timp cât fişierul din care se citesc datele de intrare nu există. Pentru fişierul în care se salvează datele nu este necesară existenţa unei astfel de bucle, deoarece un fişier deschis pentru scriere este creat în mod automat dacă nu există.

```
(deffunction first (?char ?string)
        (bind ?poz (str-index ?char ?string))
        (if (and (numberp ?poz) (<= ?poz (length ?string)) )
         then (return (sub-string 1 ?poz ?string))
         else (return FALSE))
)
(deffunction rest (?char ?string)
        (bind ?poz (str-index ?char ?string))
        (if (and (numberp ?poz) (< ?poz (length ?string)) )
         then (return (sub-string (+ ?poz 1) (length ?string) ?string))
         else (return ""))
)
(defrule deschidere_fisier
        (declare (salience 10))
        =>
        (printout t "Numele fisierului : " ) (bind ?n (read))
        (while (eq (open ?n f) FALSE)
                (printout t "Eroare la deschiderea fisierului " ?n t)
                (printout t "Numele fisierului : " ) (bind ?n (read)) )
        (set-strategy breadth)
)
```

```
(defrule continut_fisier
        =>
        (printout t "Doriti :" t "1. Scriere in pasareasca." t )
        (printout t "2. Citire din pasareasca." t "Optiune : ")
        (while (neq (bind ?r (read)) 1 2)
                (printout t "Alegeti (1 sau 2) : ") )
        (assert (optiune ?r))
)

(defrule citire_propozitii_din_fisier
        =>
        (bind ?string "")
        (while (neq (bind ?word (read f)) EOF)
                (if (and (not (numberp ?word)) (str-index . ?word))
                 then   (assert (string (str-cat ?string " " (first . ?word)) ))
                        (bind ?string (rest . ?word))
                 else
                        (bind ?string (str-cat ?string " " ?word)) ) )
)

(defrule inchidere_fisier
        (declare (salience -10))
        =>
        (set-strategy depth)
        (close)
)
```

Funcţia *first* caută apariţia unui caracter specificat *?char* într-un şir *?string*, şi în caz de succes returnează subşirul inclus între primul caracter şi caracterul căutat inclusiv. Funcţia *rest* după cum sugerează şi numele său, returnează subşirul ce porneşte de la caracterul căutat până la ultimul caracter din şirul *?string* (mai puţin caracterul specificat). Funcţia *(set-strategy breadth)* apelată în RHS-ul regulii *deschidere_fisier* are rolul de a schimba modul de aprindere a regulilor şi în acest program singurul unul din efectele pe care îl produce este faptul că utilizatorul va fi interogat la începutul programului în legătură cu salvarea rezultatelor obţinute într-un fişier (regula *deschidere_fisier_salvare* nu se mai aprinde la sfârşit). Prin schimbarea strategiei poate fi influenţată şi ordinea în care sunt salvate datele.

Utilitatea funcţiilor *first* şi *last* se observă în special în interiorul regulii *citire_propozitii_din_fisier*, unde din fişierul de intrare specificat de utilizator sunt citite propoziţiile una câte una. Se caută apariţia caracterului '.' ce marchează sfârşitul unei propoziţii. Nu mai există restricţia ca în fişierul de intrare datele să fie grupate în propoziţii aşezate câte una pe fiecare linie (textul poate fi cursiv), şi de asemenea în cazul în care a fost omisă introducerea unui spaţiu înainte sau după '.' se va putea face delimitarea celor două propoziţii. De exemplu:

```
"Buna.Vreau sa vorbesc cu Dan."  =>  (string "Buna.")
                                     (string "Vreau sa vorbesc cu Dan."
```

Se poate observa de asemenea faptul că sunt citite toate propoziţiile din fişier odată , până când se ajunge la sfârşitul fişierului EOF.

```
(defrule modificare_in_pasareasca
        (optiune 1)
        (string ?str)
        =>
        (bind ?new "")
        (loop-for-count (?i 1 (length ?str))
                (bind ?l (sub-string ?i ?i ?str))
                (if (str-index ?l "aeiou")
                 then (bind ?new (str-cat ?new ?l p ?l))
                 else (bind ?new (str-cat ?new ?l)) ) )
        (assert (new_string ?new))
)
(defrule citire_din_pasareasca
        (optiune 2)
        (string ?str)
        =>
        (bind ?new "") (bind ?j 1)
        (loop-for-count (?i 1 (length ?str))
                (if (= ?j ?i)
                 then (bind ?l (sub-string ?i ?i ?str))
                      (if (str-index ?l "aeiou")
                        then (bind ?j (+ ?j 3))
                        else (bind ?j (+ ?j 1)) )
                      (bind ?new (str-cat ?new ?l)) ) )
          (assert (new_string ?new))
)
```

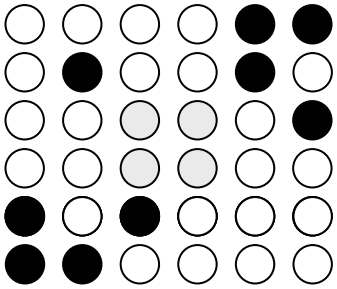
Datorită numărului de pattern-uri foarte mic cele două reguli de citire şi scriere pot fi foarte bine transformate în două funcții ce primesc la intrare ca parametru şirul ?str (propoziția ce trebuie transformată).

```
(defrule afisare_propozitie_modificata
        (new_string ?new)
        =>
        (printout t ?new t))

(defrule deschidere_fisier_salvare
        =>
        (printout t "Doriti salvare rezultate ? ")
        (if (neq (read) Y y YES Yes yes D d DA Da da)
         then (assert (salvare NO))
         else (assert (salvare YES))
              (printout t "Numele fisierului de salvare : " )
        (open (read) s "w")) )

(defrule salvare_rezultate
        (salvare YES)
        (new_string ?new)
        =>
        (printout s ?new t) )
```

3. <u>Problema monezilor</u> - Într-un pătrat se dispun simetric 36 monede (6x6). Să se scoată 9 monede în aşa fel încât pe fiecare dintre cele 4 rânduri exterioare (câte două de fiecare latură să rămână câte 4 monede).



Soluție pentru cazul în care sunt eliminate 9 monezi

După cum se poate observa în această problemă există o zonă centrală (2x2) din care nu se extrage niciodată nici o monedă. Pentru a se simplifica forma regulii de căutare a soluțiilor este suficient să observăm că afirmația că pe fiecare latură exterioară rămân 4 monede, este identică cu afirmația că de pe fiecare latură exterioară se scot 2 monede (după cum se poate observa şi în soluția reprezentată mai sus). Regula *generare_piese* realizează într-o formă elegantă (specifică CLIPS-ului) o matrice pătratică, asertând toate cele 6x6=36 elemente ale sale. Se poate folosi pentru acesta o simplă construcție deffunction sau o regulă în care sunt introduse două bucle for imbricate.

```
(deffacts date
        (numere 1 2 3 4 5 6))

(defrule generare_piese
        (numere $? ?n1 $?)
        (numere $? ?n2 $?)
        =>
        (assert (moneda ?n1 ?n2)) )
```

```
(defrule date_initiale
        => (generare 6) )

(deffunction generare (?n)
      (loop-for-count (?i 1 ?n?)
            (loop-for-count (?j 1 ?n?)
                  (assert (moneda ?i ?j)) ))
)
```

```
(deffunction validate ($?v)
      (bind ?len (length$ $?v))
      (loop-for-count (?i 1 4)
            (bind ?j (nth$ ?i (create$ 1 2 5 6)))
            (bind ?poz1 (member$ ?j $?v))
            (if (numberp ?poz1)
             then (bind ?poz2 (member$ ?j (subseq$ $?v (+ ?poz1 1) ?len)))
            (if (or (eq FALSE ?poz2)
                  (numberp (member$ ?j (subseq$ $?v (+ ?poz1 ?poz2 1) ?len))))
             then (return FALSE) )
             else (return FALSE) ) )
      (return TRUE)
)
```

```
(deffunction afisare (?str)
        (printout t "Monede : " ?str t)
        (loop-for-count (?i 1 6)
                (loop-for-count (?j 1 6)
                        (if (str-index (str-cat ?i ?j) ?str)
                         then (printout t "o")                    ; caracterul 'o'
                         else (printout t "          ") ))        ; caracterul TAB
                (printout t t) )
        (printout t "Apasa o tasta..." t t) (readline)
)
(defrule scoate_9_monede
        (moneda 1 ?c1) (moneda 1 ?c2&:(> ?c2 ?c1))
        (moneda 2 ?c3) (moneda 2 ?c4&:(> ?c4 ?c3))
        (moneda 5 ?c5) (moneda 5 ?c6&:(> ?c6 ?c5))
        (moneda 6 ?c7) (moneda 6 ?c8&:(> ?c8 ?c7))
        (moneda ?l9&3|4 ?c9&1|2|5|6)
        (test (validate (create$ ?c1 ?c2 ?c3 ?c4 ?c5 ?c6 ?c7 ?c8 ?c9)))
        =>
        (afisare (str-cat 1 ?c1 " " 1 ?c2 " " 2 ?c3 " " 2 ?c4 " " 5 ?c5 " " 5 ?c6 " "
                        6 ?c7 " " 6 ?c8 " " ?l9 ?c9))
)
```

Aceasta este una dintre problemele, care se pretează a fi rezolvate într-un mediu precum este CLIPS-ul datorită uşurinței prin care aceasta poate fi rezolvată. În regula cu ajutorul căreia se caută soluțiile problemei se specifică doar cunoştințele generale din enunțul problemei. Programul caută şi afişează toate soluțiile găsite şi aceasta fără a fi necesar un algoritm sofisticat.

Funcția *validate* returnează simbolul TRUE dacă în valoarea multiplă pe care o primeşte ca parametru, orice element din mulțimea {1, 2, 5, 6} se repetă de două ori, sau altfel spus dacă de pe prima, a doua , ultima şi penultima coloană au fost eliminate doar două piese. Dacă numărul de apariții a unui element este diferit de doi, înseamnă că nu sunt satisfăcute cerințele problemei, caz în care funcția va returna simbolul FALSE. Se poate verifica modul în care funcționează funcția direct de la prompter.

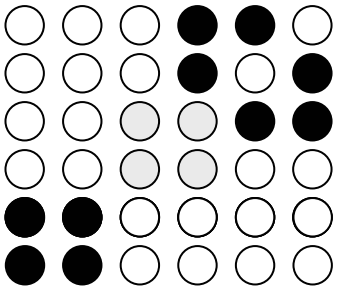
```
CLIPS> (validate 2 1 2 5 6 a 1 6 5)
TRUE
CLIPS> (validate 3 3 3 3 1 1 2 2 5 5 6 6)
TRUE
CLIPS> (validate 1 1 2 2 5 6 6)
FALSE
```

Funcția *validate* este realizată special pentru a verifica coloanele de pe care au fost eliminate piese. După cum se observă în regula *scoate_9_monede,* se specifică în mod explicit faptul că se vor elimina câte două piese de pe fiecare linie exterioară (cum se specifică şi în enunțul problemei), şi că cea de noua piesă va situată pe a treia sau a patra linie şi pe coloana 1,2,5 sau 6 (adică va aparține doar unei singure laturi exterioare).

```
(moneda 2 ?c3)    (moneda 2 ?c4&:(> ?c4 ?c3))        ;piese diferite
```

Prin modul în care au fost concepute pattern-urile regulii se asigură unicitatea pieselor eliminate, sau altfel spus se asigură faptul că nu va fi eliminată aceiaşi piesă de două sau mai multe ori (cu ajutorul condiţiei *(> ?c4 ?c3),* care forţează eliminarea pieselor ce sunt pe aceiaşi linie dar pe coloane diferite). Împreună cu funcţia *validate* se asigură astfel toate condiţiile necesare obţinerii unei soluţii valide.

Ştiindu-se faptul că de pe fiecare latură exterioară a pătratului trebuiesc scoase doar 2 monede, şi că în total vor fi eliminate 9 monezi, se deduce faptul că unele monede vor aparţine mai multor laturi exterioare (în cazul nostru 7 monezi se vor găsi de două laturi exterioare simultan şi 2 monezi vor aparţine doar unei singure laturi exterioare fiecare). Dacă s-ar fi încercat eliminarea a 10 monezi (numărul acestora s-ar schimba în 6 şi respectiv 4 monezi).



Soluţie pentru cazul în care sunt eliminate 10 monezi

În regula care furnizează soluţiile pentru cazul în care se scot 10 monede trebuie adăugată o condiţie suplimentară, astfel monezile 9 şi 10 nu trebui să fie situate pe aceiaşi poziţie (să nu fie pe aceiaşi linie sau pe aceiaşi coloană).

```
(defrule scoate_10_monede
        (moneda 1 ?c1) (moneda 1 ?c2&:(> ?c2 ?c1))
        (moneda 2 ?c3) (moneda 2 ?c4&:(> ?c4 ?c3))
        (moneda 5 ?c5) (moneda 5 ?c6&:(> ?c6 ?c5))
        (moneda 6 ?c7) (moneda 6 ?c8&:(> ?c8 ?c7))
        (moneda ?l9&3|4 ?c9&1|2|5|6)
        (moneda ?l10&3|4 ?c10&1|2|5|6)
        (test (or (neq ?l9 ?l10) (neq ?c9 ?c10)))
        (test (validate (create$ ?c1 ?c2 ?c3 ?c4 ?c5 ?c6 ?c7 ?c8 ?c9 ?c10)))
        =>
        (afisare (str-cat 1 ?c1 " " 1 ?c2 " " 2 ?c3 " " 2 ?c4 " " 5 ?c5 " " 5 ?c6 " "
                          6 ?c7 " " 6 ?c8 " " ?l9 ?c9 " " ?l10 ?c10))
)
```

Dacă numărul de monezi eliminate creşte la 11 (se pot elimina respectând cerinţele problemei un număr maxim de 12 monede), formularea condiţiilor de unicitate în acelaşi mod ca şi în regula anterioară devine anevoioasă. Se pot însă compara doar adresele faptelor şi astfel se poate ajunge la un număr de 2 condiţii:

```
(test (and (neq ?f1 ?f2 ?f3)  (neq ?f2 ?f3))
```

```
(defrule scoate_11_monede
        (moneda 1 ?c1) (moneda 1 ?c2&:(> ?c2 ?c1))
        (moneda 2 ?c3) (moneda 2 ?c4&:(> ?c4 ?c3))
        (moneda 5 ?c5) (moneda 5 ?c6&:(> ?c6 ?c5))
        (moneda 6 ?c7) (moneda 6 ?c8&:(> ?c8 ?c7))
        ?f1 <- (moneda ?l9&3|4 ?c9&1|2|5|6)
        ?f2 <- (moneda ?l10&3|4 ?c10&1|2|5|6)
        ?f3 <- (moneda ?l11&3|4 ?c11&1|2|5|6)
        (test (and (neq ?f1 ?f2 ?f3) (neq ?f2 ?f3)) )
        (test (validate (create$ ?c1 ?c2 ?c3 ?c4 ?c5 ?c6 ?c7 ?c8 ?c9 ?c10 ?c11)))
        =>
        (afisare (str-cat 1 ?c1 " " 1 ?c2 " " 2 ?c3 " " 2 ?c4 " " 5 ?c5 " " 5 ?c6 " "
                          6 ?c7 " " 6 ?c8 " " ?l9 ?c9 " " ?l10 ?c10 " " ?l11 ?c11))
)
```

Timpul în care sunt găsite soluțiile cresc odată cu numărul monezilor ce se doreşte a fi eliminate, deoarece cresc şi numărul pattern-urilor şi a condițiilor necesare pentru a fi asigurată unicitatea pieselor eliminate

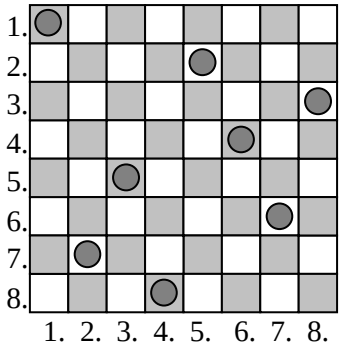
Se mai precizează faptul că în funcția de afişare, pentru a vizualiza monezile rămase şi locurile goale s-au folosit caracterele 'o' şi TAB-ul. Efectul care se obține în momentul în care este afişat un caracter TAB pe ecran este acela al unui pătrat negru. Pentru a determina poziția din care a fost eliminată o monedă s-a folosit funcția *(str-index (str-cat ?i ?j) ?str)*, care returnează simbolul FALSE dacă linia şi coloană concatenate nu aparțin şirului datelor primite de funcție ca argument la intrare, sau valoarea primei poziții în care a fost găsit subşirul.

Dacă se doreşte selectarea numărului de monede ce vor fi eliminate, de către utilizator de la tastatură, se poate realiza regula următoare. Pentru fiecare regulă *scoate_X_monede* se va introduce în LHS-ul regulii un fapt de control *nr_monezi*, cu ajutorul cărora se va activa o singură regulă de căutare a soluțiilor.

```
(defrule număr_monezi
        (declare (salience 10))
        =>
        (printout t "Cate monezi doresti să fie scoase [9-11]]? ")
        (while (or (not (integerp (bind ?n (read)) )) (< ?n 9) (> ?n 11))
                (printout t "Indroduceti un intreg intre 9 şi 11: ") )
        (assert (nr_monezi ?n))
)
```

3. Problema damelor (rezolvare recursivă) Pentru a rezolva această problemă au fost implementate un set de trei funcții *verify, solution* şi *dame*. Se folosesc două variabile globale *?*n** pentru a stoca dimensiunea tablei şi *?*sol** pentru a număra soluțiile. Dacă nu s-ar fi folosit aceste variabile, ar fi trebuit să introducem pentru funcții noi parametri. Funcția recursivă *dame* primeşte inițial ca argument lista vidă. Variabila *$?val* va stoca poziția elementelor pe coloane,

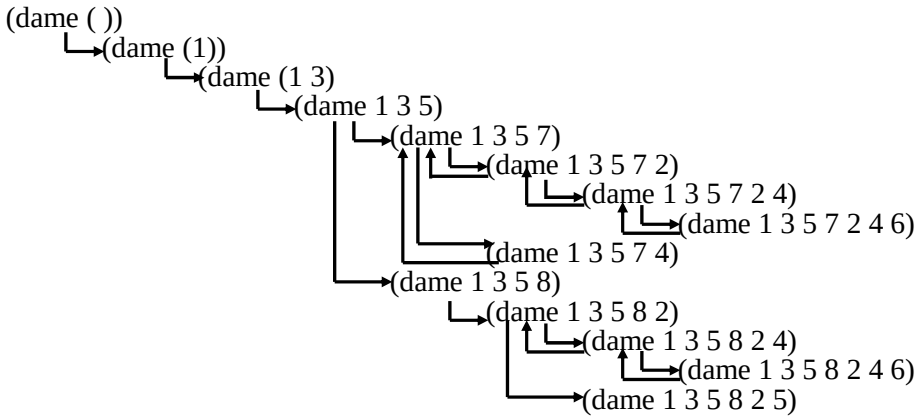
poziția pe linie fiind dată de indexul elementului. Astfel dacă *$?val=(1 3 5)*, putem spune că au fost poziționate trei dame pe pozițiile (1,1) (2,3) şi (3,5).



*?*n*=8* dimensiunea tablei
*?*sol*=1* prima soluție
$?val = (1 5 8 6 3 7 2 4) vector coloane

În regula *verify* se testează practic dacă elementul curent de coordonate *?i* şi *?j* nu este pe diagonală cu nici una din damele deja amplasate. Dacă suma (sau diferența) dintre liniile şi coloanele a două elemente este egală, atunci acestea se află pe aceiaşi diagonală principală (sau secundară) şi funcția *verify* va returna simbolul FALSE *(or (eq (- ?j ?i) (- ?c ?k)) (eq (+ ?j ?i) (+ ?c ?k)))*. Condiția ca două dame se afle pe două coloane diferite este realizată în funcția *dame* prin verificarea valorii de adevăr a expresiei *(not (member$?j $?val))*. Nu este necesară verificarea liniilor, datorită modului în care a fost abordată problema. Lungimea vectorului *$?val* +1 reprezintă linia curentă. În momentul în care se nu se mai poate amplasa nici o damă, cu ajutorul funcției *(return)*, se revine la funcția anterioară apelului, în care variabila *$?val* avea dimensiunea mai mică (un câmp mai puțin). Acesta este unul din marile avantaje ale recursivității şi este posibil datorită faptului că pentru fiecare funcție se alocă o zonă specială de memorie.

Pentru a observa mai bine modul în care se autoapelează funcția *dame*, vom reprezenta apelurile recursive prin săgeți, specificând şi parametrii de apel. După cum se poate observa valoarea multicâmpului *$?val* este diferită pentru fiecare funcție *dame*, ea comportându-se ca o variabilă locală. La revenirea dintr-o funcție se refac automat şi la valorile variabilelor locale.



```
(defglobal ?*n* = 8
           ?*sol* = 0)

(deffunction verify (?i ?j $?val)
        (bind ?len (length$ $?val))
        (loop-for-count (?k 1 ?len)
                (bind ?c (nth$ ?k $?val))                    ; ?k linia, ?c coloana
                (if  (or (eq (- ?j ?i) (- ?c ?k))            ; diag. principală diferită
                         (eq (+ ?j ?i) (+ ?c ?k)) )          ; diag. secundară diferită
                 then (return FALSE)
                )
        )
        (return TRUE)
)

(deffunction solution ($?val)
        (loop-for-count (?i 1 (length $?val))
                (bind ?col (nth$ ?i $?val))
                (loop-for-count (?i 1 (length $?val))
                        (if (eq ?i ?col) then (printout t o) else (printout t .))
                )
                (printout t t)
        )
        (printout t t "Solutia " (bind ?*sol* (+ ?*sol* 1)) " Apasa <Enter>...")
        (readline)
)

(deffunction dame ($?val)
        (bind ?len (length$ $?val))
        (loop-for-count (?j  1 ?*n*)
                (if (not (member$ ?j $?val)) then                    ;coloane diferite
                        (if (verify (+ ?len 1) ?j $?val) then
                                (dame (create$ (expand$ $?val) ?j))
                        )
                )
        )
        (if (eq ?len ?*n*)  then  (solution $?val) )
        (return)
)

(defrule start_program
        =>
        (printout t "Dimensiunea tablei (3-10):" )
        (while (and (not(integerp (bind ?*n* (read t)))) (> ?*n* 10) (< ?*n* 3))
                (printout t "Dimensiunea tablei (3-10):" ) )
        (dame)
        (printout t "Doriti reluare (Y/N):" )
        (if (neq (read t) Y y yes Yes YES D d da Da DA)
         then else (refresh start_program) )
)
```

2.6. Exemple de funcţii definite de utilizator

Funcţia *last$* returnează ultimul element dintr-un multicâmp (similar cu *first$*).

```
(deffunction last$ ($?val)
        (nth$ (length$ ?val) $?val) )
```

Funcţia *pop$* returnează toate elementele mai puţin ultimul (similar cu *rest$*).

```
(deffunction pop$ ($?val)
        (subseq$ $?val 1 (- (length$ $?val) 1)) )
```

Funcţia *count$* numără elementele unui multicâmp (identic cu *length$*).

```
(deffunction count$ ($?arg)
        (length$ $?arg) )
```

Funcţia *sign* returnează 1 dacă numărul e pozitiv, -1 dacă e negativ sau 0.

```
(deffunction sign (?num)
        (if (> ?num 0) then (return 1))
        (if (< ?num 0) then (return -1))
        0 )
```

Funcţia *hypotenuse* calculează ipotenuza unui triunghi dreptunghic.

```
(deffunction hypotenuse (?a ?b)
        (sqrt (+ (* ?a ?a) (* ?b ?b))) )
```

Funcţia *day* returnează numele zilei pentru un număr dat.

```
(deffunction day (?d)
        (switch ?d
                (case 1 then Luni)
                (case 2 then Marti)
                (case 3 then Miercuri)
                (case 4 then Joi)
                (case 5 then Vineri)
                (case 6 then Sambata)
                (case 7 then Duminica)
                (default  Eroare) ) )
```

Funcţia *hex* va returna simbolul TRUE, dacă este cifră hexazecimală sau FALSE

```
(deffunction hex (?x)
        (if (neq ?x 0  1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f)
         then FALSE
         else TRUE ) )
```

Funcţiile *deschide* şi *citeşte* realizează deschiderea unui fişier specificat de utilizator *(deschide "numar.dat")* şi asertarea tuturor cuvintelor găsite.

```
(deffunction citeste (?n)
        (if (neq (bind ?w (read file)) EOF)
         then (return ?w)) )
```

```
(deffunction deschide (?n)
        (open ?n file)
        (while (bind ?w (citeste ?n)) do
                (assert (word ?w)) )
        (close file) )
```

Funcţia *fibonacci* este recursivă şi afişează numerele din şirul lui Fibonacci după relaţia de recurenţa $a_{n+2} = a_{n+1} + a_n$, unde $a_1 = 1$ şi $a_1 = 2$. Funcţia trebuie apelată cu *(fibonacci 1 2)*.

```
(deffunction fibonacci (?d1 ?d2 $?data)
    (if (< (length$ $?data) 30)
     then (printout t (bind ?d (+ ?d1 ?d2)) t)
  (return (fibonacci ?d ?d1 ?d2 $?data))
    else (return)  )
)
```

```
(deffacts init  (data 1 2))

(defrule fibonacci
    ?f <- (data $?v ?n1 ?n2)
    (test (< (length$ $?v) 30))
    =>
    (retract ?f) (bind ?d (+ ?d1 ?d2))
    (assert (data $?v ?d1 ?d2 ?d)) )
```

Funcţia *substring* extrage primul subşir ce este inclus şi în primul şi în al doilea şir. Dacă de exemplu se apelează funcţia cu parametrii *"omida", "mineriada" => "mi"*

```
(deffunction substring (?s1 ?s2)
    (loop-for-count (?i 1 (length ?s1))
        (if (str-index (sub-string ?i ?i ?s1) ?s2)
         then (loop-for-count (?j (+ ?i 1) (length ?s1))
                  (if (str-index (sub-string ?i ?j ?s1) ?s2)
          then (bind ?poz ?j)
                   else (break) ) )
              (return (sub-string ?i ?poz ?s1))
)   )   )
```

Dacă se doreşte extragerea unei sub-secvenţe din două multicâmpuri nu putem defini o funcţie de genul *(deffunction sub-set ($?s1 $?s2)...*, deoarece nu se permite introducerea altor parametri după o variabilă multifield. Problema poate însă fi foarte uşor rezolvată cu ajutorul unei reguli. Dacă avem datele *(data1 a b c d e f)* şi *(data2 x 1 c d e a)* atunci va fi găsite toate sub-secvenţele *(c), (c d), (d), (a).*

```
(defrule sub-set
        (data1  $?  $?s  $?)
        (data2  $?  $?s &:(<> (length$ $?s) 0)  $?)
        =>
        (printout  t  $?s  t) )
```

Funcţia recursivă *conv* realizează trecerea unui număr într-o bază. Se afişează după fiecare apel al funcţiei resturile împărţiri. De exemplu pentru *(conv 14 2) => 1110*.

```
(deffunction conv (?număr ?baza)
        (bind ?cat (div ?numar ?baza) )
        (if (<> ?cat 0)
         then (conv  ?cat  ?baza) )
        (printout t (mod ?numar ?baza) )
```