Cap4. Funcţii generice. Supraîncărcarea funcţiilor

4.1. Construcţiile DEFGENERIC şi DEFMETHOD

Funcţiile generice sunt similare cu deffunction, în sensul că pot fi utilizate pentru a introduce nou cod procedural direct în CLIPS şi pot fi apelate ca orice altă funcţie. Aceste funcţii sunt însă mai puternice, deoarece pot fi supraîncărcate. Dacă pentru o funcţie obişnuită apariţia unei noi definiţii a acesteia ar fi dus la suprascrierea funcţiei (ştergerea variantei anterioare), pentru funcţiile generice putem avea mai multe funcţii cu acelaşi nume, dar care pot face lucruri diferite depinzând de tipul (sau clasa) şi numărul lor de argumente. Vom numi **metodă** fiecare componentă a unei astfel de funcţii generice multiple. Prin urmare se spune despre o funcţie generică care are mai mult de o metodă că este supraîncărcată. CLIPS-ul va fi acela care va căuta în funcţie de argumentele transmise metoda cea mai apropiată ce va fi executată. Datorită timpului de căutare a metodei aplicabile performanţele sistemului scad cu 15%-20% şi din acest motiv nu este recomandată folosirea funcţiilor generice în rutinele pentru care timpul este critic (cum ar fi de exemplu buclele). Sintaxa construcţiei este următoarea:

(**defgeneric** <name> [<comment>])

(**defmethod** <name> [<index>] [<comment>]
(<parameter-restriction>* [<wildcard-parameter-restriction>])
<action>*)

<parameter-restriction> ::= <single-field-variable> |
(<single-field-variable> <type>* [<query>])

<wildcard-parameter-restriction> ::= <multifield-variable> |
<multifield-variable> <type>* [<query>])

<type> ::= <class-name>
<query> ::= <global-variable> | <function-call>

O funcţie generică este formată dintr-un header (similar cu o declaraţie anterioară) şi una sau mai multe metode. Header-ul funcţiei generice poate fi declarat explicit de către utilizator sau implicit prin definirea a cel puţin unei metode (are un caracter opţional). O metodă este formată din şase elemente: un nume (prin care se va identifica funcţia generică căreia aparţine metoda), un index opţional, un comentariu opţional, un set de parametri restrictivi, un parametru multiplu opţional care va permite utilizarea unui număr variabil de argumente şi o secvenţă de acţiuni sau expresii care vor fi executate pe rând în momentul în care metoda va fi apelată. Parametri restrictivi sunt utilizaţi pentru a determina metoda aplicabilă pentru un set de argumente, atunci când funcţia generică este apelată.

O funcţie generică trebuie să fie declarată, printr-un header sau o metodă, înainte ca aceasta să fie apelată de altă metodă a unei funcţii generice, alt obiect,

regulă sau lansată în execuţie la top-level. Ca şi în cazul funcţiilor definite excepţie de la aceasta fac funcţiile generice recursive (care se autoapelează).

O metodă este identificată în mod unic prin nume sau prin index sau prin nume şi parametri restrictivi. Fiecărei metode a unei funcţii generice îi este asociat un index întreg unic în grupul tuturor metodelor pentru acea funcţie generică. Dacă o nouă metodă are acelaşi nume şi exact aceiaşi parametri restrictivi ca altă metodă, CLIPS-ul va înlocui automat noua metodă. Orice diferenţă în parametrii restrictivi va duce la definirea unei noi metode pe lângă celelalte metode. Pentru a înlocui o metodă deja existentă cu o alta nouă care are parametri restrictivi diferiţi se poate specifica explicit în definiţia noii metode indexul metodei ce se doreşte a fi înlocuite. Indexul asignat automat de CLIPS este unic (diferit de cel al metodelor existente pentru aceiaşi funcţie generică) şi poate fi determinat cu ajutorul comenzii **list-defmethods**.

Fiecare parametru al unei metode poate avea la definire un număr arbitrar de restricţii, nici una sau toate restricţiile. Un parametru restrictiv este aplicat argumentului funcţiei generice în momentul rulării pentru a determina în particular ce metoda va accepta acel argument. Un parametru poate avea două tipuri de restricţii: de tip sau de interogare. O restricţie de tip se referă la clasele argumentelor ce vor fi acceptate pentru un parametru. O restricţie de interogare este un test logic definit de utilizator care trebuie să fie satisfăcut pentru ca un argument să fie acceptabil. Complexitatea parametrilor restrictivi afectează în mod direct viteza de găsire a unei metode.

Un parametru care nu are nici o restricţie semnifică că acea metodă va accepta orice argument în acea poziţie. Fiecare metodă ar trebui să aibă dacă este necesar parametri restrictivi care să o facă distinctivă între toate celelalte metode. Dacă pentru o funcţie generică apelată nu este găsită nici o metodă aplicabilă, CLIPS-ul va genera o eroare pentru a semnala acest lucru.

O restricţie de tip permite utilizatorului să specifice o listă de clase, din care trebuie să se potrivească cu una (sau să fie superclasă a uneia) clasa argumentului funcţiei generice. Dacă COOL nu este instalat în configuraţia actuală a CLIPS-ului singurele tipuri (clase) permise sunt : OBJECT, PRIMITIVE, LEXEME, SYMBOL, STRING, NUMBER, INTEGER, FLOAT, MULTIFIELD, FACT-ADDRESS and EXTERNAL-ADDRESS. Dacă avem instalat suportul pentru programarea pe obiecte sunt de asemenea valabile INSTANCE, INSTANCE-ADDRESS, INSTANCE-NAME, USER, INITIAL-OBJECT şi orice clasă definită de utilizator. Clasa specificată într-o restricţie de tip trebuie să fie în prealabil definită, pentru ca aceasta să poată fi utilizată în predetermina precedenţei dintre metodele unei funcţii generice. Clasele redundante nu sunt permise lista claselor folosite într-o restricţie. De exemplu tipul parametrului următoarei metode este redundant deoarece INTEGER este o subclasă a lui NUMBER.

```
(defmethod f ((?a INTEGER NUMBER)))
```

Dacă restricţia de tip (dacă există) este satisfăcută pentru un argument, atunci restricţia de interogare (dacă există) este aplicată. CLIPS-ul evaluează expresia şi dacă aceasta este diferită de simbolul FALSE, restricţia este considerată

a fi satisfăcută. Pot exista cazuri în care în cadrul unei interogări avem mai mulți parametri şi este posibil ca unul dintre aceştia să nu fi fost definit şi în acest caz datorită acestei erori metoda respectivă să nu fie aplicabilă. Ținându-se cont de faptul că evaluarea parametrilor are loc de la stânga la dreapta, interogările multiple trebuiesc incluse în parametrii cei mai din dreapta. In exemplul următor evaluarea restricției de interogare are loc cu o întârzierea datorată verificării claselor ambilor argumente.

(defmethod f ((?a INTEGER) (?b INTEGER (> ?a ?b))))

De reținut faptul că pentru a introduce o restricție de interogare nu este obligatorie introducerea unei restricții de tip. Pentru a demonstra aceasta vor fi definite cele două metodele mai jos. După cum se va putea observa ambele sunt corecte, fiind acceptate de CLIPS.

```
CLIPS> (defmethod f ((?a  NUMBER (> ?a 1) )))
CLIPS> (list-defmethods)
f  #1  (NUMBER  <gry>)
For a total of 1 method.
CLIPS> (f a)
[GENRCEXE1] No applicable methods for f.
CLIPS> (defmethod f ((?a (> ?a 1) )))
CLIPS> (list-defmethods)
f  #1  (NUMBER  <gry>)
f  #2  (<gry>)
For a total of 2 methods.
CLIPS> (f a)
[ARGACCES5] Function > expected argument #1 to be of type be integer or float.
CLIPS> (f 2)
FALSE
CLIPS> (f 1)
[GENRCEXE1] No applicable methods for f.
```

După cum se observă deşi cele două metode au acelaşi număr de parametri (unul singur), nu se realizează o operație de suprascriere la introducerea celei de-a doua metode, deoarece restricțiile asociate parametrilor sunt diferite şi CLIPS-ul vede cele două metode diferite. Dacă la început apelul (f a) semnala faptul ca nu a fost găsită o metodă care să fie aplicabilă (restricția de tip NUMBER neacceptând argumentul *a*, imediat după definirea celei de-a doua metode, apelul (f a) găseşte ultima metodă ca fiind aplicabilă (din lipsa restricției de tip), dar se obține eroare la verificarea restricției de interogare (funcția > acceptă doar numere ca argumente). Concluzia ce se obține este că atunci când în funcțiile din interiorul restricțiilor de interogare acceptă doar un anumite tipuri de argumente, este recomandată introducerea acelor tipuri într-o restricție de tip. Se observă că pentru apelul (f 1) este satisfăcută restricția de tip, dar nefiind satisfăcută nici restricția de interogare nici una din cele două metode nu este aplicabilă. Pentru apelul (f 2) se returnează în mod implicit simbolul FALSE, deoarece în interiorul celor două metode nu este introdusă nici o acțiune.

Foarte importantă în definirea unei metode este ordinea în care sunt introduse restricţiile de tip şi cele de interogare. Dată fiind ordinea de evaluare de la stânga la dreapta şi faptul că restricţiile de interogare depind de restricţiile de tip, devine evidentă aşezarea restricţiilor de tip înaintea restricţiilor de interogare. În cazul în care nu se respectă această ordine firească, cum este şi metoda definită în exemplu de mai jos, CLIPS-ul va semnala eroare:

```
CLIPS> (defmethod f ((?a  (> ?a 1)  NUMBER)))
[GENRCPSR10] Query must be last in parameter restriction.
```

Dacă restricţia de tip nu respectă tipurile de argumente acceptate de funcţiile din restricţiile de interogare, CLIPS-ul nu semnalează eroare şi va accepta o metodă astfel definită. La apelul funcţiei generice se va obţine bineînţeles o eroare în urma încercării de evaluare a restricţiei de interogare. Astfel de metode sunt inutile deoarece nu vor putea fi niciodată aplicate. Revine programatorului sarcina de a se asigura de acest aspect.

```
CLIPS> (defmethod f ((?a LEXEME  (> ?a 1) )))
CLIPS> (f a)                    ;a este un simbol deci apartine clasei LEXEME
[ARGACCES5] Function > expected argument #1 to be of type be integer or float.
```

Într-o restricţie de interogare pot fi folosite variabile globale sau apeluri de funcţie definite de utilizator. Nu există restricţia de a fi folosite doar funcţiile system. În acest fel restricţiile pot fi realizate cu propriile funcţii.

```
CLIPS> (deffunction is_valid (?a)
                  (if (and (numberp ?a) (>= ?a 0) (<= ?a 9))
                   then (return TRUE)) )
CLIPS> (deffunction negate (?a)
                  (return (- 0 ?a)) )
CLIPS> (defglobal ?*MAX* = 10)
CLIPS> (defmethod f ((?a  (is_valid ?a)) )   TRUE)
CLIPS> (f 10)
[GENRCEXE1] No applicable methods for f
CLIPS> (f  5)
TRUE
CLIPS> (defmethod g ((?a  (> (negate ?a) –1)) )  TRUE)
CLIPS> (g 2)
[GENRCEXE1] No applicable methods for f
CLIPS> (g -1)
TRUE
CLIPS> (defmethod h ((?a  (> ?a ?*MAX*)) )    TRUE)
CLIPS> (h 9)
[GENRCEXE1] No applicable methods for f
CLIPS> (h 11)
TRUE
```

S-au construit trei exemple distincte pentru a se evidenția cât mai bine modul în care pot fi folosite variabilele globale şi funcțiile în realizarea restricțiilor de tip. Pentru metoda f este apelată de către CLIPS funcția *is_valid*, ca şi o funcție de sistem. Pentru metoda g este apelată funcția *negate* din interiorul unei funcții de sistem, pentru a se arăta faptul că sunt acceptate în restricțiile de interogare şi funcțiile ce apelează alte funcții. Pentru ultima metodă h în locul unei valori explicite este folosită o variabilă globală.

Faptul că sunt acceptate şi variabilele globale în parametrii funcțiilor folosite de restricțiile de interogare, ne oferă posibilitatea prin modificarea valorilor acestor variabile globale de a controla execuția funcțiilor generice. Astfel având definită una sau mai multe variabile globale, în cadrul a două sau mai multe metode dintr-o funcție generică, prin schimbarea valorilor acestora se schimbă practic intervalele pentru care este aplicabilă fiecare metodă.

```
CLIPS> (defglobal ?*MAX* = 10)
CLIPS> (defmethod h ((?a  (> ?a ?*MAX*)) )    TRUE)
CLIPS> (h 9)
[GENRCEXE1] No applicable methods for f
CLIPS> (bind ?*MAX* 5)
CLIPS> (h 9)
TRUE
```

Folosirea variabilelor globale în acest mod poate oferi cheia rezolvării unor aplicații, care în mod normal nu ar fi putut fi rezolvate sau care ar fi cerut pentru a se crea această funcționare dinamică implementarea unor algoritmi speciali.

În mod surprinzător se poate observa în exemplul următor, cum în locul unei restricții de interogare poate fi folosită chiar o variabilă globală. În cazul în care aceasta are ca valoare un număr, un şir sau un simbol diferit de FALSE, aceasta va fi evaluată de CLIPS ca fiind întotdeauna adevărată. În exemplu următor nu se justifică folosirea variabilei globale ?*MAX* atât timp cât valoarea ei este un număr (metoda este întotdeauna adevarată).

```
CLIPS> (defmethod h ((?a  ?*MAX*))  TRUE)
CLIPS> (h 1)
TRUE
```

Prin simplul fapt că o variabilă globală poate lua orice valoare, inclusiv simbolul FALSE sau simbolul TRUE, s-a găsit răspunsul la întrebarea “Poate fi controlată execuția unei funcții generice?”. În felul acesta poate fi simulată de exemplu execuția unui ‘**semafor**’, dacă sunt îndeplinite condițiile atunci variabila globală ia valoarea TRUE şi funcția generică este aplicabilă, şi invers dacă nu sunt îndeplinite condițiile de aprindere atunci variabila globală ia valoarea FALSE şi funcția generică nu mai este aplicabilă. Se poate gândi această problemă şi folosind mai multe variabile globale, câte una pentru fiecare condiție sau set de condiții. În acest fel se pot eventual verifica doar condițiile care nu au fost încă îndeplinite.

Când toate variabilele globale vor fi evaluate ca fiind adevărate (diferite se simbolul FALSE), metoda va deveni aplicabilă.

```
CLIPS> (defglobal ?*BOOL* = FALSE)
CLIPS> (defmethod f ((?a  ?*BOOL*) )    (printout t  "Parametru: " ?a t))
CLIPS> (f  10)
[GENRCEXE1] No applicable methods for f
CLIPS> (bind  ?*BOOL*  TRUE)
CLIPS> (f 10)
Parametru: 10

CLIPS> (defmethod negate ((?a  SYMBOL))
                (if ?a then (return FALSE) else (return TRUE)) )
CLIPS> (defmethod semafor ((?a  ?*BOOL*) )    "S-a aprins")
CLIPS> (defmethod semafor ((?a (negate ?*BOOL*)) ) "Stins")
CLIPS> (bind ?*BOOL*)                          ;ia valoarea initiala
CLIPS> (semafor 1)
"Stins"
CLIPS> (bind ?*BOOL* TRUE)
CLIPS> (semafor 1)
"S-a aprins"
```

Pentru a acoperi şi situaţiile în care variabila globală este egală cu simbolul FALSE şi când nu se doreşte executarea vreunei acţiuni se poate realiza o metodă care este aplicabilă, atunci când nu este cealaltă aplicabilă. Se elimină în acest fel şi mesajele eroare care apar atunci când nu este găsită o funcţie aplicabilă. Se reaminteşte faptul că în urma execuţiei funcţiei (bind ?*BOOL*) fără a se specifica o valoare, variabila globală ia valoarea de care a fost legată în construcţia *defglobal*. În exemplul următor sunt suprascrise cele două metode de mai sus, fiecare ele va returna acum o valoare. Aceste valori pot fi folosite de exemplu pentru a opri execuţia unei bucle.

```
CLIPS> (defmethod semafor ((?a  ?*BOOL*) )
                        (printout t "S-a aprins! " t)  FALSE)
CLIPS> (defmethod semafor ((?a (negate ?*BOOL*)) )
                        (printout t "Stins..." t) TRUE)
CLIPS> (progn (bind ?time (time))
              (bind ?*BOOL*)
              (while (semafor 1) do
                     (if (> (time) (+ ?time 10))
                      then (bind  ?*BOOL*  TRUE)) )
              )
       )
```

În exemplul următor se stă într-o buclă timp de 10 secunde. Funcția *time* va returnează o valoare corespunzătoare timpului actual. După ce vor trece 10 secunde se schimbă valoarea variabilei globale ?*BOOL* şi odată cu ea şi metoda ce va fi aplicată. Această metodă va returna simbolul TRUE şi condiția de execuție a buclei *while* nu va mai fi îndeplinită.

O metodă este considerată a fi aplicabilă dacă toate argumentele respectă restricțiile de tip şi de interogare. Când mai mult de o metodă este aplicabilă unui set de argumente CLIPS-ul va determina relația de precedență în funcție de parametri restrictivi, pentru a selecta una din aceste metode.

Se poate realiza de exemplu o supraîncărcare a operatorului '+' de adunare, care să realizeze concatenarea a două şiruri, atunci când îi sunt transmise şiruri ca argumente. In acelaşi timp operatorul '+' va realiza şi adunarea aritmetică, atunci când îi sunt transmise ca argumente numere. In acest caz spunem ca avem două metode: una explicită definită de utilizator (**user-defined**) pentru cazul în care se realizează operația de concatenare de şiruri şi cealaltă implicită în care avem operatorul standard de adunare aritmetică (**system function**).

Pentru a câştiga ceva timp se poate amplasa funcția generică ce realizează operația de supraîncărcare exact înaintea regulii care o foloseşte, şi în acest fel celelalte reguli vor folosi implicit funcția sistem până la momentul realizării operației de supraîncărcare.

```
CLIPS> (defmethod + ((?a STRING) (?b STRING))
                          (str-cat ?a ?b) )
CLIPS> (+ 1 2)
3
CLIPS> (+ "mot" "oras")
"motoras"
CLIPS> (+ "ca" "ram" "ida")              ;sunt acceptate doar 2 argumente
[GENRCEXE1] No applicable methods for +.
FALSE
```

O metodă poate accepta *exact* sau *cel puțin* un număr specificat de argumente, depinzând de folosirea sau nu a parametrului multiplu. Parametrii obişnuiți specifică numărul minim de argumente care trebuie transmişi metodei. Fiecare din aceşti parametri poate fi referit ca o variabilă simplă (singur-câmp). Dacă parametrul multiplu este prezent, atunci trebuiesc transmişi metodei un număr de argumente mai mare sau egal cu minimul. Dacă parametrul multiplu nu este prezent, atunci trebuiesc transmişi metodei un număr exact de argumente specificat de parametrii obişnuiți. Argumentele metodei care nu corespund parametrilor obişnuiți pot fi grupate într-o valoare multicâmp, care poate fi referită în corpul metodei. Funcțiile standard multicâmp ale CLIPS-ului, cum ar fi length$ şi expand$, pot fi aplicate parametrilor multipli.

```
CLIPS> (defmethod + (($?s STRING))
                          (str-cat (expand$ $?s)) )

CLIPS> (+ "ca" "ram" "ida")
"caramida"
```

```
CLIPS> (+ a 1 b “v”)
[GENRCEXE1] No applicable methods for +.
CLIPS> (defmethod + ($?s)
                        (str-cat (expand$ $?s)) )
CLIPS> (+ 1 2 3)
6
CLIPS> (+ 1 2 a)
“12a”
```

A fost reluat cazul în care se realizează supraîncărcarea operatorului “+” , de această dată metoda nou creată foloseşte un parametru multiplu şi vor putea fi concatenate mai multe de două şiruri de caractere. Imediat după aceasta s-a dorit realizarea unei variante care să accepte şi argumente de tip SYMBOL sau NUMBER şi pentru aceasta a fost eliminată restricția de tip. Exista însă riscul în acest moment ca atunci când toți parametri erau numere să fie efectuată operația de concatenare şi nu cea de adunare cum ar fi fost normal. Dacă prin suprascrierea operatorului “+” s-ar fi pierdut tocmai operația de adunare, atunci ar fi trebuit rescrisă ultima metodă, punând o condiție de genul “dacă unul din parametri nu este număr, atunci execută acțiunile funcției generice”.

Un parametru multiplu se comportă în mod similar ca şi parametrii obişnuiți. Unica diferență este aceea că atunci când vrem să realizăm o restricție de interogare pentru un parametru multiplu (care trebuie să verifice fiecare din valorile acestuia), această restricție nu poate fi asociată în mod direct variabilei multiple. Deşi la definirea următoarei metode nu este semnalată o eroare, dar în momentul în care această metodă va fi aplicată, deoarece funcția > nu acceptă argumente de tip multicâmp se obține eroare la evaluarea restricției de interogare.

```
CLIPS> (defmethod +  (($?any INTEGER (> $?any 0)) ))
CLIPS> (+ 1 2)
[ARGACCES5] Function > expected argument #1 to be of type be integer or float.
3
```

Se poate observa cum pentru exemplul de mai sus, deşi s-a obținut eroare la execuția metodei definite, a fost executată funcția de adunare. Pentru a lămuri acest se vor urmări cu ajutorul comenzii **(watch methods)**, toate metodele aplicabile. Va fi folosită de asemenea şi funcția **preview generic** pentru a vizualiza toate funcțiile generice, care sunt aplicabile pentru anumite argumente, fără a executa vreo funcție. Trebuie specificat mai întâi numele funcției generice şi apoi argumentele ce sunt folosite la apelul acesteia.

```
CLIPS> (defmethod + ((?a INTEGER) (?b INTEGER))
                        (* (- ?a ?b) (- ?b ?a)))
CLIPS> (list-defmethods +)
+  #3   (INTEGER) (INTEGER)
+  #2   ($? INTEGER <gry>)
+  #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
For a total of 3 methods.
CLIPS> (preview-generic + 1 2)
```

```
+  #2  (INTEGER) (INTEGER)
[ARGACCES5] Function > expected argument #1 to be of type be integer or float.
+  #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
CLIPS> (watch methods)
CLIPS> (+ 1 2)
MTH >> +:#2  ED:1 (1 2)
MTH << +:#2  ED:1 (1 2)
-1                              ;a fost executată functia generică
CLIPS> (unwatch methods)
CLIPS> (preview-generic + 1 2 3)
[ARGACCES5] Function > expected argument #1 to be of type be integer or float.
+  #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
CLIPS> (preview-generic + 1.5  3)
+  #SYS1  (NUMBER) (NUMBER) ($? NUMBER)
```

Se observă cum pentru un apel (+ 1 2) sunt aplicabile trei metode, pentru un apel (+ 1 2 3) două reguli, iar pentru un apel (+ 1.5 3) este aplicabilă doar funcția sistem. De observat cum o metodă + care are restricția de tip INTEGER sau FLOAT este aplicată şi nu funcția sistem de adunare al cărei restricție de tip este mai generală NUMBER. Foarte interesant de observat este şi faptul că pentru un apel de genul (+ 3 1.5), comanda *preview-generic* arată că este aplicabilă şi metoda în care a fost introdusă în mod incorect restricția de interogare.

Expresiile formate din parametrii multipli pot fi utilizate în interogări, dar pentru a ne referi la argumente individuale dintr-un parametru multiplu se utilizează variabile speciale în restricțiile de interogare: **?curent-argument**. Această variabilă este folosită doar în interogare şi nu are nici un corespondent în corpul metodei. De exemplu pentru a crea o variantă a operatorului + pentru cazul în care toate argumentele sunt numere întregi pare, care returnează suma acestora înjumătățită :

```
CLIPS> (defmethod +  (($?any INTEGER (evenp ?current-argument)))
                     (div (call-next-method) 2))
CLIPS> (+ 1 2)
3
CLIPS> (+ 4 6 4)
7
```

Cu ajutorul funcției *call-next-method* este apelată cealaltă metoda aplicabilă (în cazul nostru funcția sistem de adunare). Cu ajutorul acesteia sunt adunate argumentele,iar apoi valoarea returnată de aceasta este împărțită la 2.

Este importantă folosirea restricțiilor de tip şi de interogare asupra parametrilor multipli deoarece acestea sunt aplicate fiecărui argument grupat în multicâmp. In exemplul următor functțile > şi **length$** sunt apelate de trei ori, pentru fiecare argument în parte.

```
CLIPS> (defmethod f  (($?any  (> (length$ ?any) 2)))  yes)
CLIPS> (f  1 red 3)
yes
```

În plus o restricție de interogare nu va fi examinată niciodată dacă nu avem nici un argument în parametrul multiplu. De exemplu, metoda anterioară este aplicabilă şi poate fi apelată de o funcție generică care nu are argumente deoarece restricția de interogare nu este evaluată niciodată în acest caz.

```
CLIPS> ( f )
yes
```

În mod frecvent o restricție aplicată des parametrilor multipli este cea legată de cardinalitate (numărul argumentelor transmişi metodei). În anumite cazuri când tipul nu este important pentru a fi testat, se poate ataşa restricția de interogare parametrilor obişnuiți pentru a creşte performanța. În acest sens metoda anterioară poate fi implementată astfel:

```
CLIPS> (clear)
CLIPS> (defmethod f ((?arg (> (length$ ?any) 1)) $?any) yes)
CLIPS> (f)
[GENRCEXE1] No applicable methods for f.
FALSE
CLIPS> (f  1 2)                ; parametru any are doar valoarea 2
[GENRCEXE1] No applicable methods for f.
CLIPS> (f  1 2 3)              ;parametru any are valorile {2,3}
yes
```

Această aproximare nu ar trebui utilizată dacă tipul argumentelor grupate în multicâmp trebuie verificat înainte pentru a putea efectua în siguranță restricția de tip.

Nu pot fi supraîncărcate toate funcțiile sistem existente. În tabelul de mai jos au fost enumerate acele funcții de sistem ce nu pot fi supraîncărcate, deoarece CLIPS-ul va genera erori.

activeduplicateinstance	delayeddoforallinstances	messagemodifyinstance
activeinitializeinstance	doforallinstances	**modify**
activemakeinstance	doforinstance	modifyinstance
activemessageduplicateinstance	**duplicate**	nexthandlerp
activemessagemodifyinstance	duplicateinstance	nextmethodp
activemodifyinstance	**expand$**	objectpatternmatchdelay
anyinstancep	findallinstances	overridenexthandler
assert	findinstance	overridenextmethod
bind	**if**	**progn**
break	makeinstance	**progn$**
callnexthandler	initializeinstance	**return**
callnextmethod	**loop-for-count**	**switch**
callspecificmethod	messageduplicateinstance	**while**

4.2. Avantajele oferite de supraîncărcarea funcțiilor

În unele cazuri, când utilizatorul introduce date greşit de la tastatură, o funcție care are ca parametri aceste valori greşit introduse va furniza erori. Spre exemplu funcțiile >, < , <=, >=, <> sunt folosite numai pentru a compara numere. Dacă se încearcă compararea a două şiruri, execuția programului este brusc întreruptă semnalându-se această eroare. De exemplu:

```
(deftemplate person
        (slot name)
        (slot ocupation)
        (slot age))

(deffacts data_base
        (person (name Maria) (age 62) (ocupation teacher))
        (person (name Somebody) (age none) (ocupation actor)) )

(defrule old_person
        (person (name ?n) (age ?a) )
        (test (> ?a 60))
        =>
        (printout t  ?n " is a old person. " crlf) )
```

```
CLIPS> (reset)
[ARGACCES5]  Function > expected argument #1 to be of type integer or float

[DRIVE1]  This error occured in the join network
        Problem resides in join #1 in rule(s):
                old_person
```

Utilizatorul poate modifica construcția deftemplate, adăugând o serie de restricții dar rezultatul va fi acelaşi când se va aserta un fapt greşit.

```
(deftemplate person
        (slot name (type SYMBOL))
        (slot ocupation (type LEXEME))
        (slot age (type INTEGER) (range 1 ?VARIABLE)) )

CLIPS> (assert (person (name DAN) (age zece)))

[CSTRNCHK1] A literal slot value found in the assert comand
does not match the allowed typr of age
```

Adăugarea acestor restricții este recomandată deoarece elimină posibilitatea existenței în baza de fapte a unor date eronate. Dacă avem această posibilitate de a adăuga restricții în construcțiile deftemplate, pentru faptele obişnuite ea nu există. Se pot verifica manual datele introduse într-o buclă de tip **while** până acestea sunt corect introduse. Putem obține însă şi aici erori de programare.

```
(defrule get-number
        =>
        (printout t "Introduceti un număr : " )
        (bind ?n (read))
        (while (or (< ?n 1) (not (integerp ?n)) )
                (printout t "Introduceti un număr : " )
                (bind ?n (read)) )
        (assert (number ?n))
)
CLIPS> (reset)
CLIPS> (run)
Introduce-ti un numar : 5
```

[ARGACCES5] Function < expected argument #1 to be of type integer or float
[PRCODE4] Execution halted during the action od defrule det-number

Eroarea obținută se datorează faptului ca ordinea de evaluare a condițiilor din bucla while este de la stânga la dreapta. Expresia (or (< ?n 1) (not (integerp ?n))) poate fi înlocuită cu **(or (not (integerp ?n)) (< ?n 1))** şi în acest fel primul test va fi realizat prin funcția predicativă *integerp* şi dacă variabila ?n nu este un număr întreg, atunci evaluarea expresiei este întreruptă, fără a mai fi apelată şi funcția >. Astfel de erori de programare pot fi de cele mai multe ori evitate.

Se poate întâmpla ca într-un program să avem fapte cu acelaşi nume, dar a căror semnificație să fie difertă. Astfel faptul **(limita infinit)** poate semnifica faptul că o funcție nu are limită, pe când (limita 3) faptul că limita funcției este finită.

```
(defrule limite-diferite
        (limita ?l1)
        (limita ?l2 &: (<> ?l1 ?l2))
        =>
        (printout  t  "Limita la dreapta este diferita de limita la stanga." t)
        (printout  t  "Funcția este discontinua." crlf)
)
CLIPS> (assert (limita infinit)
                (limita 3))
```

În acest caz eroarea care este semnaltă se datorează funcției <>, care acceptă doar argumente de tip integer sau float. Se poate folosi în loc funcția **neq** a cărei argumente pot fi de orice tip (număr, şir, simbol). În acest caz **nu** se poate rezolva problema adăugând teste suplimentare care să verifice dacă variabilele ?l1 şi ?l2 sunt numere (funcția predictivă **numberp**). Un test în LHS-ul regulii de genul în care este folosită tot funcția <> :

(test (and (numberp ?l1) (numberp ?l2) (<> ?l1 ?l2)))

ar conduce la aceiaşi eroare. Unica soluție rămâne folosirea funcției *neq*:

```
(defrule limite-diferite
        (limita ?l1)
        (limita ?l2 &:(neq ?l1 ?l2))
        =>
        (printout  t  "Limita la dreapta este diferita de limita la stanga." t)
        (printout  t  "Funcția este discontinua." crlf)
)
```

O metodă mai elegantă de a rezolva aceste probleme este supraîncărcarea opertorilor <, > etc. In acest fel forțăm funcțiile să accepte şi argumente de alte tipuri decât cele pentru care au fost definite. De exemplu:

```
(defgeneric >)              ;declarația headerului este opțională
(defmethod > ((?a LEXEME) (?b LEXEME))
        (> (str-compare ?a ?b) 0)
)
```

Funcția > acceptă însă mai mult de două argumente şi pentru aceasta putem adăuga o nouă metoda, care să accepte un număr variabil de argumente.

```
(defmethod > ((?a LEXEME) ($?arg LEXEME))
        (loop-for-count (?i 1 (length$ $?arg))
                (if (<= (str-compare ?a (nth$ ?i $?arg)) 0)
                 then (return FALSE) )
        )
        TRUE
)
CLIPS> (> c b a)
TRUE
CLIPS> (> c b c)
FALSE
```

Se poate modifica această metodă în aşa fel încât să accepte şi un număr de zero argumente.

```
(defmethod > (($?arg LEXEME))
        (bind ?a (nth$ 1 $?arg))
        (loop-for-count (?i 2 (length$ $?arg))
                (if (<= (str-compare ?a (nth$ ?i $?arg)) 0)
                 then (return FALSE) )
        )
        TRUE
)
CLIPS> (>)
TRUE
CLIPS> (> 1 a)
[GENRCEXE1]    No applicable methods for >
```

Pentru selectarea primului element este folosită funcţia **(nth$ 1 $?arg)** şi nu **(first$ $?arg)** , deoarece cea din urmă returnează o variabilă multiplă şi nu o variabilă simplă.

Până în acest moment sunt acceptate pentru funcţia generică > argumente de tip integer şi de tip lexeme, dar nu şi de ambele tipuri. Dacă dorim acest lucru putem elimina restricţia de tip şi argumentele vor putea fi de orice tip. Trebuie ţinut cont şi de faptul că funcţia *str-compare* nu acceptă decât argumente de tip şir sau simbol, şi că în acest caz toate argumentele trebuiesc convertite în şiruri.

```
(defmethod > ($?arg )                    ; argumente de orice tip
        (bind ?a (implode$ (first$ $?arg)))
        (loop-for-count (?i 2 (length$ $?arg))
                (if (<= (str-compare ?a (implode$ (subseq$ $?arg ?i ?i))) 0)
                  then (return FALSE) )
        )
        TRUE
)
CLIPS> (> 1 a)
FALSE
CLIPS> (>a 1)
TRUE
CLIPS> (> c 1 'a' "b")
TRUE
```

4.3. Întrebări

1. Care sunt diferenţele dintre funcţiile generice definite cu ajutorul construcţiilor *defgeneric* şi *defmethod* şi funcţiile definite de utilizator cu ajutorul construcţiei *deffunction*?
2. Care este diferenţa între o metodă şi o funcţie generică (multiplă)?
3. Când este o funcţie generică supraîncărcată ? În condiţiile în care o funcţie este supraîncărcată, care sunt criteriile ce vor stabili ce metodă va fi folosită?
4. Pentru a declara o funcţie generică folosirea headerului este obligatorie? Trebuie ca acesta să fie declarat în mod explicit?
5. Odată declarată o metodă cu ajutorul construcţiei *defmehod*, cum se identifică funcţia generică din care aceasta face parte?
6. Pentru o metodă trebuie întotdeauna folosit un număr fix de argumente? Cum se stabileşte dacă un parametru este restrictiv sau nu?
7. La definirea unei metode utilizarea unui parametru multiplu este obligatorie? Pot fi folosiţi mai mulţi parametri multipli? Dacă da, care este numărul maxim de astfel de parametri ?
8. Un parametru multiplu poate fi şi el considerat parametru restrictiv?

9. O metodă a unei funcții generice poate fi apelată de o metodă altei funcții generice? Dar de o metodă ce aparține aceleiaşi funcții generice?
10. Care este condiția ca o metodă să poată fi apelată de alt obiect, regulă, de o altă metodă a unei funcții generice sau lansată în execuție la top-level? Cum se procedează în cazul funcțiilor generice recursive?
11. Poate fi o funcție generică recursivă, sau numai funcțiile definite cu *deffunction* pot fi recursive?
12. Cum se stabileşte faptul că două sau mai multe metode aparțin aceleiaşi funcții generice? În ce condiții poate fi înlocuită o metodă a unei funcții generice cu o alta metodă recent definită?
13. Cum este indexul asignat în mod automat de către CLIPS fiecărei metode. Care este funcția cu ajutorul căreia poate fi acesta vizualizat?
14. Există posibilitatea ca în interiorul unei funcții generice recursive să se apeleze metode diferite?
15. Ce fel de restricții se pot asocia la definire argumentelor unei metode? Este afectată în mod direct viteza de găsire a unei metode de complexitatea parametrilor restrictivi?
16. Este obligatorie asocierea de restricții pentru fiecare parametru? Sunt permise clasele redundante în lista claselor folosite într-o restricție de tip?

```
(defmethod f ((?a  SYMBOL  LEXEME)))
```

17. Pentru a introduce o restricție de interogare este obligatorie introducerea unei restricții de tip? Care din metodele definite mai jos sunt corecte?

```
(defmethod f ((?a  NUMBER (> ?a 1) )))
(defmethod f ((?a (> ?a 1) )))
```

18. Pentru metodele de mai sus, deoarece au acelaşi număr de parametri va fi realizată o suprascriere?
19. Dacă se doreşte intyroducerea pentru un parametru a unei restricții de tip şi a unei restricții de interogare este obligatorie introducerea acestora într-o anumită ordine? Sunt ambele metode definite mai jos corecte?

```
(defmethod f ((?a  NUMBER  (> ?a 1) )))
(defmethod f ((?a  (> ?a 1) NUMBER)))
```

20. Dacă restricția de tip nu respectă tipurile de argumente acceptate de funcțiile din restricțiile de interogare se semnalează eroare? Va accepta CLIPS-ul definirea unei metode cum este cea de mai jos?

```
(defmethod f ((?a  STRING  (> ?a 1) )))
```

21. Pot fi folosite într-o restricție de interogare variabile globale sau apelurile de funcții definite de utilizator? Există restricția de a fi folosite doar funcțiile de

sistem? Dacă avem definite funcţiile *is_valid*, *negate* şi variabila globală *?*MAX** sunt corecte următoarele metode?

```
(defmethod f ((?a  (is_valid ?a)) ))
(defmethod g ((?a  (> (negate ?a) –1)) ))
(defmethod h ((?a  (> ?a ?*MAX*)) ))
```

22. Când mai mult de o metodă este aplicabilă unui set de argumente vor fi aplicate toate aceste metodele la apelul funcţiei generice?
23. Poate fi controlată execuţia funcţiilor generice prin folosirea variabilelor globale? Dacă da, exemplificaţi.
24. Funcţiile de sistem pot fi supraîncărcate, sau pentru aceasta pot fi folosite doar funcţiile definite de utilizator?
25. Cum se poate calcula numărul de parametri pe care îi poate accepta o funcţie generică? Acest număr este fix?
26. Pot fi folosiţi mai mulţi parametri multipli? Se pot asocia restricţii de tip sau de interogare parametrilor multipli? Aceşti modifică în vre-un fel numărul argumentelor pe care le poate accepta funcţia generică?
27. Cum se aplică restricţiile de interogare unui parametru multiplu? Ce va genera următoarea funcţie generică pentru apelul

```
(defmethod +  (($?any INTEGER (> $?any 0)) ))
```

28. Ce valoare are variabila ?current-argument după apelarea următoarei metode?

```
(defmethod +  (($?any INTEGER (evenp ?current-argument)))
        (printout  t  ?current-argument))
```

29. Dacă nu avem nici un argument în parametrul multiplu este examinată o restricţie de interogare? Ce va returna apelul unei funcţii generice următoare dacă nu specifică valori *(f)* pentru parametru multiplu?

```
(defmethod f  (($?any  (> (length$ ?any) 2)))  yes)
```

30. Folosirea unei restricţii de interogare, ce foloseşte un argument la care nu s-a ajuns încă in lista de parametri duce la apariţia unei erori?

```
(defmethod f  ((?arg  (> (length$ ?any) 1))  $?any)  yes)
```

4.4. Probleme rezolvate

1. Supraîncărcarea operatorului de împărţire – Să presupunem că după efectuarea operaţiei de împărţire între două numere dorim să vizualizăm un număr mai mare de zecimale. În mod normal CLIPS-ul pentru o operaţie de genul:

```
CLIPS> ( /  3  7 )
0.4285714285714285          ;17 cifre – 16 zecimale
CLIPS> ( /  112234  7)
16033.42857142857           ;16 cifre – 11 zecimale
```

oferă un rezultat ce conține 16 cifre (în cazul în care partea întreagă este nulă, atunci cifra 0 ce apare la început nu este luată în considerație). Ceea ce se poate observa foarte uşor mai sus este faptul că numărul maxim de zecimale ce poate fi vizualizat este de 16.

Pentru simplitatea problemei se va exemplifica doar cazul în care cei doi operanzi sunt numere întregi pozitive. Dacă se doreşte efectuarea de calcule cu numere negative, atunci mai întâi se determină semnul rezultatului şi dacă acesta este negativ, atunci se va pune în fata caracterul minus. Dacă se doreşte efectuarea de operații cu numere flotante, atunci se calculează operația presupunând cei doi operanzi ca fiind întregi şi în funcție de poziția virgulei din fiecare se va deplasa virgulei în rezultatul obținut. Pseudocodul funcției este următorul:

```
Funcția împărțire
calculează şi afişează câtul împărțirii (şi punctul)
(se calculează partea fracționară)
 deîmpărțitul este egal cu restul împărțirii, înmulțit cu zece
 repetă cât timp nu s-au calculat numărul de zecimale specificate
        dacă deîmpărțitul este zero
        atunci opreşte bucla
        afişează o nouă zecimală, câtul împărțirii
        deîmpărțitul este egal cu restul împărțirii, înmulțit cu zece
 sfârşit buclă
sfârşit funcție
```

După calcularea părții întregi, egală practic cu câtul împărțirii, faptul că noul deîmpărțit este egal de fiecare dată nu restul împărțirii înmulțit cu zece, ne asigură automat faptul că la următoarea operație de împărțire câtul va fi de maxim o cifră (între 0 şi 9). În acest fel se calculează rând pe rând fiecare cifră zecimală. Algoritmul de mai sus realizează doar afişarea rezultatului, dar în cazul în care se doreşte şi returnarea rezultatului, atunci dat fiind faptul că în CLIPS suntem limitați la un anumit interval pentru numerele întregi sau numerele flotante (după depăşirea acestuia are loc o eroare de "depăşire") problema se complică.

Ideea foarte simplă prin care se poate rezolva totuşi şi această problemă constă în stocarea rezultatului într-un şir de caractere. În acest fel rezultatul furnizat la ieşire poate avea un număr foarte mare de zecimale (chiar **10000**).

```
(defmethod /    ((?a INTEGER (> ?a 0)) (?b INTEGER (> ?b 0))
                 (?nr_cifre INTEGER (> ?nr_cifre 0)))
        (bind ?rez (str-cat   ""   (div ?a ?b) "."))
        (bind ?a (* (mod ?a ?b) 10))
        (loop-for-count (?i 1 ?nr_cifre)
                (if (= ?a 0) then (break))
                (bind ?rez (str-cat ?rez (div ?a ?b) ))
                (bind ?a (* (mod ?a ?b) 10)) )
        (return ?rez)
)
```

Din acest moment putem practic efectua operaţii de împărţire direct de la prompterul CLIPS-ului, dar în cazul în care cei trei operanzi (deîmpărţitul, împărţitorul şi numărul de zecimale) nu sunt numere întregi pozitive, CLIPS-ul va folosi funcţia sistem pentru a efectua împărţirea.

```
CLIPS> ( /  2.3  7  30)
0.03285714285714286
CLIPS> (/  -2  3  30)                ;se împarte -2 la 3 şi apoi la 30 <=> -2 / 90
-0.0222222222222222
CLIPS> ( /  3  7  30)
“0.428571428571428571428571428571”          ;se returnează un şir
```

Pentru a forţa introducerea corectă a operanzilor se poate realiza eventual o regulă, unde pentru fiecare operand se stă într-o buclă cât timp nu se introduce la intrare un număr întreg şi pozitiv.

```
(defrule introducere_date
        =>
        (printout t " Functia (impartire ?a ?b ?c) " t)
        (printout t ", unde ?a este deimpartitul" t)
        (printout t "       ?b este impartitorul" t)
        (printout t "    si ?c numarul de zecimale (1 -10000) "t)
        (printout t "a: ")
        (while (and (not (integerp (bind ?a (read)))) (> ?a 0))
                (printout t "a (intreg pozitiv) :") )
        (printout t "b: ")
        (while (and (not (integerp (bind ?b (read)))) (> ?b 0))
                (printout t "b (intreg pozitiv) :") )
        (printout t "c: ")
        (while (and (not (integerp (bind ?c (read)))) (> ?c 0))
                (printout t "c (intreg pozitiv) :") )
        (printout t  (/ ?a ?b ?c))
)
```

Pentru cazul în care s-ar dori implementarea unei metode care să accepte ca parametri şi numere flotante, eventual chiar şi numere negative putem determina semnul operaţiei şi eventual deplasamentul virgulei pentru cazul în care am presupune că cele două numere sunt întregi (se elimină punctul). Pentru a obţine o funcţie generică mai mică, se vor defini în afara metodei funcţiile *sign, del* şi *ins*.

Funcţia *sign* returnează un şir vid dacă ambele argumente sunt pozitive sau negative, sau caracterul ‘-‘, dacă unul dintre argumente este negativ.

```
(deffunction sign (?a ?b)
        (if (or (and (neq ?a (abs ?a)) (eq ?b (abs ?b)) )
                (and (eq ?a (abs ?a)) (neq ?b (abs ?b)) ) )
         then (return "-")
         else (return "") )
)
```

Prin expresia din condiția instrucțiunii IF se realizează practic o restricție de tip XOR (sau exclusiv), a cărei valoare de adevăr este TRUE, numai în cazul în care cei doi parametri au valorile de adevăr diferite.

$$a \oplus b = \bar{a}b + a\bar{b}$$

Funcția următoare *del* şterge un caracter dintr-un şir de pe o poziție specificată şi returnează şirul astfel format. Dacă pentru argumentul *?char* se introduce un şir de caractere, ştergerea acestuia este realizată numai în cazul în care acesta este un subşir al lui *?string*.

```
(deffunction del (?char ?string)
        (if (bind ?poz (str-index ?char ?string))
         then   (bind ?first (sub-string 1 (- ?poz 1) ?string))
                (bind ?last (sub-string (+ ?poz 1) (length ?string) ?string)) )
        (return (str-cat ?first ?last))
)
```

În adiție pentru funcția anterioară este realizată funcția ins, care introduce un caracter într-un şir într-o poziție specificată. De asemenea dacă se introduce ca argument pentru *?char* un şir de caractere, acesta va fi introdus în *?string* începând cu poziția specificată de *?poz*.

```
(deffunction ins (?poz ?char ?string)
        (bind ?first (sub-string 1 (- ?poz 1) ?string))
        (bind ?last (sub-string ?poz (length ?string) ?string))
        (return (str-cat ?first ?char ?last))
)
```

Se pot face verificări asupra corectitudinii funcțiilor nou create:

```
CLIPS> (sign -3 4)
TRUE
CLIPS> (sign -3 -7)
FALSE
CLIPS> (del  "."   "23.45")
2345
CLIPS> (ins  3  ".0"   "234")
23.04
```

În acest moment vom realiza o metodă care acceptă ca argumente şi numere flotante şi numere negative. După ce argumentele vor fi transformate în numere întregi pozitive, se va apela recursiv funcția de împărțire. De această dată, va fi aplicată prima metodă, care avea ca restricție de tip pentru variabilele *?a* şi *?b* tipul INTEGER. “Pentru a transforma un şir din care a fost eliminat caracterul “.” (ce separa partea întreagă de partea fracționară dintr-un număr flotant), într-un număr întreg se foloseşte funcția **eval**. Această funcție, după cum se ştie este folosită de obicei pentru evaluarea unor expresii (de exemplu *(eval “(+ 2 3)”)* ar furniza ca rezultat 5). Dacă şirul de caractere primit ca argument este un număr această funcție returnează în urma evaluării expresiei numărul respectiv. Funcția generică poate fi scrisă şi într-o formă mai compactă, fără a se mai apela din

interiorul acesteia alte funcţii definite de utilizator, dar în acest caz se pierde din claritatea codului şi se măreşte foarte mult dimensiunea corpului funcţiei.

```
(defmethod  /   ((?a NUMBER) (?b NUMBER) (?nr_cifre INTEGER (> ?nr_cifre 0)))
        (bind ?rez (sign ?a ?b))    (bind ?a (abs ?a))    (bind ?b (abs ?b))
        (if (neq ?a (integer ?a))                                   ;daca este flotant
         then    (bind ?a (str-cat ?a))
                 (bind ?na (- (length ?a) (str-index "." ?a)))
                 (bind ?a (eval (del "." ?a)))
         else (bind ?na 0) )
        (if (neq ?b (integer ?b))                                   ;daca este flotant
         then    (bind ?b (str-cat ?b))
                 (bind ?nb (- (length ?b) (str-index "." ?b)))
                 (bind ?b (eval (del "." ?b)))
         else (bind ?nb 0) )
        (bind ?rez  (str-cat ?rez  (/  ?a  ?b ?nr_cifre)))          ;apel recursiv
        (bind ?nc (str-index "." ?rez))
        (return (ins (+ (- ?nb ?na) ?nc) "."  (del "." ?rez)))
)
```

Variabilele *?na* şi *?nb* păstrează numărul de zecimale pe care îl conţine variabila *?a* şi respectiv *?b*. Pentru a determina poziţia pe care o are virgula în şirul final se păstrează şi poziţia virgulei în şirul obţinut în urma apelului recursiv în variabila *?nc*. Diferenţa *(- ?nb ?na)* oferă practic valoarea cu care trebuie deplasată virgula la stânga sau la dreapta. Se mai observă faptul că după stabilirea semnului operaţiei, ambii operanzi devin pozitivi, cu ajutorul funcţiei *abs*.

Pentru obţinerea numerelor întregi (eliminarea virgulei), au fost folosite numai operaţii cu şiruri de caractere. Se poate implementa şi o variantă în care se determină poziţia virgulei fără a se folosi astfel de funcţii:

```
(deffunction virg (?n)
        (bind ?poz 0)
        (while (neq (- (integer ?n) ?n)    0   0.0)
                (bind ?poz (+ ?poz 1)) (bind ?n (* ?n 10)) )
        (return ?poz)
)
```

Funcţia **virg** definită mai sus, verifică practic într-o buclă dacă diferenţa între numărul primit ca argument înmulţit cu zece şi partea sa întreagă este 0 sau 0.0. De reţinut faptul că în CLIPS numărul întreg 0 nu este identic cu numărul flotant 0.0 (se poate verifica la prompter, (eq 0 0.0) returnează simbolul FALSE) şi pentru a nu ajunge se obţine o buclă infinită sau la alte erori trebuiesc verificate ambele valori. Se arată şi cum rulează bucla WHILE.

```
CLIPS> (virg  2.345)          |   2.345  –   2   = 0.0     FALSE
3                             |   23. 45 –  23   = 0.0     FALSE
CLIPS> (virg 3)               |   234.5  –  234  = 0.0     FALSE
0                             |   2345.0 – 2345 =  0.0     TRUE
```

Fără a mai insista asupra modificărilor ce trebuiesc efectuate asupra funcției generice ce apelează funcția *virg* definită mai sus, va mai fi prezentată o variantă mult simplificată a metodei ce accepta ca argument numere flotante.

De multe ori codul unei funcții poate fi restrâns, folosindu-ne de facilitățile folosite de CLIPS pentru anumite funcții. În cazul de față, cunoscând faptul că funcțiile **div** şi **mod** care calculează câtul şi respectiv restul unei operații de împărțire, acceptă nu numai argumente de tip întreg, ci şi numere flotante se pot elimina toate funcțiile definite mai sus, care determinau poziția virgulei sau semnul operației. Metoda următoare nu diferă de prima metodă (ce accepta ca argumente doar numere întregi pozitive) decât prin două funcții *abs,* introduse în momentul în care se trece la calculul părții fracționare. Practic la primul apel *div*, se stochează semnul operației (dacă este necesar), şi odată problema semnului fiind rezolvată, funcția *mod* va primi ca argumente numere pozitive (deoarece dacă unul dintre argumente este negativ şi rezultatul poate fi negativ *(abs –6 4)* returnează -2).

```
(defmethod   /  ((?a NUMBER) (?b NUMBER) (?nr_cifre INTEGER (> ?nr_cifre 0)))
        (bind ?rez "")
        (bind ?rez (str-cat ?rez (div ?a ?b) "."))
        (bind ?a (* (mod (abs ?a) (abs ?b)) 10))
        (loop-for-count (?i 1 ?nr_cifre)
                (if (= ?a 0) then (break))
                (bind ?rez (str-cat ?rez (div ?a ?b) ))
                (bind ?a (* (mod ?a ?b) 10)) )
        (return ?rez)
)
```

Pentru a realiza o afişare deosebită a rezultatului se poate implementa o funcție care să calculeze şi perioada unui număr flotant stocat într-un şir. Astfel pentru operația *(/ 3 7 30)* se obține 0,428571428571428571428571428571 şi această valoare poate fi scrisă mai simplu 0,(428571) ținând cont de faptul că numerele se repetă. Prin determinarea perioadei pot fi astfel scrise rezultatele operației de împărțire într-o formă mai simplă.

```
(deffunction period (?str)
        (bind ?len (length ?str))
        (loop-for-count (?i 1 (- ?len 1))
                (loop-for-count (?j (+ ?i 1) ?len)
                        (if (eq (sub-string ?i ?i ?str) (sub-string ?j ?j ?str))
                         then    (if  (verif ?i ?j ?str)
                                  then (return (str-cat (sub-string 1 (- ?i 1) ?str)  "("
                                                        (sub-string ?i (- ?j 1) ?str)   ")"  ))
        )       )       )       )
        (return ?str)
)
```

Funcția *period* definită mai sus, caută pornind de la primul caracter al şirului *?str*, pozițiile în care acesta apare din nou. În condițiile în care se găseşte o

astfel de poziție, este apelată funcția *verif*, care returnează simbolul TRUE dacă subşirul dintre cele două poziții se repetă în mod periodic până la sfârşitul şirului. Dacă se găseşte un astfel de subşir atunci execuția funcției este oprită folosindu-se comanda *return*, mult mai avantajoasă în acest caz decât folosirea unor comenzi *break*, necesare ieşirii din cele două bucle for imbricate. Şirul de caractere returnat păstrează identic şirul sursă până la poziția în care a fost găsită perioada, şi încadrează între două paranteze deschise subşirul ce se repetă.

```
(deffunction verif  (?i ?j ?str)
        (bind ?k ?j)
        (bind ?len (length ?str))
        (bind ?bool TRUE)
        (bind ?step (- ?j ?i))
        (bind ?period (sub-string  ?i  (- ?j 1) ?str))
        (while (and (<= (+ ?k ?step -1) ?len) ?bool)
                (bind ?bool (eq  ?period   (sub-string ?k (+ ?k ?step -1) ?str)))
                (bind ?k (+ ?k ?step)) )
        (return ?bool)
)
```

După cum se observă în ultima funcție definită se verifică într-o buclă while dacă subşirul cuprins între pozițiile *?i* şi *?j* se repetă, mai puțin ultimele caractere din şir, dacă acestea nu mai pot forma o perioadă. Acest fapt nu ar trebui să genereze pentru şiruri lungi de caractere (peste 20) şi ar prezenta şi avantajul neglijării ultimei cifre în cazul în care aceasta a fost rotunjită.

```
CLIPS> (period  (/  3  7  30)
0.(428571)
CLIPS> (period  “2342348”)
(234)
CLIPS> (verif  3  5  “0.12312312312345”)                ;se găseşte 0.(123)
TRUE
```

Pentru a verifica dacă ultimele caractere fac parte din perioadă, se poate modifica comanda *(return ?bool)* cu

(return (and ?bool (str-index (sub-string ?k ?len ?str) ?period)))

O altă problemă constă în faptul că pentru determinarea perioadei, nu se verifică doar partea fracționară şi acest fapt poate fi remediat, prin schimbarea valorii de start a variabilei ?i din *(loop-for-count (?i 1 (- ?len 1))*

(loop-for-count (?i (str-index "." ?str) (- ?len 1))

Funcția *period* poate fi folosită şi pentru determinarea unui subşir care se repetă într-un şir, fără a avea condiția ca datele de intrare să fie numere flotante (nefiind necesară în acest caz determinare părții fracționare). Funcția poate fi adaptată pentru a se determina acele şiruri care se repetă periodic de un număr de ori specificat la intrare printr-un argument.

(period "copiii") =====> *cop(i)*

2. Operații cu numere lungi – Dacă în exemplul anterior se afişa rezultatul unei împărțiri cu un număr mare de zecimale exacte, de această dată se doreşte efectuarea unor operații de adunare, înmulțire sau chiar ridicare la putere folosind astfel de numere. Pentru a realiza acest lucru vor fi folosite pentru stocarea numerelor tot şirurile de caractere.

Cu ajutorul metodei definite mai jos este supraîncărcat operatorul +, algoritmul prin care este implementată această funcției generică fiind foarte simplu. Primul operand *?a* trebuie să fie mai mic decât al doilea operand *?b*. Dacă această condiție nu este îndeplinită la apelul funcției, atunci se mai apelează o dată metoda (apel recursiv), de această dată poziția parametrilor fiind inversată. Se adună cele două numere cifră cu cifră, transportul spre cifrele următoare fiind stocat într-o variabilă *?carry* (inițial nulă). În momentul când cifrele primului număr (mai mic decât cel de-al doilea) au fost toate citite, se adună cifrele rămase din cel de-al doilea şir, ținându-se cont şi de variabila de transport *?carry*. În final dacă această variabilă nu este nulă cifrele(-a) acesteia trebuiesc adăugate la şirul *?rez*.

```
(defmethod + ((?a STRING) (?b STRING))
    (bind ?rez "")      (bind ?n (length ?a))
    (bind ?carry 0)     (bind ?m (length ?b))
    (if (> ?n ?m)
     then (+ ?b ?a)
     else (loop-for-count (?i 0 (- ?n 1))
              (bind ?ca (eval (sub-string (- ?n ?i) (- ?n ?i) ?a)))
              (bind ?cb (eval (sub-string (- ?m ?i) (- ?m ?i) ?b)))
              (bind ?suma (+ ?ca ?cb ?carry))
              (bind ?rez (str-cat (mod ?suma 10) ?rez))
              (bind ?carry (div ?suma 10))
          )
          (loop-for-count (?i ?n (- ?m 1))
              (if (neq ?carry 0)
                then (bind ?suma (+ (eval (sub-string (- ?m ?i) (- ?m ?i) ?b)) ?carry))
                     (bind ?rez (str-cat (mod ?suma 10) ?rez))
                     (bind ?carry (div ?suma 10))
                 else (return (str-cat (sub-string 1 (- ?m ?i) ?b) ?rez))
          )   )
         (if (neq ?carry 0)
          then (return (str-cat ?carry ?rez))
          else (return ?rez) ))
)
```

În locul apelului funcției *(eval (sub-string (- ?n ?i) (- ?n ?i) ?a)))*, care returnează cifra stocată în şirul de caractere *?a* pe poziția *(- ?n ?i)*, s-ar putea folosi o defini o funcție *digit* şi apelul în acest fel ar fi mai simplu *(digit (-?n ?i) ?a)*.

```
(deffunction digit (?poz ?str)
        (return (eval (sub-string ?poz ?poz ?str))) )
```

Se poate observa lansând comenzi direct la prompter cum pentru următoarea operație de adunare *(+ 12345678901 1)* se obține un rezultat eronat (overflow) *–539222986,* dar pentru apelul *(+ "12345678901" "1")* şirul returnat reprezintă rezultatul corect al operației *"12345678902"*. Din acest moment pot fi realizate operații de adunare cu numere foarte mari (mii de cifre).

Pentru a implementa şi operațiile de înmulțire cu astfel de şiruri, se defineşte o metodă >, care supraîncarcă operatorul "mai mare decât" şi care adaugă zerouri în partea dreaptă a numărului. Această funcție este foarte utilă deoarece după cum se poate observa mai jos, după ce primul operand este înmulțit cu fiecare cifră acelui de-al doilea operand, se adună rezultatele astfel obținute.

```
  1 2 3 4 *
      2 1
  1 2 3 4          1234*11 = 1234*20 +1234*1
2 4 6 8  .         CLIPS> (+ (> (*  "1234" "2") 1)  (*  "1234" "1") )
2 5 9 1 4
```

Pentru a evita situația când numărul întreg ?poz ce semnifică numărul de deplasări aplicate şirului, este introdus ca prim argument şi CLIPS-ul ar putea furniza o eroare, deoarece nu ar găsi o metodă aplicabilă, se defineşte şi o funcție care să accepte argumentele introduse în această ordine.

```
(defmethod > ((?str STRING) (?poz INTEGER))
        (loop-for-count (?i 1 ?poz)
                (bind ?str (str-cat ?str "0"))  )
        (return ?str)
)
(defmethod > ((?poz INTEGER) (?str STRING))
        (> ?str ?poz) )                          ;se apeleaza prima metoda >

(defmethod * ((?str STRING) (?n NUMBER))
        (bind ?rez "")
        (bind ?carry 0)
        (bind ?len (length ?str))
        (loop-for-count (?i 0 (- ?len 1))
                (bind ?suma (+ ?carry (* (digit (- ?len ?i) ?str) ?n)))
                (bind ?carry (div ?suma 10))
                (bind ?rez (str-cat (mod ?suma 10) ?rez))
        )
        (if (neq ?carry 0)
         then (return (str-cat ?carry ?rez))
         else (return ?rez) )
)

(defmethod * ((?n NUMBER) (?str STRING))
        (* ?str ?n) )
```

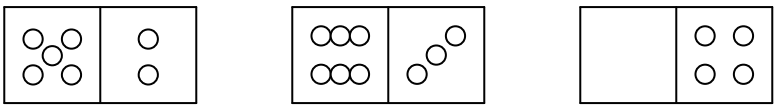
Pentru înmulțirea unui şir cu un număr (în cazul nostru o cifră) se supraîncarcă operatorul de înmulțire. Şi în această funcție este folosită o variabilă *?carry* care păstrează valoarea transportului spre biții imediat următori. Această funcție generică va fi apelată de metoda definită mai jos, valoarea returnată va fi deplasată în funcție de poziția cifrei la care este înmulțit primul parametru, şirul de caractere astfel obținut fiind adăugat la rezultatul final. Sunt apelate astfel trei metode definite mai sus +, > ,* . Se observă faptul că odată definite un număr de funcții generice cu operațiile de bază, se pot implementa alte funcții folosindu-ne de cele deja existente.

```
(defmethod * ((?a STRING) (?b STRING))
        (bind ?rez "")
        (bind ?m (length ?b))
        (loop-for-count (?i 0 (- ?m 1))
              (bind ?rez (+ (> (* ?a (eval (sub-string (- ?m ?i) (- ?m ?i) ?b))) ?i) ?rez))
         )
        (return ?rez)
)
```

Supraîncărcarea operatorului ** de 'ridicare la putere' este foarte uşor de realizat, deoarece această operație corespunde înmulțirii bazei ?b cu ea însăşi de numărul ?p ce specificat puterea sau exponentul.

```
(defmethod ** ((?b INTEGER (> ?b 0)) (?p INTEGER (> ?p 0)))
        (bind ?rez (str-cat ?b))
        (loop-for-count (?i 1 ?p)
                (bind ?rez (* ?rez ?b))
        )
        (return ?rez)
)
```

3. <u>Jocul de domino</u> – După cum se ştie jocul de 'Domino' conține 28 de piese, fiecare piesă având asociate pentru fiecare jumătate a sa două valori (între 0 şi 6).



La începutul partidei fiecare jucător extrage 6 piese din set. Cel care este primul la mutare pune o piesă jos şi din acest moment regula de joc constă în alipirea unei piese la unul din cele două capete ale şirului de piese depuse.
Presupunând dat şirul următor: | 1 | 0 | | 0 | 6 | | 6 | 5 | | 5 | 5 | jucătorul la mutare va trebui să pună jos o piesă a cărei capete sunt 1 sau 5. Dacă acesta nu are în mână o astfel de piesă, atunci va extrage piese din pachetul rămas, până când va putea să-şi facă mutarea. Dacă s-au terminat piesele din pachet şi un jucător nu poate muta, atunci se cedează mutarea celuilalt jucător. Atunci când un jucător are o piesă bună de mutat, el este forțat să facă acea mutare. Oricum strategia jocului constă în a lua

cât mai puține piese de jos şi în a forța adversarul să extragă piese, deoarece la final câştigător este acela care termină primul piesele din mână.

Pentru a juca cât mai bine, calculatorul trebuie să țină cont în permanență de piesele jucate şi de cele din mână. Pentru aceasta sunt folosite trei variabile globale *?*s** care păstrează piesele care au rămas de extras, *?*c** pentru piesele calculatorului şi *?*t** pentru piesele care au fost jucate. Pentru a fi "fair-play" calculatorul nu va ține cont în jocul său de piesele din stivă *?*s** şi astfel el nu va şti niciodată care este piesa (piesele) următoare de extras sau care sunt piesele oponentului uman. Numărul total al pieselor rămase este păstrat de variabila *?*n**.

```
(defglobal ?*n* = 0                          ;numărul pieselor rămase
           ?*s* = (create$ )                 ;stiva cu piesele
           ?*c* = (create$ 0 0 0 0 0 0 0)    ;evidenta piese calculator
           ?*t* = (create$ 0 0 0 0 0 0 0))   ;evidenta tuturor pieselor jucate

(deffacts date-initiale
        (puncte 0 1 2 3 4 5 6)
        (piese_calculator )
        (piese_human ) )

(deffunction incr (?poz1 ?poz2 $?sir)
        (bind ?poz1 (+ ?poz1 1)) (bind ?poz2 (+ ?poz2 1))
        (bind $?sir (replace$ $?sir ?poz1 ?poz1 (+ (nth$ ?poz1 $?sir) 1)))
        (if (neq ?poz2 ?poz1)
         then (bind $?sir (replace$ $?sir ?poz2 ?poz2 (+ (nth$ ?poz2 $?sir) 1))) )
        (return $?sir) )

(deffunction decr (?poz1 ?poz2 $?sir)
        (bind ?poz1 (+ ?poz1 1)) (bind ?poz2 (+ ?poz2 1))
        (bind $?sir (replace$ $?sir ?poz1 ?poz1 (- (nth$ ?poz1 $?sir) 1)))
        (if (neq ?poz2 ?poz1)
         then (bind $?sir (replace$ $?sir ?poz2 ?poz2 (- (nth$ ?poz2 $?sir) 1))) )
        (return $?sir) )

(defrule creare_piese
        (declare (salience 20))
        (puncte $? ?p1 $?)
        (puncte $? ?p2&:(>= ?p1 ?p2) $?)
         =>
        (assert (piesa (bind ?*n* (+ ?*n* 1)) ?p2 ?p1))
        (set-strategy random) )

(defrule amestecare_piese
        (declare (salience 10))
        (piesa ?n ? ?)
        =>
        (bind ?*s* (create$ ?*s* ?n)) )
```

```
(defrule cine_muta_primul
        =>
        (set-strategy depth)
        (printout t "Cine muta primul (C: Calculatorul ; H:Omul)? ")
        (while (neq (bind ?r (read)) C c H h C O o)
                (printout t "Cine muta primul (C: Calculatorul ; H:Omul)? ") )
        (if (or (eq ?r C) (eq ?r c))
         then (assert (player C))
        else (assert (player H)) ) )

(defrule 6_piese_pentru_fiecare
        (declare (salience 10))
        (player ?p&C|H)
        (piese_calculator $?v1)
        (piese_human $?v2 &: (< (length (create$ $?v2 $?v1)) 12))
        (not (capete ? ?))
         =>
        (if (or (and (eq ?p C) (< (length $?v1) 6))
                (and (eq ?p H) (= (length $?v2) 6)) )
         then (assert (calculatorul_ia_piesa (nth$ 1 ?*s*)))
         else (assert (human_ia_piesa (nth$ 1 ?*s*)))
        (bind ?*s* (delete$ ?*s* 1 1)) )
```

Pentru a modifica numărul pieselor au fost realizate special două funcţii *incr* şi *decr*. Acestea sunt foarte utile atunci când calculatorul pune (sau extrage) o piesă jos, fiind necesară creştea (sau micşorarea) numărului pieselor în funcţie de capetele acestora. Pentru crearea pieselor sunt necesare două fapte ce stochează numărul punctelor de pe capetele piesei. Regula se va aprinde de exact 28 ori, cât este necesar pentru a forma pachetul pieselor de domino. Pentru o activare aleatorie a regulii *amestecare_piese* a fost folosită strategia **random**. Odată formată stiva *?*s** se poate stabili cine începe primul şi se extrag 6 piese pentru fiecare jucător. Regula se va aprinde atât timp cât cei doi oponenţi nu au extras numărul de piese necesar pentru a începe jocul ... *&:(< (length (create$ $?v2 $?v1)) 12))*. Pentru fiecare piesă luată, valoarea variabilelor globale se modifică. Următoarele două reguli se vor activa de fiecare dată când unul dintre jucători va extrage o piesă.

```
(defrule calculatorul_ia_o_piesa
        (declare (salience 10))
        ?f1 <- (calculatorul_ia_piesa ?n)
        ?f2 <- (piese_calculator $?v1)
        (piesa ?n ?p1 ?p2)
        =>
        (retract ?f1 ?f2)
        (assert (piese_calculator $?v1 ?n ))
        (bind ?*n* (- ?*n* 1))
        (bind ?*c* (incr ?p1 ?p2 ?*c*)) )
```

```
(defrule human_ia_o_piesa
        (declare (salience 10))
        ?f1 <- (human_ia_piesa ?n)
        ?f2 <- (piese_human $?v2)
        (piesa ?n ?p1 ?p2)
        =>
        (retract ?f1 ?f2)
        (bind ?*n* (- ?*n* 1))
        (assert (piese_human $?v2 ?n)) )

(defmethod comp (?p1 ?p2)
   (bind ?p1 (+ ?p1 1)) (bind ?p2 (+ ?p2 1))
   (return (> (+ (nth$ ?p1 ?*c*)(nth$ ?p1 ?*t*))
                 (+ (nth$ ?p2 ?*c*)(nth$ ?p2 ?*t*)) ))
)
(defmethod comp (?p1 ?p2 ?p3 ?p4)
   (bind ?p1(+ ?p1 1)) (bind ?p2(+ ?p2 1)) (bind ?p3(+ ?p3 1)) (bind ?p4(+ ?p4 1))
   (return (>= (+(nth$ ?p1 ?*c*)(nth$ ?p2 ?*c*)(nth$ ?p1 ?*t*)(nth$ ?p2 ?*t*))
                 (+(nth$ ?p3 ?*c*)(nth$ ?p4 ?*c*)(nth$ ?p3 ?*t*)(nth$ ?p4 ?*t*)) ))
)
```

Deoarece s-a dorit implementarea unei funcţii care să accepte un număr de doi sau patru parametri s-a apelat la funcţiile generice. Din acest moment pentru fiecare mutare ce o va face calculatorul va folosi funcţia *comp*, pentru a verifica dacă mutarea pe care o va face este cea mai avantajoasă dintre toate (**forall**) mutările posibile.

```
(defrule prima_mutare_calculator
        ?f1 <- (player C)
        ?f2 <- (piese_calculator $?f ?n1 $?l)
        ?f3 <- (piesa ?n1 ?p1 ?p2)
        (forall  (piesa ?n2&:(or(member$ ?n2 $?f) (member$ ?n2 $?l)) ?p3 ?p4)
                 (test (comp ?p1 ?p2 ?p3 ?p4)) )
        (not (capete ? ?))
        =>
        (retract ?f1 ?f2 ?f3)
        (bind ?*c* (decr ?p1 ?p2 ?*c*)) (bind ?*t* (incr ?p1 ?p2 ?*t*))
        (assert (player H) (capete ?p1 ?p2) (piese_calculator $?f $?l))
        (printout t "Calculatorul pune piesa: " ?p1 ?p2 t "Capete: " ?p1 ?p2 t) )

(defrule afisare_piese_human
        (declare (salience 5))
        (player H)
        (piese_human $? ?n $?)
        (piesa ?n ?p1 ?p2)
        =>
        (printout t ?n "-[" ?p1 "," ?p2 "]; " ) )
```

```
(defrule alegere_mutare_human
        (player H)
        (piese_human $?h)
        =>
        (printout t t "Human pune piesa : ")
        (while (not (member$ (bind ?n (read)) $?h))
                (printout t "Human pune piesa : ") )
        (assert (human ?n)) )

(defrule prima_mutare_human
        ?f1 <- (player H)
        ?f2 <- (human ?n)
        ?f3 <- (piese_human $?f ?n $?l)
        ?f4 <- (piesa ?n ?p1 ?p2)
        (not (capete ? ?))
        =>
        (retract ?f1 ?f2 ?f3 ?f4)
        (assert (player C) (capete ?p1 ?p2) (piese_human $?f $?l))
        (bind ?*t* (incr ?p1 ?p2 ?*t*))
        (printout t "Capete: " ?p1 ?p2 t) )

(defrule mutare_human_invalida
        (player H)
        ?f <- (human ?n)
        (piesa ?n ?p1 ?p2)
        (capete ?c1&~?p1&~?p2 ?c2&~?p1&~?p2)
         =>
        (retract ?f)
        (refresh alegere_mutare_human)
        (printout t "Incorect!!! Capete: " ?c1 ?c2)  )

(defrule mutare_human_valida
        ?f1 <- (player H)
        ?f2 <- (human ?n)
        ?f3 <- (piesa ?n ?p1 ?p2)
        ?f4 <- (piese_human $?f ?n $?l)
        (or ?f5<-(capete ?c1&?p1|?p2 ?c2) ?f5<-(capete ?c1 ?c2&?p1|?p2))
        =>
        (retract ?f1 ?f2 ?f3 ?f4 ?f5)
        (assert (player C) (piese_human $?f $?l))
        (bind ?*t* (incr ?p1 ?p2 ?*t*))
        (if (eq ?p1 ?c1)
        then (assert(capete ?p2 ?c2)) (printout t "Capete: " ?p2 ?c2 t)
        else (if (eq ?p1 ?c2)
              then (assert(capete ?c1 ?p2)) (printout t "Capete: " ?c1 ?p2 t)
              else (if (eq ?p2 ?c1)
                    then (assert (capete ?p1 ?c2)) (printout t "Capete: " ?p1 ?c2 t)
                    else (assert (capete ?c1 ?p1)) (printout t "Capete: " ?c1 ?p1 t) ) ) ) )
```

```
(defrule mutare_calculator_1
        ?f1 <- (player C)
        ?f2 <- (piese_calculator $?f ?n1 $?l)
        ?f3 <- (capete ?c1 ?c2)
        ?f4 <- (piesa ?n1 ?p1&?c1|?c2  ?p2)
        (not(exists (piesa ?n2&:(or(member$ ?n2 $?f)(member$ ?n2 $?l)) ?p3 ?p4)
                (test (or (and (or(= ?p3 ?c1)(= ?p3 ?c2)) (comp ?p4 ?p2))
                          (and (or(= ?p4 ?c1)(= ?p4 ?c2)) (comp ?p3 ?p2)) )) ))
        =>
        (retract ?f1 ?f2 ?f3 ?f4)
        (bind ?*c* (decr ?p1 ?p2 ?*c*)) (bind ?*t* (incr ?p1 ?p2 ?*t*))
        (assert (player H) (piese_calculator $?f $?l))
        (printout t "Calculatorul pune piesa: [" ?p1 "," ?p2 "]" t)
        (if (eq ?p1 ?c1)
         then (assert (capete ?p2 ?c2)) (printout t "Capete: " ?p2 ?c2 t)
         else (assert (capete ?c1 ?p2)) (printout t "Capete: " ?c1 ?p2 t) ) )

(defrule mutare_calculator_2
        ?f1 <- (player C)
        ?f2 <- (piese_calculator $?f ?n1 $?l)
        ?f3 <- (capete ?c1 ?c2)
        ?f4 <- (piesa ?n1 ?p1 ?p2&?c1|?c2)
        (not(exists (piesa ?n2&:(or(member$ ?n2 $?f)(member$ ?n2 $?l)) ?p3 ?p4)
                (test (or (and (or(= ?p3 ?c1)(= ?p3 ?c2)) (comp ?p4 ?p1))
                          (and (or(= ?p4 ?c1)(= ?p4 ?c2)) (comp ?p3 ?p1)) )) ))
        =>
        (retract ?f1 ?f2 ?f3 ?f4)
        (bind ?*c* (decr ?p1 ?p2 ?*c*)) (bind ?*t* (incr ?p1 ?p2 ?*t*))
        (assert (player H) (piese_calculator $?f $?l))
        (printout t "Calculatorul pune piesa: [" ?p1 "," ?p2 "]" t)
        (if (eq ?p2 ?c1)
         then (assert (capete ?p1 ?c2)) (printout t "Capete: " ?p1 ?c2 t)
         else (assert (capete ?c1 ?p1)) (printout t "Capete: " ?c1 ?p1 t) ) )
```

Exceptând prima mutarea a calculatorului, când nu există nici o piesă jos şi este necesară alegerea unei piese care să fie avantajoasă pentru ambele capete (doar atunci este folosită funcţia generică *comp* cu 4 parametri), pentru mutările următoare deoarece după alipire va mai rămâne un singur capăt liber, se încercă crearea unor şanse minime de mutare pentru oponentul uman. În funcţie de piesele puse jos şi de cele din 'mâna' calculatorului se doreşte obţinerea unor capete libere pentru care se deţine majoritatea pieselor rămase. Printr-un astfel de joc închis se încearcă forţarea adversarului să extragă piese şi mărirea şanselor de câştig.

Regulile *mutare_calculator_1* şi *mutare_calculator_2* pot fi compactizate într-o singură regulă, dar codul devine mai greu de citit şi de înţeles. Se preferă această formă în care se poate uşor evidenţia capătul piesei ce a rămas liber după alipirea cu şirul pieselor depuse.

Mai trebuie realizată o regulă care să forțeze jucătorii să extragă piese atunci când nu mai au cum să facă o mutare. În acest fel se asigură faptul că un jucător va lua o piesă de jos, doar dacă într-adevăr nu mai are nici o piesă în mână, care să se potrivească capetelor rămase libere.

```
(defrule tragere_piesa_fortata
        (declare (salience 10))
        (or (and ?f <- (player ?p&C) (piese_calculator $?v))
            (and ?f <- (player ?p&H) (piese_human $?v)) )
        (capete ?c1 ?c2)
        (not (exists (piesa ?n&:(member$ ?n $?v) ?p1 ?p2)
                    (test (or(= ?p1 ?c1)(= ?p1 ?c2)(= ?p2 ?c1)(= ?p2 ?c2))) ))
        =>
        (bind ?n (nth$ 1 ?*s*))
        (if (neq ?n nil)
         then (if (eq ?p C)
                 then (assert (calculatorul_ia_piesa ?n))
                 else (assert (human_ia_piesa ?n )) )
                      (bind ?*s* (delete$ ?*s* 1 1))
        else (retract ?f)
             (if (eq ?p C)
              then (assert (player H))
              else (assert (player C)) ) )
)

(defrule stop_joc
        (declare (salience 20))
        (or (and ?f <- (player ?p&H) (piese_calculator $?v))
            (and ?f <- (player ?p&C) (piese_human $?v)) )
        (test (eq (length$ $?v) 0))
        (capete ? ?)
        =>
        (retract ?f)
        (if (eq ?p H)
         then (printout t t "AI PIERDUT !!!" t)
         else (printout t t "AI CASTIGAT !!!" t) ) )
```

Pentru a înțelege mai bine cum funcționează programul se recomandă rularea acestuia pas cu pas folosind comanda *(run1)* şi urmărind modificările realizate asupra variabilelor globale. Se pot lansa funcțiile incr, decr, comp de la prompter şi se poate observa modul de funcționare al fiecăreia:

```
CLIPS> (incr 2 3 (create$ 0 1 2 3 4 5))
(0 1 3 4 4 5))
CLIPS> (decr 4 5 (create$ 0 1 3 4  4 5))
(0 1 3 4 3 4)
```