Cap.4 Funcţii de Intrare/ Ieşire

4.1. Funcţii matematice elementare

Deşi avem în CLIPS şi funcţii pentru realizarea de calcule, trebuie reţinut faptul că un limbaj pentru realizarea sistemelor expert cum este CLIPS-ul nu este destinat calculelor matematice. Aceste funcţii matematice sunt folosite doar în cazul în care aplicaţia pe care o rulăm necesită astfel de operaţii. CLIPS-ul dispune de următoarele funcţii matematice:

*+ adunare - scădere * înmulţire / împărţire ** ridicare la putere*

Expresiile numerice sunt reprezentate în CLIPS ca şi în LISP, astfel încât expresia 2+3 trebuie scrisă în formatul prefix (+ 2 3). Trecerea de la formatul infix folosit de obicei în scrierea expresiilor matematice la formatul prefix, se realizează prin aşezarea operatorului înaintea operanzilor. Realizarea acestei conversii este relativ simplă. Să luăm de exemplu inecuaţia următoare, unde avem date două puncte P1(x1,y1) şi P2(x2,y2) şi dorim să aflăm dacă segmentul se află în interiorul cercului de raza r şi cu centrul în unul din cele două puncte:

(x1 - x2) ** 2 + (y1 – y2) ** 2 > r ** 2

Se poate face (mintal) înlocuirea lungimii segmentului cu (D):

D ** 2 > r ** 2

Formatul prefix pentru inecuaţia de mai sus este:

(> (D ** 2) (r ** 2))

Fiecare dintre expresiile (X) şi (Y) este convertită în formatul prefix:

(> (+ (** (- x1 x2) 2) (** (- y1 y2) 2)) (** r 2))

Cel mai simplu mod de a verifica dacă o expresie numerică este scrisă corect este să o scriem la prompter (la **top level**). De exemplu dacă se va introduce (+ 2 2) sau (/ 2 3) la prompter în urma evaluării expresiei se va obţine:

```
CLIPS> (+2 2)
4
CLIPS> (/ 2 3)
0.66666667                ; eroare de aproximare in ultimul digit.
```

Majoritatea funcţiilor (ca şi funcţiile matematice) returnează o valoare, iar aceasta poate fi de tip integer, float, symbol, string, sau chiar o valoare multicâmp. Există şi funcţii care nu returnează o valoare, de exemplu *(facts)* sau *(agenda*), efectul acestora fiind afişarea listei de fapte sau de reguli activate, spre deosebire de celelalte funcţii care afişează rezultatul obţinut în urma evaluării unei expresii.

Valoarea returnată de operaţiile + , - , * sunt un întreg numai atunci când toate argumentele funcţiei sunt întregi. In cazul în care măcar unul dintre argumente este de tipul float, atunci şi rezultatul (valoarea returnată) este de tipul float, acest lucru putând duce uneori la erori. De exemplu:

```
CLIPS> ( +  2  3.0)
5.0
CLIPS> ( *  2  3.0)
6.0
```

Folosirea unei notaţii prefix (sau postfix) permite folosirea unui număr variabil de argumente, lucru imposibil la notaţia infix. Multe funcţii în CLIPS folosesc un număr variabil de argumente, printre acestea şi funcţiile matematice (mai puţin ** ridicarea la putere care cere doar două argumente). De exemplu:

```
CLIPS> ( +  2  3  4)
9
CLIPS> ( /  2  3  4)
0.1666667
```

Important de reţinut este faptul că în CLIPS, ca şi în LISP nu există o **precedenţă** fixată între operaţiile aritmetice. Dacă în alte limbaje înmulţirea şi împărţirea au un rang mai mare decât adunarea şi scăderea, fiind efectuate mai întâi operaţiile de rang mai înalt, în CLIPS însă neexistând o prioritate pre-integrată, ordinea efectuării operaţiilor este de la stânga la dreapta, precedenţa fiind fixată de paranteze. La scrierea calculelor mixte în forma prefix, trebuie respectată ordinea parantezelor şi a operanzilor. Astfel dacă se doreşte calcularea expresiei: 2 + 3 * 4 se face mai întâi înmulţirea lui 3 cu 4 şi apoi adunarea lui 2. Precedenţa ar trebui realizată în CLIPS astfel:

```
CLIPS> (+ 2 (* 3 4))
14
```

Dacă nu s-ar fi respectat această ordine pentru operatori, atunci expresia evaluată ar fi fost echivalentă cu (2+3)*4:

```
CLIPS> (* (+ 2 3) 4)
20
```

In general, expresiile matematice (şi nu numai) pot fi introduse în alte expresii (de exemplu plasarea unei expresii într-o comandă *assert*):

```
CLIPS> (assert (rezultat (+ 3 4))
                    (intreg (integer 2.3)) )
CLIPS> (facts)
f-0     (rezultat 7)
f-1     (intreg 2)
```

De asemeni, deoarece numele funcţiilor matematice sunt tot simboluri, acestea pot fi utilizate ca şi cum am folosi oricare alte simboluri (prin excluderea parantezelor CLIPS-ul nu le va mai identifica ca fiind funcţii matematice). Astfel acestea pot fi utilizate drept câmpuri într-un fapt:

```
CLIPS> (assert (expresie  2 + 3 +4))
CLIPS> (facts)
f-0     (expresie 2 + 3 + 4)
```

Parantezele însă sunt utilizate de CLIPS ca delimitatori şi din acest motiv nu este posibilă utilizarea lor ca şi în cazul altor simboluri. Pentru a le amplasa în fapte sau ca argumente la o funcție, acestea trebuiesc puse între ghilimele (pentru a fi transformate în şiruri).

4.2. Sumarea valorilor utilizând reguli

Ca exemplu de utilizare a funcțiilor pentru a realiza calcule, vom considera problema sumării ariilor unui grup de dreptunghiuri. Lungimea şi lățimea dreptunghiurilor vor fi specificate prin folosirea unei construcții *deftemplate*. Suma ariilor dreptunghiurilor poate fi păstrată într-un fapt ordonat: *(suma 20)*.

```
(deftemplate  dreptunghi   (slot lungime)   (slot latime))

(deffacts informatii-initiale
                (dreptunghi  (lungime 6) (latime 8))
                (dreptunghi  (lungime 2) (latime 5))
                (suma 0) )

(defrule suma-dreptunghi
                (dreptunghi (lungime ?lungime) (latime ?latime))
                ?suma  <- (suma  ?total)
                =>
                (retract ?suma)
                (assert  (suma (+  ?total  (* ?lungime ?latime))))  )
```

Regula de mai sus de sumare a ariilor produce o buclă infinită, în care se adună aria unui singur fapt *dreptunghi*, deoarece faptului *suma* este retractat şi reasertat în aceiaşi regulă. O soluție pentru a rezolva această problemă este ştergerea faptului *dreptunghi* după ce aria acestuia a fost adăugată la faptul *suma*. Acest lucru împiedică aprinderea regulii de două ori pentru acelaşi dreptunghi. Dar presupunând faptul că nu se doreşte alterarea bazei de date (faptele *dreptunghi* pot fi folosite mai apoi şi de alte reguli) această soluție nu este indicată. Ar trebui în acest caz realizat unui fapt temporar care memorează aria dreptunghiului şi care să fie retractat într-o regulă imediat următoare ce realizează sumarea ariilor (ştergerea faptului în cadrul altei reguli va împiedica crearea unei alte bucle infinite).

```
(defrule  suma-dreptunghi
        (dreptunghi (lungime  ?lungime) (latime ?latime))
        =>
        (assert (aduna-arie  (*  ?lungime  ?latime))) )

(defrule suma-arii
        ?f1 <- (suma ?total)
        ?f2 <- (aduna-arie ?aria)
        =>
        (retract ?f1 ?f2)
        (assert (suma (+ ?total ?aria)))
        (printout  t  "Suma actuala: "  (+ ?total ?aria) crlf) )
```

```
CLIPS> (reset)
CLIPS> (run)
Suma actuala:10
Suma actuala:58
```

4.3. Funcţia BIND

Adesea este utilă stocarea unei valori într-o variabilă temporară pentru evitarea recalculării (de exemplu în regula de mai sus unde s-a folosit pentru a doua oară în funcţia de afişare *printout* expresia de calculare a ariei). Evitarea recalculării poate fi crucială când funcţiile produc efecte secundare. Funcţia **bind** legă de o variabilă valoarea unei expresii. Sintaxa acesteia este:

```
(bind  <variable>  <value>)
```

Trebuie reţinut faptul că dacă variabila <variable> este de tipul *single-field* (un singur câmp) atunci şi noua valoare <value> trebuie să fie de tipul *single-field*. De asemenea acelaşi lucru trebuie să se întâmple şi atunci când valoarea expresiei <value> este de tipul *multifield* (multicâmp). De exemplu pentru regula:

```
(defrule  suma-areas
        ?suma <- (suma ?total)
        ?new-area <- (add-to-suma ?area)
        =>
        (retract ?suma ?new-area)
        (printout  t  "Adding " ?area "  to " ?total crlf)
        (printout  t  "New suma is " (+ ?total ?area) crlf)
        (assert (suma (+ ?total ?area))) )
```

se observă folosirea în cadrul RHS-ului regulii de două ori a expresiei *(+ ?total ?area)*. Prin înlocuirea acestor două evaluări cu o funcţie *bind* se elimină calculele inutile. Astfel noua regulă va arăta astfel:

```
(defrule  suma-areas
        ?suma <- (suma ?total)
        ?new-area <- (add-to-suma ?area)
        =>
        (retract ?suma ?new-area)
        (printout  t  "Adding " ?area "  to " ?total crlf)
        (bind ?new-total  (+ ?total ?area))
        (printout  t  "New suma is " ?new-total crlf)
        (assert (suma ?new-total)) )
```

In plus la crearea noilor variabile în partea de concluzie a regulii (RHS), funcţia *bind* poate fi utilizată şi pentru a 'relega' o variabilă utilizată în partea stângă a unei reguli (LHS) de o nouă valoare. De reţinut faptul că funcţia *bind* nu poate fi folosită în nici un caz în LHS-ul regulii (ea fiind o acţiune şi în nici un caz o condiţie), în astfel de situaţii CLIPS-ul semnalând eroare de sintaxă.

4.4. Funcția READ

Un sistem expert cere date de intrare utilizatorului. CLIPS asigură citirea de date de la tastatură cu ajutorul funcției **read**. Sintaxa funcției *read* este:

(read [<logical-name>])

Această funcție citește un singur câmp de la <logical-name> specificat. Se observă că argumentul funcției este opțional, în cazul când funcția nu are nici un argument datele fiind citite implicit de la tastatură. Fie exemplul următor:

```
CLIPS> (clear)
CLIPS> (defrule  introduceti_numele_mic
                 =>
                 (printout  t  " Care este primul tau nume? ")
                 (bind  ?raspuns  (read))
                 (assert  (nume-utilizator ?raspuns))
                 (printout  t  "Buna, " ?raspuns "!"crlf) )
```

Trebuie reținut faptul că funcția *read* așteaptă un carriage return (CR) după care va citi simbolul introdus. Deoarece funcția *read* poate fi folosită doar pentru a citi un singur câmp odată, pentru cazul în care s-ar introduce şi primul şi al doilea nume aceasta ar citi doar pe primul (chiar dacă variabila ar fi declarată de tip multicâmp s-ar obține acelaşi lucru). Toate caracterele suplimentare aflate după primul câmp sunt ignorate. De exemplu dacă s-ar introduce următoarele:

```
CLIPS> (reset)
CLIPS> (run)
Care este primul tau nume?  Marius Dan CR↵
Buna, Marius!
CLIPS> (refresh  introduceti_numele_mic)
CLIPS>(run)
Care este primul tau nume?  "Marius Dan" CR↵
Buna, Marius Dan!
CLIPS> (facts)
f-0     (initial-fact)
f-1     (nume-utilizator  Marius)
f-2     (nume-utilizator  "Marius Dan")
For a total of 2 facts.
```

Pentru a forța funcția *read* să citească toate simbolurile introduse, acestea trebuiesc introduse între ghilimele şi în acest caz datele de intrare vor forma un şir de caractere şi deci un singur câmp. Funcția *read* returnează întotdeauna un tip de dată primitivă şi acceptă să fie introduse şi câmpuri care nu sunt simboluri, întregi sau float. In acest caz el va plasa primul câmp între paranteze şi îl va considera ca fiind un şir după cum se poate vedea şi în următoarea linie de comandă:

```
CLIPS> (read)
(↵
")"
```

4.5. Funcţia OPEN

Pe lângă introducerea de la tastatură şi ieşirea pe terminal CLIPS-ul poate de asemenea realiza citirea şi scrierea într-un fişier. Înainte de a fi accesat pentru citire sau scriere un fişier trebuie deschis cu funcţia **open**. Numărul total de fişiere care pot fi deschise simultan este dependent de sistemul se operare şi hardware-ul maşinii. Sintaxa funcţiei *open* este următoarea:

(open <file-name> <logical-name> [<mode>])

Argumentele funcţiei *open* sunt în ordinea apariţiei : 1. numele fişierului reprezentat printr-un şir de caractere în care poate fi inclusă şi calea (directorul în care se află fişierul); 2. numele logic care va fi folosit de celelalte funcţii de I/O pentru accesarea fişierului (acesta deşi poate fi uneori identic cu numele fişierului se recomandă pentru evitarea confuziei folosirea unui alt nume); 3. modul de accesare al fişierului, care poate fi unul dintre următoarele patru moduri posibile:

1. "r" -Read access only
2. "w" -Write access only
3. "r+" -Read and write access
4. "a" -Append access only

Dacă nu se specifică nici un mod de deschidere a unui fişier atunci acesta este în mod implicit "r" (deschis pentru citire). In exemplul:

(open "C:\\Temp\\fisier.clp" date "a")

se referă prin numele logic *date* fişierul *fisier.clp* (în locul acestuia putea fi folosit orice nume). Marele avantaj al folosirii acestor nume logice este acela de a modifica în cazul în care se doreşte accesarea unui alt fişier doar a numelui din funcţia *open*, toate celelalte funcţii de intrare / ieşire folosind numele logic.

Fişierul de mai sus este deschis pentru adăugare. Ca şi în cazul funcţiilor predicative, funcţia *open* returnează TRUE în caz de succes şi FALSE în caz de eroare la deschiderea fişierului (de ex. deschiderea pentru citire a unui fişier care nu există). Valoarea returnată poate fi folosită pentru oprirea eventuală a programului sau pentru a se cere introducerea datelor de la tastatură.

4.6. Funcţia CLOSE

După ce un fişier deschis nu mai este util, acesta poate fi închis. Dacă un fişier nu este închis, nu există nici o garanţie că informaţia scrisă în acesta va fi corect salvată. De asemenea cu cât un fişier rămâne mai mult timp cu atât creşte probabilitatea ca în cazul unei căderi de tensiune sau altă funcţionare proastă a sistemului, informaţia să nu fie salvată. Din acest motiv se recomandă realizarea unei reguli care să realizeze o salvare automată la anumite intervale de timp (prin închiderea şi redeschiderea fişierului). Sintaxa funcţiei este următoarea:

(close [<logical-name>])

unde argumentul opțional specifică numele logic al fişierului care va fi închis. Dacă nu este precizat nici un argument atunci toate fişierele deschise vor fi închise. In cazul terminării unui program CLIPS-ul nu va interoga utilizatorul dacă vrea să închidă fişierele rămase deschise. Funcția returnează TRUE în cazul închiderii cu succes a fişierului, sau FALSE în caz contrar. Unica problemă în cazul închiderii unui fişier, constă în faptul că această operație trebuie să se realizeze într-o regulă de prioritate mică, după ce au fost citite (sau scrise) datele care erau necesare pentru o funcționare corectă a programului.

```
CLIPS> (open "myfile.clp" writeFile "w")
TRUE
CLIPS> (open "MS-DOS\\directory\\file.clp" readFile)
TRUE
CLIPS> (close writeFile)
TRUE
CLIPS> (close writeFile)
FALSE
CLIPS> (close)
TRUE
CLIPS> (close)
FALSE
```

4.7. Scrierea şi citirea dintr-un fişier

In toate exemplele anterioare datele de intrare au fost citite de la tastatură şi afişate pe ecran. Funcția **printout** a folosit până în acest moment ca nume logic doar **t** (adică terminalul – ieşirea standard). Orice alt nume logic valid poate însă fi folosit şi astfel se poate trimite la ieşire date spre altă destinație decât pe ecran. In mod similar când este folosit **t** , ca nume logic pentru intrarea standard la o funcție de intrare, este utilizată tastatura *(read t)* este echivalent cu *(read)*. Următorul exemplu ilustrează scrierea într-un fişier:

```
CLIPS> (open "exemplu.dat" exemplu "w")
TRUE
CLIPS> (printout exemplu "Manuela" crlf)
CLIPS> (printout exemplu 2 crlf)
CLIPS> (close exemplu)
TRUE
```

In fişierul *exemplu* deschis pentru scriere sunt introduse două valori (un simbol şi un întreg). Se poate realiza verificarea valorilor scrise cu ajutorul funcției **read**. Sintaxa comenzii este următoarea :

```
(read [<logical-name>])
```

unde numele logic este în mod implicit **t** - tastatura. De această dată va fi folosit numele logic ataşat fişierului *exemplu.dat* deschis pentru citire.

```
CLIPS> (open "exemplu.dat" exemplu "r")        ; "r" este opțional
TRUE
```

```
CLIPS> (read exemplu)
Manuela
CLIPS> (read exemplu)
2
CLIPS> (read exemplu)
EOF                                    ; EOF - sfârşit de fişier
CLIPS> (close exemplu)
TRUE
```

Verificând valoarea returnată de funcţia *read* se poate determina când nu au mai rămas date de citit în fişier (caz în care valoarea returnată este EOF).

4.8. Funcţia FORMAT

De multe ori se doreşte formatarea ieşirii dintr-un program CLIPS, de exemplu aranjarea datelor în tabele. Faţă de funcţia *printout* există o funcţie specială **format** care oferă o mare varietate de formaturi şi un control mult mai bun asupra datelor la ieşire. Ea este similară cu funcţia **printf** din limbajul C.

(format <logical-name> <control-string> <parameters>*)

Parametrii funcţiei sunt în ordinea apariţiei :1. numele logic spre care este trimisă ieşirea (poate fi şi ieşirea standard – **t** terminalul); 2. şirul de control care trebuie să fie cuprins între ghilimele şi constă în indicatori de format numiţi şi **format flag**, aceştia indicând modul în care vor fi tipăriţi parametrii; 3. lista de parametri. Numărul indicatorilor de format (*format flag)* va determina câţi parametri vor fi specificaţi. Parametrii pot fi valori constante sau expresii, cu restricţia ca în lista de parametri să nu existe valori multicâmp sau expresii ce returnează astfel de valori. Valoarea returnată de funcţia *format* este şirul formatat. Dacă este folosit numele logic **nil**, atunci ieşirea nu tipăreşte nimic nici la terminal, nici într-un fişier, dar şirul formatat este returnat.

In exemplul următor funcţia *format* va fi folosită pentru a crea un şir fomatat în care numele persoanei are rezervate 15 spaţii, fiind urmat de vârsta persoanei. Utilitatea unei astfel de funcţii se observă mai ales atunci când se doreşte o afişare tabelată, pe coloane de date.

```
CLIPS> (format nil "Name: %-15s Age: %3d" "Lucian Blaga" 25)
"Name: Lucian Blaga   Age: 25"
CLIPS> (format nil "Name: %-15s Age: %3d" "Mihai Eminescu" 24)
"Name: Mihai Eminescu Age: 24"
```

Format flag începe cu **%**. Şirurile obişnuite cum ar fi "Name: " pot fi de asemeni puse în *control string* şi tipărite la ieşire. Unii indicatori de format nu sunt asociaţi parametrilor. De exemplu **%n** este folosit pentru a trimite la ieşire un cariage return / line feed, fiind echivalent cu **crlf** folosit de comanda *printout*. In exemplul nostru **%-15s** este folosit pentru tipărirea unui şir pe 15 caractere, unde semnul – semnifică faptul că afişarea se va face prin aliniere la stânga, iar caracterul **s** faptul că se va afişa un ***şir*** sau un ***simbol***. Indicatorul de format **%3d**

arată că se afişează prin aliniere la dreapta o valoare întreagă, pe o coloană de trei caractere. Dacă s-ar fi folosit de exemplu ca valoarea 5.25 ca parametru pentru vârstă, atunci s-ar fi tipărit valoarea 5 (prin conversie), deoarece partea fracționară nu este permisă într-un format întreg.

De observat că atunci când funcția *format* este folosită în partea dreaptă a unei reguli (RHS), valoarea sa este în general ignorată. In aceste cazuri numele logic folosit va fi sau **t** pentru a trimite output-ul la ecran sau un nume logic asociat cu un fişier. Specificarea generală a unui format este :

% - M . N x

caracterul - este opțional şi înseamnă aliniere la stânga (implicit aliniere la dreapta)

M specifică lățimea câmpului în coloane. Cel puțin M caractere vor fi scoase. Spațiile sunt în mod normal utilizate pentru a compartimenta ieşirea (se realizează M spații, dar dacă M începe cu un zero, în acest caz se afişează zerouri). Dacă ieşirea depăşeşte M coloane, atunci funcția format va extinde câmpurile atât cât e nevoie.

N este o specificare opțională a numărului de digiți peste virgulă, care vor fi tipăriți.(default = 6 digiți peste virgula zecimală pentru numere float)

x specificarea formatului display. Acesta poate fi:

d –întreg **f** -float	**s** -string **%** -caracterul % însuşi
e -exponențiala (în puterea a 10-a)	**g** -general (formatul cel mai scurt)
o -octal (număr fără semn)	**x** -hexazecimal (număr fără semn)
n -Cariage Return (CR)/ line feed	**r** -carriage return la ieşire (output)

4.9. Funcția READLINE

Funcția **readline** este folosită pentru citirea unei linii întregi la intrare, spre deosebire de funcția *read* care citea doar un singur câmp. Sintaxa acesteia este:

(readline [<logical-name>])

Numele logic este opțional, iar dacă nu se dă nici un nume logic atunci numele logic utilizat în mod implicit este **t** (intrarea va fi citită de la dispozitivul standard de intrare - tastatură). Funcția *readline* va returna următoarea linie de la sursa de intrare specificată cu numele logic, incluzând carriage return-ul. Dacă se citeşte dintr-un fişier şi s-a ajuns la sfârşitul acestuia va fi returnat simbolul EOF. Următorul exemplu ilustrează utilizarea funcției *readline*:

```
CLIPS> (clear)
CLIPS> (defrule introduce_numele
                =>
                (printout  t  "Care este numele tau? ")
                (bind  ?raspuns  (readline))
                (assert (nume-utilizator ?raspuns)) )
```

```
CLIPS> (reset)
CLIPS> (run)
Care este numele tau? Stefan cel Mare
CLIPS> (facts)
f-0     (initial-fact)
f-1     (nume-utilizator  “Stefan cel Mare”)
For a total of 2 facts.
```

Se observă faptul că în acest exemplu numele este stocat în faptul *nume-utilizator* ca un singur câmp. Din acest motiv nu este posibilă separarea primului nume de celelalte nume. Folosind funcţia **explode$** care acceptă un singur argument de tipul şir şi îl converteşte într-o valoare multicâmp, aceasta putând fi asertată ca o serie de câmpuri într-un fapt *nume-utilizator*.

```
CLIPS> (defrule introduce_numele
                =>
                (printout  t  “Care este numele tau? “)
                (bind  ?raspuns  (explode$  (readline)))
                (assert (nume-utilizator ?raspuns)) )
CLIPS> (reset)
CLIPS> (run)
Care este numele tau? Stefan cel Mare
CLIPS> (facts)
f-0     (initial-fact)
f-1     (nume-utilizator  Stefan cel Mare)
For a total of 2 facts.
```

4.10. Jocul grămezilor

Un joc simplu care se joacă în doi, va fi folosit ca exemplu pentru a demonstra diverse tehnici de control într-un limbaj bazat pe reguli. Iniţial, pe câmpul de joc se află mai multe grămezi de pietre. Alternativ, fiecare jucător îşi alege o grămadă formată din cel puţin două pietre şi o împarte în două grămezi mai mici. Mişcările se fac rând pe rând, până când toate grămezile conţin câte o singură piatră. Câştigă acela care face ultima mişcare.

Pentru simplitate vom considera că avem o singură grămadă şi că fiecare jucător extrage un număr de piese de maxim jumătate din câte mai sunt în grămadă, piese pe care le pune deoparte. Din grămada rămasă va extrage piese celălalt jucător şi tot aşa până când unul dintre cei doi va fi nevoit să extragă ultima piesă. Obiectivul jocului este de a evita faptul de a fi forţat să iei ultima ‘piatră’ din grămadă. Euristica (trucul) pentru a câştiga acest joc este găsită observând că poţi forţa adversarul să piardă dacă este rândul tău şi extragi un număr de piese, astfel încât numărul celor rămase în grămadă este de forma 2^n-1.

Astfel un jucător la a cărui mutare au rămas 2^n-1 pietre, a pierdut. In concluzie numărul de piese extrase ar trebui să fie de 3 sau 7 sau 15 … pietre. Dacă muţi primul şi grămada are unul dintre aceste numere “care pierd”, atunci nu poţi

câştiga decât dacă adversarul face o greşeală. Dacă muți primul şi grămada nu este constituită dintr-un număr care pierde, atunci poți întotdeauna să câştigi.

4.10.1. Tehnici de input

Înainte de a începe jocul, programul trebuie să determine unele informații. Programul va juca cu un partener uman, deci va trebui să determine cine mută primul. De asemenea trebuie determinată dimensiunea de pornire a grămezii de pietre. Această informație ar putea fi plasată într-o construcție *deffacts* , dar este simplu să se realizeze o interogare în legătură cu aceasta. Următorul exemplu arată cum poate fi utilizată funcția *read* pentru a introduce datele inițiale:

```
(deffacts date_initiale
        (faza alegere-jucator) )

(defrule alegere_jucator
        (faza alegere-jucator)
        =>
        (printout t t "Cine muta primul (Computer: c ; Human: h)? ")
        (assert (alegere-jucator (lowcase (read))) ))

(defrule alegere_jucator_corecta
        ?f1 <- (faza alegere-jucator)
        ?f2 <- (alegere-jucator ?jucator&c|h|C|H)
        =>
        (retract ?f1 ?f2)
        (assert (faza numar-piese)
                (mutare-jucator ?jucator)) )
```

Ambele reguli utilizează pattern-ul *(faza alegere-jucator)* pentru a indica aplicabilitatea lor doar când un anume fapt există în lista de fapte. Acest tip de pattern poartă numele de *control pattern* pentru că în mod deosebit este utilizat pentru a controla când este aplicabilă regula. ,deoarece control pattern-ul pentru aceste câmpuri conține doar câmpuri literale(fact de control trebuie să facă direct corespondența la pattern). Acest fapt de control va fi util pentru corecția erorii când intrarea primită de funcția *read* nu se potriveşte cu valorile aşteptate (c sau h pentru “computer” sau “human”).

Retractând şi reasertând faptul de control, regula *alegere_jucator* poate fi reaprinsă. Următorul rezultat la ieşire demonstrează cum regulile *alegere_jucator* şi *alegere_jucator_corectă* lucrează pentru a determina cine ar trebui să mute primul. Regulile precedente funcționează dacă sunt introduse răspunsuri corecte pentru c sau h, dar dacă se introduce un răspuns incorect, nu se relizează nici o verificare pentru erori. Sunt multe situații când se cere realizarea unei *bucle* de solicitare a unei intrări. Următoarea regulă utilizată cu regulile *alegere_jucator* şi *alegere_jucator_corectă* realizează verificarea unei erorii de intrare.

```
(defrule alegere_jucator_incorecta
        ?f1 <- (faza alegere-jucator)
        ?f2 <- (alegere-jucator ?jucator&~c|~h|~C|~H)
        =>
        (retract ?f1 ?f2)
        (assert (faza alegere-jucator))
        (printout t "Alege c sau h." crlf crlf) )
```

Se observă utilizarea *control-pattern*-ului *(faza alegere-jucator)*.Aceasta realizează controlul de bază pentru bucla de intrare şi evită "aprinderea" acestui grup de reguli în timpul altor faze de execuţie în program.

Cele două reguli *alegere_jucator* şi *alegere_jucator_incorectă* lucrează împreună, fiecare regulă furnizând faptele necesare activării celeilalte. Dacă răspunsul jucătorului la întrebarea este incorect, regula *alegere_jucator_incorectă* va retracta faptul de control *(faza alegere-jucator)* şi apoi îl va reaserta cauzând reactivarea regulii *alegere_jucator* şi astfel reintroducerea datelor.

4.10.2. Testarea informaţiilor

Anterior au fost discutate şi exemplificate elementele condiţionale *or, and, not, exists, forall, logical* şi *test*. Dintre acestea elementul condiţional **test** are o mare importanţă atunci când se efectuează teste asupra valorilor unor variabile sau se realizează repetarea unor calcule. In exemplul anterior se realizase o buclă din două reguli, care se repeta până când utilizatorul răspundea corect la o anumită întrebare. De multe ori însă bucla trebuie să se oprească ca rezultat al evoluţiei unei expresii arbitrare (de exemplu la apăsarea unei taste).

Funcţiile folosite de CE *test* sunt în marea lor majoritate funcţiile predicative ale CLIPS-ului (dacă acestea returnează în urma evaluării expresiei simbolul FALSE, atunci *CE test* nu este satisfăcut).Sintaxa pentru *CE test* este:

```
(test <predicate-function> )
```

unde funcţiile predicative pot fi luate din tabelul următor sau folosind Help-ul.

Să presupunem că este rândul jucătorului uman să extragă un număr de piese. Dacă a mai rămas o singură piatră în grămadă, atunci acesta a pierdut. Dacă numărul de piese este mai mare decât unu, atunci acesta trebuie întrebat câte pietre vrea să extragă din grămadă. Regula care realizează această interogare trebuie să verifice mai întâi dacă au rămas în grămadă mai mult de o piatră (s-ar putea utiliza funcţia predicativă > într-un CE *test*).

```
(test  (>  ?n  1))
```

După ce jucătorul uman a introdus numărul de piese pe care vrea să-l scoată din grămadă, acest răspuns trebuie verificat dacă este valid (numărul de piese trebuie să fie un întreg mai mare sau egal cu 1 şi mai mic sau egal cu jumătate din numărul de 'pietre' din grămadă). Toate aceste constrângeri pot fi puse într-un singur *test CE* folosind o funcţie predicativă *and*.

```
(test   (and   (integerp ?v)
               (>=  ?v  1)
               (<= ?v  (/ ?n 2)) ) )
```

Variabilele *?v* şi *?n* conţin numărul de 'pietre' care se doreşte a fi extras şi respectiv numărul de piese din grămadă. Similar dacă s-ar dori un test pentru un număr invalid vom nega expresia de mai sus (numărul să nu fie întreg, sau să fie mai mic decât 1, sau mai mare decât jumătate, sau mai mare egal cu numărul total).

```
(test    (or    (not  (integerp ?v))
               (<  ?v  1)
               (> ?v  (/ ?n 2)) ) )
```

Următoarele reguli se folosesc de precedentele elemente condiţionale *test* pentru a verifica validitatea numărului introdus de jucătorul uman. Faptul *numar-piese* folosit de aceste reguli stochează numărul de piese rămas în grămadă.

```
(defrule numar_de_piese
        (faza numar-piese)
        =>
        (printout t t "Introduceti numarul de piese: ")
        (assert (numar-piese (read))) )

(defrule numar_de_piese_incorect
        ?f1 <- (faza numar-piese)
        ?f2 <- (numar-piese ?n)
        (test  (or  (not (integerp ?n))  (< ?n 2)  (> ?n 100)  ))
        =>
        (retract ?f1 ?f2)
        (printout t "Alege un numar intreg intre 2 si 100" t t)
        (assert (faza numar-piese)) )

(defrule numar_de_piese_corect
        f1 <- (faza numar-piese)
        (numar-piese ?n)
        (test  (and  (integerp ?n)  (>= ?n 2)  (<= ?n 100)  ))
        =>
        (retract ?f1) )
```

Dacă până acum s-a verificat dacă numărul de piese se află în intervalul admis (2 100). Odată introdus acest număr, jocul poate începe, fiecare jucător extrăgând din grămadă un număr de pietre. Din acest moment devine necesară verificarea numărului de piese pe care le ia jucătorul uman, de aceasta ocupându-se următoarele trei reguli (nu este necesară şi verificarea mutării calculatorului).

```
(defrule mutare_jucator_uman
        (mutare-jucator h)
        (numar-piese ?n)
        (test (> ?n 1))              ;verifica daca au mai ramas piese de extras
        =>
        (printout t crlf "Cite piese vrei sa iei? ")
        (assert (omul-ia (read))))
```

```
(defrule mutare_om_corecta
        ?f1 <- (mutare-jucator h)
        ?f2 <- (numar-piese ?n)
        ?f3 <- (omul-ia ?o)
        (test  (and  (integerp ?o)  (>= ?o 1)  (<= ?o (/ ?n 2))  ))
        =>
        (retract ?f1 ?f2 ?f3)
        (assert (numar-piese (- ?n ?o))
                (mutare-jucator c))
        (printout t "Au mai ramas " (- ?n ?o) " piese." crlf))

(defrule mutare_om_incorecta
        ?f1 <- (mutare-jucator h)
        (numar-piese ?n)
        ?f3 <- (omul-ia ?o)
        (test (or (not (integerp ?o))  (< ?o 1)  (> ?o 3)  (> ?o (/ ?n 2))  ))
        =>
        (retract ?f1 ?f3)
        (printout t "Introduceti un intreg [1-" (integer (/ ?n 2)) "]." crlf)
        (assert (mutare-jucator h)) )
```

Acest program realizează încă două bucle, fiind utilizate fapte de control pentru a reaprinde regula de introducere a valorii, în cazul unui răspuns incorect. Regula *mutare_om_incorecta* şi *numar_de_piese_incorect* va reaserta faptul de control *(mutare-jucator h)* sau *(faza numar-piese)* pentru a reactiva regula *mutare_jucator_uman* sau *numar_de_piese* (jucătorul uman va reintroduce astfel numărul de piese de extras). După cum s-a văzut strategia jocului constă în eliminarea unui număr de piese astfel încât numărul celor rămase în grămadă să fie de forma 2^n-1. Regula *calculeaza_valori_cheie* furnizează un set de astfel de valori, care să ajute calculatorul să câştige. Singurul caz defavorabil pentru calculator este cel în care numărul de piese în momentul când acesta mută este deja 2^n-1.

```
(defrule calculeaza_valori_cheie
        (numar-piese ?n)
        (putere ?v)
        (test (<= (- (* 2 ?v) 1) ?n))
        =>
        (assert (valoare (- (* 2 ?v) 1) )
                (putere (* ?v 2)) ))

(defrule mutare_calculator
        ?f1 <- (mutare-jucator c)
        ?f2 <- (numar-piese ?n&:(> ?n 1))
        (valoare ?v&:(< ?v ?n))
        (test (<= (- ?n ?v) (/ ?n 2)) )
        =>
        (retract ?f1 ?f2)
        (printout t t "Calculatorul ia " (- ?n ?v) " piese." crlf)
        (printout t "Au mai ramas " ?v " piese." crlf)
        (assert (numar-piese ?v)
                (mutare-jucator h) ))
```

```
(defrule mutare_fortata_calculator
        ?f1 <- (mutare-jucator c)
        ?f2 <- (numar-piese ?n&:(> ?n 1))
        (valoare ?n)
        =>
        (retract ?f1 ?f2)
        (bind ?v (+ 1 (mod (random) (integer (/ ?n 2))) ))
        (printout t t "Calculatorul ia " ?v " piese." crlf)
        (printout t "Au mai ramas " (- ?n ?v) " piese." crlf)
        (assert (numar-piese (- ?n ?v))
                (mutare-jucator h) ))
```

Regula *mutare_fortata_calculator* este realizată pentru cazul în care numărul pieselor din grămadă este de 2^n-1 şi când omul are şanse de câştig. In acest caz calculatorul va extrage un număr aleator *(+ 1(mod (random) (integer (/ ?n 2))))* (în intervalul [1,n/2]), aşteptând o greşeală din partea jucătorului uman. In regulile următoare dacă numărul pieselor rămase în grămadă este egal cu 1, înseamnă că jocul s-a terminat şi ca unul dintre jucători trebuie desemnat învingător.

```
(defrule omul_pierde
        (numar-piese 1)
        (mutare-jucator h)
        =>
        (printout t crlf "Trebuie sa iei ultima piesa !" crlf)
        (printout t "AI PIERDUT !!!" crlf))

(defrule omul-castiga
        (numar-piese 1)
        (mutare-jucator c)
        =>
        (printout t crlf "Calculatorul trebuie sa ia ultima piesa !" crlf)
        (printout t "AI INVINS !!!" crlf))
```

Reluarea jocului este realizată în ultimele două reguli dacă răspunsul este afirmativ. După cum se poate vedea regula *play-again* acceptă mai multe variante de răspunsuri *(y |Y |yes |Yes |YES |d |D |Da |da |DA)*, în acest fel utilizatorul nefiind restricționat la o singură variantă de răspuns. In RHS-ul regulii sunt introduse pentru a restarta programul cele două comenzi clasice *(reset) (run)*. Acest lucru putea fi realizat şi reasertând faptele inițiale (regulile s-ar fi aprins din nou).

```
(defrule reluare
        (numar-piese 1)
        =>
        (printout t crlf "Vrei sa mai joci odata (y/n) ?")
        (assert (play (read))))

(defrule play-again
        (play ?c&y |Y |yes |Yes |YES |d |D |Da |da |DA)
        =>
        (reset)
        (run))
```

4.10.3. Câmpul condiţional predicativ

Spre deosebire de celelalte câmpuri condiţionale *and, or, not* (notate cu &, | , ~) care erau legate de o variabilă, câmpul condiţional predicativ este legat de o funcţie predicativă şi se notează cu **:** ,fiind util atunci când se doreşte efectuarea testelor predicative direct în pattern-uri. In multe privinţe este similar cu un element condiţional *test* aflat imediat după un pattern, dar se va vedea mai târziu că uneori este mai eficientă utilizarea unui câmp condiţional predicativ, decât a unui *test CE*. Câmpul condiţional predicativ poate fi utilizat la fel ca orice câmp condiţional literal. Poate fi o condiţie de sine stătătoare într-un câmp sau poate fi o parte a unui câmp mai complex utilizând câmpurile condiţionale conective ~, &, |. Câmpul condiţional predicativ este întotdeauna urmat de o funcţie de evaluare. Ca şi în cazul lui *test CE* aceasta poate fi o funcţie predicativă. De exemplu în regula anterioară *mutare_jucator_uman* se foloseau două pattern-uri pentru a verifica dacă grămada conţine mai mult de o piesă, care pot fi compactate într-un singur pattern.

```
(numar-piese ?n)                    (numar-piese ?n&:(> ?n 1) )
(test (> ?n 1))
```

Există câteva situaţii în care se recomandă folosirea câmpului condiţional predicativ. Una ar fi cea de mai sus unde imediat după ce a fost asociată o valoare unei variabile se verifică o condiţie. S-ar putea citi un astfel de pattern în felul următor "leagă o valoare de variabila *?size* dacă această valoare îndeplineşte condiţia". Regulă următoare verifică dacă se adăuga la un total o dată numerică.

```
(defrule adauga_la_suma
        (data ?val&:(numberp ?val))
        ?f <- (total ?total)
        =>
        (retract ?f)
        (assert (total (+ ?total ?val))) )
```

Următoarele două reguli realizează tot astfel de verificări de erori însă se vor folosi câmpurile condiţionale conective ~ şi | pentru a asocia mai multe câmpuri condiţionale predicative. Regula *gaseste_data_tip_1* verifică dacă o dată este un şir sau un simbol, utilizând funcţiile predicative *stringp* şi *symbolp*. Regula *gaseste_data_tip_2* verifică dacă o dată nu este un întreg folosind funcţia predicativă *integerp* ,asociată cu un câmp conectiv ~ pentru a nega o valoare.

```
(defrule gaseste_data_tip_1
        (data ?item&:(stringp ?item) | :(symbolp ?item))
        =>
        (printout  t  ?item  " este şir sau simbol." crlf) )

(defrule gaseste_data_tip_2
        (data ?item&~:(integerp ?item))
        =>
        (printout  t  ?item " nu este un intreg."  crlf) )
```

4.10.4. Câmpul condiţional = , valoare returnată

Câmpul condiţional valoare returnată, = (the return value field constraint), permite ca valoarea returnată a unei funcţii să fie utilizată pentru comparaţia în pattern. Câmpul condiţional valoare returnată poate fi utilizată în conjuncţie cu conectivele & , | , ~ ca şi câmpul condiţional predicativ. De asemenea acest câmp condiţional trebuie urmat întotdeauna de o funcţie (şi de obicei această funcţie nu trebuie să fie predicativă, excepţia fiind atunci când dorim compararea valorii cu simbolurile TRUE sau FALSE). Singura restricţie este că această funcţie să aibă o valoare returnată de un singur câmp.

Să presupunem că avem doi jucători care dau cu zarul şi obţin un anumit punctaj pentru fiecare zar. Pentru a implementa această problemă putem construi un deftemplate zar, care să conţină sloturile valoare şi punctaj.

```
(deftemplate zar
      (slot punctaj)
      (slot valoare))

(deffacts punctaje-zaruri
      (zar (punctaj 0) (valoare 1))
      (zar (punctaj 1) (valoare 2))
      (zar (punctaj 2) (valoare 3))
      (zar (punctaj 4) (valoare 4))
      (zar (punctaj 8) (valoare 5))
      (zar (punctaj 10) (valoare 6)) )

(defrule simulare-aruncare
      (zar (punctaj  ?p) (valoare   =(+ (mod (random) 6) 1) )
      =>
      (printout t " Ai câştigat  " ?p " puncte la aceasta aruncare a zarului" t))
```

Regula *simulare-aruncare* determină numărul corespunzător de puncte pe care trebuie să le ia un jucător pentru o aruncare a zarului. Funcţia **mod** (provenită de la *modulus*) returnează întregul rămas ca urmare a divizării primului argument prin cel de al doilea (restul împărţirii).Când citeşti un pattern care are un câmp condiţional valoare returnată este util să ne gândim la acesta ca la un “este egal cu”.

```
=(mod ?n 6)
```

poate fi citit “câmpul este egal cu ?n modulo 6”. Funcţia **random** returnează un număr aleator, căruia dacă i se aplică operaţia de modulo 6 se obţin resturile împărţirii unui număr la 6 (0-5), la care ar trebui adăugat 1.

De reţinut faptul că = câmp condiţional poate fi folosit doar în interiorul unui pattern şi nu este acelaşi cu funcţia predicativă =, care poate fi utilizată în interiorul unui câmp condiţional predicativ, *test CE*, partea de acţiune (RHS) a unei reguli, sau în introducerea datelor la top level (prompter). Putem utiliza câmpurile conective ~, & , | între câmpurile condiţionale = pentru a realiza un câmp condiţional mai complex.

```
(valoare =(mod ?n 4) &~ =(mod ?n 3))
```

4.11. Comanda SYSTEM

Comanda **system** permite execuția comenzilor sistemului de operare:

(system <lexeme-expresion>+)

De exemplu, următoarea regulă va da o listă a directoarelor şi fişierelor pentru un director al cărui nume este specificat într-un fapt, pe o maşină care utilizează sistemul de operare UNIX.

```
(defrule list-directory-UNIX
        (list-directory ?directory)
        =>
        (system “ls “ ?directory))
```

In acest exemplu, primul argument al comenzii *system* ,“ls “, este comanda UNIX pentru listing-ul unui director. De reținut faptul că în cazul nostru este obligatorie existența spațiului după caracterele “ls”.

Comanda *system* pune cap la cap toate argumentele (pot fi şiruri sau simboluri) pentru a forma un singur şir înainte de ai permite sistemului de operare să proceseze comanda. Toate spațiile necesare trebuie incluse ca parte a unui argument (sau chiar ca argument) în comanda *system*. Efectele acestei comenzi variază de la un sistem de operare la altul. Nu toate sistemele de operare prevăd funcționalitatea unei implementări a comenzii *system*, de aceea nu ne putem bizui pe faptul că această comandă va fi valabilă în CLIPS.

Această comandă nu returnează o valoare, deci nu este posibilă returnarea unei valori în CLIPS după execuția unei comenzi a sistemului de operare. Pentru a observa funcționarea acestei comenzi sub sistemul de operare DOS se va rula **CLIPS386.exe** (varianta CLIPS-ului pentru DOS). Regula de mai sus ar fi:

```
(defrule list-directory-DOS
        (list-directory ?directory)
        =>
        (system “dir “ ?directory))
```

Se pot da oricare din comenzile sistemului de operare:

```
CLIPS> (system “edit”)
CLIPS> (system “format a:”
CLIPS> (system “copy c:\\a.dat  d:”)
CLIPS> (system “dir “ c:)                ;argumentele pot fi şi simboluri
```

4.12. Comanda BATCH

Această comandă permite introducerea comenzilor şi răspunsurilor care ar fi trebuit să fie introduse în mod normal la nivelul prompter-ului (top level) să fie scrise direct într-un fişier. Sintaxa comenzii **batch** este următoarea:

(batch <file-name>)

De exemplu, presupunând dialogul arătat la rularea programului baghetelor, se pot introduce toate comenzile şi răspunsurile într-un fişier "gramezi.**bat**" (extensia fişierului nu trebuie să fie neapărat ".bat", aceasta fiind folosită prin convenție pentru a arăta că fişierul respectiv este executabil),care mai apoi poate fi rulat cu comanda **batch**. Avantajele acestei comenzi se observă mai ales în faza de corectare a erorilor programului, când utilizatorul nu mai este nevoit să introducă de fiecare dată datele de intrare. Fişierul nostru ar avea stocată în el următoarea informație:

```
(clear) ↵
(load "gramezi.clp") ↵
(reset) ↵
(run) ↵
c↵
18↵
3 2 1 1↵
```

Odată ce toate comenzile şi răspunsurile au fost citite în fişierul *batch* , interacțiunea cu tastatura la nivelul promter-ului revine la normal. Când se rulează pe un sistem de operare care suportă argumente în linia de comandă (cum ar fi UNIX) , atunci CLIPS-ul poate executa automat la startare comenzile dintr-un fişier batch. Sintaxa pentru execuția unui fişier *batch* la startare este:

clips –f <file-name>

Folosirea opțiunii –f este similară cu introducerea comenzii *(batch <file-name>)* imediat după startarea CLIPS-ului.

4.13. Comanda DRIBBLE-ON şi DRIBBLE-OFF

Comanda **dribble-on** poate fi folosită pentru a stoca o înregistrare pentru toate ieşirile de pe terminal şi toate intrările de la tastatură. Sintaxa acesteia este:

(dribble-on <file-name>)

Odată ce comanda *dribble-on* a fost executată, toate ieşirile trimise spre terminal şi toate intrările de la tastatura vor fi scrise în fişierul <file-name> întocmai ca pe ecran (terminal). Pentru a stopa efectul comenzii *dribble-on* se foloseşte comanda *dribble-off* a cărei sintaxă este următoarea:

(dribble-off)

4.14. Intrebări

1. Care dintre formatele următoare: *infix, postfix* şi *prefix* este cel folosit în scrierea expresiilor numerice din CLIPS ?

2. Cum trebuiesc scrise expresiile următoare pentru a putea fi calculate de CLIPS şi care sunt formele lor minimale ?

   12 * 5 + 1.2 – 3**2 / 8

   (5 * (5 + 6 + 7) – ((3 * (4 / 9) + 2) / 8)
3. Pentru ce este folosită funcţia **bind** ? Când este ste utilă folosirea acesteia ?
4. Poate fi utilizată funcţia **bind** în LHS-ul regulii ? De ce?
5. Care este sintaxa funcţiilor **read** şi **open** şi semnificaţia parametrilor acestora ?
6. Ce returnează funcţiile **open** şi **close**?
7. Care sunt diferenţele între funcţiile **read** şi **readline**? Dar între **printout** şi **format** ?
8. Ce reprezintă parametrii **%-M .N *x*** în cadrul funcţiei format ?
9. Care este simbolul utilizat pentru a semnala faptul că avem asociat un câmp condiţional predicativ unei variabile ?
10. Câmpul condiţional valoare returnată poate fi legat de o variabilă ? Care este simbolul utilizat pentru reprezentarea acestuia ?
11. Care sunt avantajele folosirii comenzii **batch** ?

4.15. Aplicaţii

1. Scrieţi un program în CLIPS care să ceară utilizatorului să introducă o culoare şi care apoi să afişeze toate ţările a căror steag conţine acea culoare. In cazul în care se introduc de la tastatură mai multe culori numărul ţarilor a căror steag conţin acele culori va fi din ce în ce mai mic. Se dă lista următoare de ţari, care poate fi eventual mărită:

Ţara	Culori steag
U.S.A.	Roşu, alb şi albastru
Belgia	Negru , galben şi roşu
Polonia	Alb şi roşu
Monaco	Alb şi roşu
Suedia	Galben şi albastru
Panama	Roşu, alb şi albastru
Jamaica	Negru, galben şi verde
Columbia	Galben, albastru şi roşu
Italia	Verde, alb şi roşu
Irlanda	Verde, alb şi portocaliu
Grecia	Albastru şi alb
Botswana	Albastru, alb şi negru
România	Roşu, galben şi albastru

2. Realizaţi un program în citeşte dintr-un fişier lista rezultatelor candidaţilor înscrişi la facultatea de Automatică şi Calculatoare. Pentru fiecare din candidaţi

alături de nume sunt introduse media la bacalaureat şi cea de la examenul de admitere. Să se afişeze într-un alt fişier lista candidaţilor admişi (dacă numărul de locuri este mai mic decât cel al candidaţilor înscrişi baza de date trebuie trunchiată) în ordinea mediilor totale. Media totală se calculează după formula:

Media_totală=0.3*Media_bacalaureat+0.7*nota_test

Intr-un alt fişier se pot reprezenta candidaţii respinşi, ordonaţi alfabetic. Pentru aceasta se poate folosi funcţia de comparare a şirurilor **str-compare**.

3. In jocul de Bridge fiecare jucător primeşte un număr de 13 cărţi şi în funcţie de acestea va licita un număr de perechi pe care acesta crede că le va realiza. Considerând că asul valorează 4 puncte, popa 3 puncte, dama 2 puncte, valetul 1 punct şi că restul cărţilor pot fi neglijate la licitaţie, realizaţi un program care să evalueze mâna dumneavoastră şi să afişeze valoarea totala a cărţilor.
4. Modificaţi problema baghetelor astfel încât la sfârşit, utilizatorul să fie întrebat dacă mai vrea sau nu să joace încă o dată. In caz afirmativ problema va fi relansată în execuţie (cu ajutorul comenzii **reset**).
5. Să se modifice problema baghetelor să se permită pe lângă jocul între om şi calculator şi jocul între 2 oameni.
6. Să se calculeze minimul şi maximul dintre numerele citite dintr-un fişier. Se poate verifica eventual dacă ceea ce s-a citit este sau nu este un număr (caracterele permise sunt '-','+' pentru semn, '.' pentru partea zecimală şi cifrele de la 0 la 9). Dacă se doreşte se poate folosi direct funcţia predicativă **numberp** (care returnează TRUE dacă argumentul său este un număr şi FALSE în caz contrar) sau funcţiile **integerp** şi **floatp** (dacă se doreşte a se afla exact dacă numărul este un întreg sau un flotant).

4.16. Probleme rezolvate

1. **Problema numărului lipsă** - Se citesc date dintr-un fişier în care avem un număr de numere întregi pozitive şi distincte, situate într-un interval [1..N]. Dintre toate aceste numere lipseşte unul singur şi se cere aflarea numărului lipsă. Problema va fi realizată ţinându-se cont în special de faptul că numerele sunt distincte şi că suma numerelor de la 1 la N este N*(N+1)/2. Chiar dacă acest număr N nu ar fi specificat este suficientă numărarea tuturor numerelor din fişier, la acesta adăugând numărul lipsă pentru al obţine pe N. De exemplu pentru: 3 5 7 8 2 1 9 6 , avem N=9 şi numărul lipsă 4.

```
(deffacts initial
        (suma 0)
        (idx 1) )                    ;contorizeaza numerele citite

(defrule deschide_fisier
        (declare (salience 1))
        =>
        (open "numar.dat"  ex) )
```

```
(defrule inchide_fisier
        (numar EOF)
        =>
        (close ex) )

(defrule citeste_un_numar
        (not (exists (numar EOF)))
        =>
        (bind ?val (read ex))
        (assert (numar ?val))
        (refresh citeste_un_numar) )

(defrule aduna_la_suma
        ?f1 <- (suma ?s)
        ?f2 <- (idx ?i)
        ?f3 <- (numar ?numar&~EOF)
        =>
        (retract ?f1 ?f2 ?f3)
        (assert (suma (+ ?s ?numar ))
                (idx (+ ?i 1)) )  )

(defrule afisare_numar_lipsa
        (declare (salience -1))
        (idx ?i)
        (suma ?suma)
        =>
        (bind  ?no   (- (/ (* ?i (+ ?i 1) ) 2) ?suma) )
        (printout t "Numarul lipsa este "  ?no crlf) )
```

Pentru a evita folosirea unor fapte de tip flag, care ar marca trecerea de la o etapă a programului la o alta şi care ar trebui să fie şterse şi reasertate de fiecare dată când este nevoie, se recomandă folosirea lui **salience**, cu ajutorul căruia se pot organiza regulile pe nivele de prioritate. Se observă folosirea funcției **refresh**, care forţează o regulă să se aprindă de mai multe ori. Fără această funcţie regula *citeste_ un_numar* s-ar aprinde o singură dată (s-ar citi numai primul număr din fişier).

De reținut faptul că funcția **read** returnează **EOF** la întâlnirea sfârşitului de fişier. In LHS-ul regulii *aduna_la_suma* se verifică dacă numărul este diferit de EOF, deoarece operația de adunare se efectuează numai asupra numerelor, în cazul în care parametrii acesteia sunt simboluri sau şiruri CLIPS-ul generând un mesaj de eroare. Regula *citeşte_un_numar* poate fi scrisă mai compact, eliminând funcția bind.

```
(defrule citeste_un_numar
        (not (exists (numar EOF)))
        =>
        (assert (numar (read ex)))
        (refresh citeste_un_numar) )
```

2. **Problema automatului** – Fie un automat cu un număr de N (N<100) celule aşezate în cerc, fiecare celulă comunicând cu cei doi vecini ai săi. Fiecare celulă se poate găsi în două stări : excitată sau liniştită, şi poate lua valorile 0 sau 1, (1 – valoarea corespunzătoare stării de excitaţie a unei celule, 0 – semnifică lipsa excitaţiei). O celulă excitată la un moment T va emite un semnal care ajunge la cele două celule vecine cu ea în momentul T+1.

   O celulă este excitată atunci şi numai atunci când la ea ajunge un semnal de la una din celulele vecine; dacă semnalele ajung deodată din ambele părţi, atunci ele se anulează şi celula nu se mai excită. Fiind dată o configuraţie iniţială de celule, să se determine dacă excitaţia se menţine indefinit sau se va linişti. Dacă se ajunge într-o stare a automatului care se repetă, înseamnă că suntem într-o buclă care se va repeta infinit. Problema se poate modifica în sensul că se poate încerca scoaterea din buclă prin introducerea la anumite momente în celule neexcitate a stării de excitare.

```
(deffacts nume
        (stare 1 0 1 0 1 1 0 1 1)
        (stare_curenta 1)
        (new_list )        )

(defrule aprinde1
        (declare (salience 3))
        (stare  ?c  ?d  $?  ?s)
        =>
        (assert (new_data (mod(+ ?s ?d)2))) )

(defrule aprinde2
        (declare (salience 2))
        (stare  $?  ?s  ?c  ?d  $?)
        =>
        (assert (new_data (mod(+ ?s ?d)2))) )

(defrule aprinde3
        (declare (salience 1))
        (stare  ?d  $?  ?s  ?c)
        =>
        (assert (new_data (mod(+ ?s ?d)2))) )
```

S-au construit trei reguli de ‘aprindere’ (excitare) a unei celule, deoarece dintre cele N celule aşezate în cerc, exceptând prima şi ultima celulă, doar elementele centrale din listă vor avea vecinul din dreapta şi pe cel din stânga în imediata lor vecinătate *(stare $? ?s ?c ?d $?)*.

Pentru primul element din listă vecinul din stânga va corespunde ultimului element şi cel din dreapta celui de-al doilea element *(stare ?c ?d $? ?s)* ,iar pentru ultima celulă vecinul din dreapta va fi prima celulă şi cel din stânga penultima celulă *(stare ?d $? ?s ?c)*. Priorităţile sunt puse pentru a fi siguri că se va începe cu prima celulă şi se va termina cu ultima.

```
(defrule new_data
        ?f1 <- (new_data ?v)
        ?f2 <- (new_list $?l)
        =>
        (retract ?f1 ?f2)
        (declare (salience 10))
        (assert (new_list $?l ?v)) )

(defrule stare_noua
        (declare (salience -10))
        ?f1 <- (stare_curenta ?n)
        ?f2 <- (new_list $?v)
        (test (neq (length$ $?v) 0))
        =>
        (assert  (stare $?v)
                 (new_list)
                 (stare_curenta   (+ ?n 1 )))
        (retract ?f1 ?f2 ) )
```

Regula *new_data* va lua fiecare nouă stare asertată de regulile de aprindere şi o va adăuga la o nouă listă (iniţial vidă). După ce toate stările celor N celule au fost introduse rând pe rând în lista nouă (se poate urmări cu **(run 1)**, pas cu pas) se va aserta o nouă stare şi se va crea o listă vidă pentru starea următoare.

De reţinut este faptul că în mod normal (implicit) CLIPS-ul nu acceptă două fapte identice în lista de fapte şi că în momentul în care se ajunge într-o stare a automatului, prin care acesta a mai trecut anterior, noua stare nu mai este asertată deoarece ea există deja în lista de fapte. Pentru a modifica această variabilă de stare se poate folosi funcţia **set-fact-duplication** (sau se poate selecta din meniul Execution, în Options… configuraţia pe care o dorim). În cazul în care CLIPS-ul acceptă duplicate programul va rula infinit dacă excitaţia se menţine.

Dacă ultima stare a automatului nu a fost cea în care toate celulele sunt neexcitate **0,0,0…0** , atunci aceasta este ultima stare înainte ca automatul să intre în buclă. Valoarea faptului *stare_curenta* va da valoarea stării în care se intră în buclă. Dacă se dau la intrare stările celulelor automatului *(stare 1 0 1 0 1 0), (stare 1 1 1 1 1)* sau *(stare 1 0 1 1 0 1 0 1)*, se obţine un automat în care excitaţia nu se menţine.

```
(defrule automat_finit
        (declare (salience -5))
        (new_list $?v&:(not(member$ 1 $?v))
                       &:(neq (length$ $?v) 0))
        =>
        (printout t "Excitatia nu se mentine infinit automat." t) )

(defrule automat_infinit
        (declare (salience -5))
        (new_list $?v&:(member$ 1 $?v))
        (stare $?v)
        =>
        (printout t "Automatul are un numar nedefinit de stari. t) )
```

Cele două reguli introduse semnalează intrarea automatului într-o buclă sau stingerea excitației (corespunzătoare stării 0). Dacă se doreşte acestea pot fi modificate pentru a se afişa şi numărul de stări după care se produce unul din cele două evenimente, prin simpla afişare a valorii faptului *stare_curenta*.

Vom modifica modul de introducere a faptelor în această problema, astfel încât acestea să fie citite dintr-un fişier sau de la tastatură. Dacă se doreşte citirea de la tastatură cel mai indicat mod de citire ar fi cu ajutorul funcției **readline**, eliminându-se astfel o eventuală buclă în care s-ar introduce valoarea fiecărei stări caracter cu caracter (0 sau 1).

```
(defrule citeste_de_la_tatstatura
        =>
        (printout t "Introduceti starile automatului: ")
        (assert (data (explode$ (readline)))) )

(defrule data_invalida
        (or ?f <- (data  $?  ~0&~1  $?)
            ?f <- (data $?v&: (< (length$ $?v) 3)) )
        =>
        (retract ?f)
        (printout  t  "Introduceti valori de 1 sau  0 separate prin spatii." t)
        (refresh citeste_de_la_tatstatura) )

(defrule data_valida
        (declare (salience –5))
        ?f <- (data  $?v)
        =>
        (retract ?f)
        (assert (stare $?v)) )
```

In regula *citeste_de_la_tatstatura* se transforma data intrare de tip şir de caractere returnată de funcția *readline*, într-un multifield folosindu-se funcția **explode$**. Regula *data_invalida* verifică dacă există vre-un element care nu este 0 sau 1, sau dacă lungimea şirului de intrare este mai mică decât 3 (o celulă trebuie să aibă doi vecini, deci dimensiunea minimă a automatului este de minim 3 celule). Regula *data_valida* având prioritate mai mică decât cea în care se verifică corectitudinea datelor de intrare, primeşte un listă de elemente care au fost verificate şi poate aserta starea inițială a automatului.

Dacă datele se citesc dintr-un fişier, atunci se mai adaugă două reguli, una pentru deschiderea fişierului şi una pentru închiderea acestuia. In cazul în care datele de intrare sunt incorecte nu se mai foloseşte funcția de reîmprospătare **referesh**. Se recomanda de asemenea citirea unei linii întregi cu funcția **readline**.

```
(defrule deschide_fisier
        =>
        (open "date.txt"  ex) )
(defrule inchide_fisier
        (declare (salience -5))
        =>
(close ex) )
```

```
(defrule citeste_din_fisier
        =>
        (assert (data (explode$ (readline ex)))) )
```

Dacă se presupune că datele de intrare nu sunt pe aceiaşi linie atunci în locul funcției *readline*, trebuie folosită funcția **read**.