CAP.1 PREZENTARE GENERALĂ

1.1. Intrarea şi ieşirea în CLIPS

Pe calculatorul pe care s-a instalat CLIPS-ul se va lansa în execuţie executabilul corespunzător sistemului de operare în care rulează programul. De exemplu pentru Windows avem *clipswin.exe,* pentru DOS *clips386.exe*, etc. Dacă CLIPS-ul a fost instalat corect va apare pe ecran prompter-ul:

CLIPS>

Din acest moment utilizatorul poate introduce comenzi, acestea fiind executate direct de CLIPS. Acest mod de a lansa direct comenzi mai este denumit şi **top level**. Modul normal de ieşire din CLIPS este comanda:

(exit)

dar mai există uneori şi alte posibilităţi (de exemplu sub Windows se poate închide fereastra sau se poate folosi scurtătura **CTRL-Q**, corespunzătoare comenzii ***Quit CLIPS*** din meniul ***File***).

De reţinut este faptul ca în limbajul CLIPS se folosesc o serie de delimitatori cum ar fi **<>** , **[ ]** , **{ }** , **( )** şi că aceştia trebuiesc folosiţi cu atenţie, (numărul parantezelor deschise să coincidă cu cel al parantezelor închise). Deşi s-ar putea observa o asemănare cu C-ul unde, la fel, ‘ **{** ‘ marchează începutul unui bloc, iar ‘ **}** ‘ marchează sfârşitul unui bloc, totuşi aceste paranteze sunt moştenite de CLIPS de la maşinile LISP pe care a fost iniţial dezvoltat şi unde parantezele erau, de asemeni, folosite ca delimitatori.

Diferenţa între **(exit)** cu paranteze şi **exit** fără paranteze este că în CLIPS, primul va fi identificat cu o comandă, iar al doilea va fi asociat cu un simbol. Convenţii de notaţie pentru înţelegerea sintaxei CLIPS: parantezele [ ] închid între ele un element opţional; < > reprezintă necesitatea atribuirii unor valori efective de tipul specificat între aceste semne; caracterul ‘ | ’ exprimă operatorul “sau”; simbolul ‘ $^{+}$ ’ reprezintă faptul că vom avea cel puţin una sau mai multe valori care să înlocuiască elementul descris sintactic; simbolul ‘ * ’ reprezintă zero sau mai multe valori care pot înlocui expresia. Astfel, expresia <integer>$^{+}$ este echivalentă cu <integer><integer>*.

ELEMENTE PRIMARE DE PROGRAMARE

In CLIPS avem trei elemente primare de scriere a programelor: tipuri primitive de date, funcţii pentru manipularea datelor şi constructori pentru adăugarea datelor la baza de cunoştinţe.

1.2. Tipuri de date

In CLIPS există următoarele tipuri de date primitive: **float**, **integer**, **symbol**, **string**, **external address**, **fact address**, **instance name** şi **instance address**. Informațiile de tip numeric pot fi reprezentate, evident, prin întregi sau prin tipul virgulă flotantă iar informațiile de tip simbolic pot fi reprezentate prin simboluri sau şiruri de caractere.

Primele două tipuri de câmpuri, **float** şi **integer**, sunt numite câmpuri numerice sau numere. Un câmp numeric este alcătuit din trei părți: semnul, valoarea şi exponentul. Semnul (+ sau -) poate fi opțional. Valoarea conține una sau mai multe cifre (0÷9), un punct opțional (.) şi din nou una sau mai multe cifre. Exponentul este şi el opțional şi constă în litera e sau E, urmată opțional de semn şi de una sau mai multe cifre. Orice număr format doar din semnul opțional şi din cifre este considerat a fi de tip întreg. Toate celelalte numere sunt de tip float.

Exemple de întregi :	1	+7	-2	79
Exemple de float:	1.3	3.0	8e-1	6.5E+3

In concluzie un întreg foloseşte următorul format:

<integer> ::= [+| -] <digit>+
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Un număr în virgulă mobilă foloseşte formatul următor:

<float> ::= <integer> <exponent> |
<integer> **.** [exponent] |
. <unsigned integer> [exponent] |
<integer > **.** <unsigned integer> [exponent]

<exponent> ::= e|E <integer>
<unsigned-integer> ::= <digit>+

Un **symbol** este un câmp ce începe cu un caracter ASCII printabil şi este urmat de unul sau mai multe caractere. Sfârşitul unui simbol este marcat de un **delimitator** şi anume: orice caracter ASCII neprintabil (**spațiu**, **tab**, **CR** – Enter, **LF** – Line feeds), ghilimelele **“** , paranteza deschisă **(** , paranteza închisă **)** , punct şi virgulă ‘ **;** ’, ampersand-ul ‘**&**’, bara verticala ‘ | ’, tilda ‘ ~ ’, caracterul mai mic decât ‘<’. Un simbol nu poate conține un delimitator (singura excepție este caracterul ‘<’ care poate fi doar primul caracter din simbol). De asemenea **?** şi **$?** sunt secvențe de caractere ce nu pot să apară la începutul unui simbol deoarece acestea sunt folosite pentru a marca variabilele. Orice secvență de caractere care nu este un câmp numeric este tratată ca un simbol. Exemple de simboluri:

data-an viata	Mihai	nr_prim
!#$%?$?	1234A	091-345-67

Se observă folosirea underscore-ului şi a liniuței de unire pentru utilizarea mai multor simboluri într-un singur câmp. CLIPS este **case-sensitive** şi deci face deosebire între literele mari şi cele mici, din acest motiv următoarele simboluri sunt considerate ca fiind diferite:

nr-prim Nr-prim NR-PRIM nr-Prim .

Următorul tip de câmpuri este **string** (şir de caractere). Un şir trebuie să înceapă şi să se termine cu ghilimele. Trebuie reținut faptul că ghilimelele fac parte din şir şi că între acestea pot fi zero sau mai multe caractere inclusiv delimitatorii.

"Introduceți caracterele" "caractere !@#$%^^&* " "Pop N. Ionescu"

Trebuie reținut faptul că şirul "abcd" şi simbolul *abcd* nu sunt unul şi acelaşi lucru. Deşi conțin aceleaşi caractere prezența ghilimelelor face ca acestea să fie tipuri de date diferite. De obicei spațiul este folosit în CLIPS ca delimitator pentru a separa câmpurile (mai ales pentru simboluri) şi celelalte grupuri de caractere cu semnificație specială pentru limbaj (numite **tokens**). Spațiile adiționale utilizate între aceste grupuri (tokens) sunt neglijate. Intr-un şir însă spațiul este considerat ca aparținând şirului, oriunde ar fi acesta amplasat. Din acest motiv sunt considerate a fi şiruri diferite:

"spațiu" " spațiu" "spațiu ".

Fără ghilimele spațiile ar fi ignorate. Mai trebuie reținut faptul că nu se pot plasa direct într-un şir ghilimelele şi de exemplu înşiruirea de caractere:

"" trei_câmpuri ""

este interpretată de CLIPS ca fiind trei grupuri separate :

""
trei_câmpuri
"" .

Pentru introducerea într-un şir a ghilimelelor este necesară folosirea caracterului **backslash** \ . De exemplu: **"\" un_câmp \""**
este interpretat de CLIPS ca fiind câmpul: **"" un_câmp ""** .

De această dată a fost creat un singur câmp deoarece folosirea caracterului backslash împiedică interpretarea ghilimelelor ca fiind delimitatori. Introducerea unui caracter backslash într-un şir presupune folosirea a două astfel de caractere, succesiv. De exemplu : **"\\ un_câmp \\"**
va fi interpretat de CLIPS ca fiind câmpul: **"\ un_câmp \"** .

Celelalte patru tipuri de câmpuri **external address**, **fact address**, **instance name** şi **instance address** prezintă un interes mai mic pentru utilizatorul obişnuit. External address reprezintă adresa unei structuri de date externe returnate de un **user-defined function** (funcție definită de utilizator scrisă într-un limbaj ca C sau Ada şi legată cu CLIPS-ul pentru a adăuga o funcționalitate adițională).

Dar valoarea unei adrese externe nu poate fi specificată ca fiind o secvență de caractere ce formează un tokens şi nu este posibilă crearea unui astfel de câmp de către utilizator într-o variantă nemodificată a CLIPS-ului. Reprezentarea unei astfel de adrese externe este:

<pointer-XXXXXX> unde XXXXXX este external address.

Un fapt este o listă de valori atomice şi este referit poziţional, fapte ordonate - ordered facts, sau non-ordered/template facts - fapte neordonate (cadru sau şablon). Referirea unui fapt se face cu ajutorul unui index sau adresă. Formatul pentru fact address este:

<Fact-XXX> unde XXX este indexul de fapt.

Instance name şi instance address sunt câmpuri utilizate în conjuncție cu COOL. Obiectele în CLIPS sunt definite a fi floats, integers, symbols, strings, multifield values (valori multicâmp), external addresses, fact addresses sau o instanță a unei clase utilizator (user-defined class). O clasă definită de utilizator este realizată, folosind un constructor **defclass**. O instanță a unei astfel de clase este creată cu funcția **makeinstance** şi poate fi referită în mod unic cu ajutorul adresei acesteia. Câteva exemple de instance name sunt:

[pump-1] [foo] [+++] [123-890]

Parantezele pătrate nu sunt parte componentă din instanță, ele indicând doar faptul că simbolul închis între acestea este un nume instanțial (instance-name). O adresă de instanță poate fi obținută prin asocierea unei variabile, valorii returnate de funcția **instance-address**. Reprezentarea unei adrese instanțiale este:

<instance-XXX> unde XXX este numele instanței.

1.3. Funcțiile

O funcție în CLIPS este un element format din cod executabil cu un nume specific, care returnează o valoare utilă sau are un anumit efect asupra informației. Există două tipuri de funcții: **user defined functions** şi **system defined functions**, acestea fiind elemente de cod scrise în limbaje externe (cum ar fi C, FORTRAN sau ADA) şi linkeditate cu mediul CLIPS. Funcțiile definite de sistem sunt acele funcții ce au fost definite intern de mediul CLIPS-ului, iar funcțiile definite de utilizator sunt funcții care au fost definite extern mediului CLIPS.

Constructorul **deffunction** ajută utilizatorii să definească noi funcții direct în mediul CLIPS-ului utilizând sintaxa acestuia. Pot fi definite şi funcții generice cu ajutorul constructorilor **defgeneric** şi **defmethod**. Funcțiile apelate în CLIPS folosesc o notație prefix, argumentele unei funcții apar întotdeauna după numele funcției. Funcția apelată începe cu o paranteză deschisă ‘ **(** ‘ , urmată de numele

funcției, de argumentele acesteia ,fiecare argument fiind separat de unul sau mai multe spații de celelalte argumente şi se termină cu o paranteză închisă ‘ **)** ‘ .

Argumentele pot fi tipuri primitive de date, variabile sau chiar alte funcții apelate de aceasta. De exemplu pentru expresiile următoare care apelează funcțiile de adunare ‘**+**’ şi de multiplicare ‘*’, amplasarea greşită a parantezelor duce la modificarea numărului de argumente şi de aici la obținerea unui rezultat greşit.:

```
(+ 1 2)                 (* 3 4.5 6)
(+ 3 (* 7 8) 4)         (* 5 (+ (* 2 3 4) 1 2) (* 5 6))
```

1.4. Constructorii

Mai mulți constructori de definiție apar în CLIPS: **defmodule**, **defrule**, **deffacts**, **deftemplate**, **defglobal**, **deffunction**, **defclass**, **definstances**, **defgeneric**, **defmessagehandler** şi **defmethod**. Toate construcțiile în CLIPS sunt cuprinse între paranteze, începând cu o paranteză deschisă ‘ **(** ‘ şi terminându-se cu o paranteză închisă ‘ **)** ‘. Aceşti constructori se folosesc pentru a adăuga o informație la baza de cunoştințe a CLIPS-ului.

Față de funcțiile apelate aceştia nu returnează niciodată o valoare. Se oferă posibilitatea adăugării unui comentariu la numele construcției (excepție face doar defglobal). Comentariile pot fi amplasate în CLIPS utilizând punctul şi virgula ‘ **;** ’, iar CLIPS-ul va ignora tot ce va găsi după punct şi virgulă. Dacă avem la început de linie punct şi virgulă întreaga linie va fi tratată ca fiind un comentariu. In momentul în care se va dori lansarea în execuție a unui program CLIPS-ul va încerca toate construcțiile ignorând comentariile.

1.5. Faptele

In CLIPS sunt trei forme principale de reprezentare a informației: fapte (**facts**), obiecte şi variabile globale. Faptele sunt forma principală de reprezentare a informației în sistemul CLIPS. Fiecare fapt reprezintă un element de informație care va fi plasat în lista de fapte numită **factlist**. Faptul este forma fundamentală de date folosită de **reguli** şi constă într-un **nume de relație** urmat de zero sau mai multe **sloturi** (câmpuri simbolice cărora li se pot asocia valori). Forma generală a unui fapt va fi următoarea:

fact <==> **(nume_relație [(slot)*] [(multislot)*])**

Parantezele pătrate indică faptul că existența sloturilor este opțională, iar semnul * arată faptul că putem avea zero sau mai multe sloturi. Se observă faptul că în CLIPS trebuie să cuprindem între paranteze atât numele de relație cât şi sloturile. Un slot este la rândul său format dintr-un nume de slot, căruia i se poate asocia o valoare. Există însă şi posibilitatea asocierii de mai multe valori şi în acest

caz spunem ca avem un **multislot**. Acest lucru este posibil datorită existenţei în CLIPS a multicâmpului (**multifield**) care este de fapt o secvenţă de zero sau mai multe câmpuri. Afişarea în CLIPS a unei valori multicâmp se va realiza prin închiderea acestuia între paranteze. Trebuie reţinut faptul că o valoare multifield ***(a)*** nu este unul şi acelaşi lucru cu o valoare câmp (**single field**) ***(a)*** .

slot => **(nume_slot valoare)**

multislot => **(nume_slot [valoare]*)**

Un slot trebuie să aibă obligatoriu asociată o valoare, pe când la un multislot asocierea unei valori nu e obligatorie. Valoarea pe care o asociem unui slot poate lua forma unui tip de dată primitivă (integer, float, symbol, string). Exemple de fapte ar putea fi :

```
(start)    (persoana (nume Irina Vlad)          (adresa (strada “Bistritei 32”)
                     (culoare_par ))                     (bloc X1)
                                                         (scara A)
                                                         (apartament 12))
```

In primul caz faptul (*start*) nu are nici un slot. In al doilea caz faptul cu numele de relaţie *adresa* are mai multe sloturi şi anume : *strada*, *bloc*, *scara*, *apartament*. Se observă faptul că putem asocia unui slot valori de diferite tipuri: valoarea slotului *strada* este un şir de caractere, valoarea slotului *apartament* este un întreg, iar valorile sloturilor *bloc* şi *scara* sunt simboluri. In ultimul caz, pentru faptul persoana, atât slotul *nume*, cât şi slotul *culoare-par*, pentru care nu s-a precizat o valoare explicită, sunt multisloturi (au două şi respectiv zero valori).

Trebuie reţinut faptul că în CLIPS nu are nici o importanţă ordinea apariţiei sloturilor şi de aceea două fapte care au schimbată ordinea sloturilor sunt văzute ca fiind identice.

```
(student (nume Viorel B. Radu)          (student (nr_grupa 1407)
         (nr_grupa 1407)                         (nume Viorel B. Radu)
         (an IV))                                (an IV)) .
```

1.5.1. Faptele compuse. Constructorul deftemplate

Pentru crearea unor fapte compuse (**şablon**), care sunt formate din sloturi, trebuie mai întâi ca acestea să fie declarate, astfel încât CLIPS-ul să poată să interpreteze acele sloturi ca fiind valide şi nu ca fiind nişte funcţii. Special pentru acest lucru avem construcţia **deftemplate** care va face ca o grupare de fapte cu acelaşi nume de relaţie să conţină un anumit tip de informaţie.

Un constructor deftemplate are rolul de a crea un şablon ce va fi folosit pentru a accesa câmpurile faptului după nume. Acesta este identic cu o înregistrare sau structură definită într-un limbaj de programare cum ar fi Pascal sau C. Sintaxa unei construcţii deftemplate este:

(deftemplate <deftemplate-name> [<comment>]
<slot-definition>*)

<slot-definition> ::= (slot <slot-name>) | (multislot <slot-name>)

Folosind această sintaxă faptul *student* de mai sus ar putea fi descris astfel:

(deftemplate student "Comentariu optional"
(multislot nume)
(slot nr_grupa)
(slot an))

Redefinirea unui deftemplate duce la ignorarea definiției anterioare. Nu putem redefini un deftemplate cât timp acesta este folosit (de ex. dacă o regulă foloseşte un fapt definit de acel deftemplate) şi se va primi în acest caz un mesaj de avertizare din partea CLIPS-ului. O construcție deftemplate poate avea orice combinație de sloturi single-field şi sloturi multi-field. CLIPS-ul va aplica întotdeauna definițiile pentru sloturi şi de aceea vor apărea întotdeauna erori la asocierea unei valori multiple pentru un câmp single-field.

1.5.2. Fapte ordonate

Un fapt ordonat (**ordered fact**) constă într-un nume de relație urmat de zero sau mai multe câmpuri separate de spații. Acesta ar putea fi comparat cu un slot de tip multifeld (multi-câmp) în care se pun toate valorile ce urmează după numele de relație. Lipsa sloturilor nu face necesară existența unei construcții deftemplate care să le expliciteze. Faptele ordonate sunt create automat de un deftemplate **implicit** al CLIPS-ului. Faptele create cu o construcție deftemplate **explicită** se mai numesc şi **deftemplate facts** sau **non-ordered facts**. Astfel se poate implementa relativ uşor o stivă sau o coadă (şiruri de elemente). De exemplu:

(numere_prime 2 3 5 7 11) (numere_pare 2 4 6 8 10)

Aceste fapte se pot realiza în mod echivalent, folosind o construcție deftemplate:

(deftemplate numere_pare (numere_pare (valori 2 4 6 8 10))
(multislot valori))

Câmpurile dintr-un fapt ordonat pot fi orice tip de dată primitivă (cu excepția primului câmp care reprezintă numele de relație şi care trebuie să fie de tip simbol) şi nu există restricții privitoare la ordinea acestora. Totuşi, următoarele simboluri sunt rezervate şi nu pot fi primul câmp dintr-un fapt (numele de relație): **test**, **and**, **or**, **not**, **declare**, **logical**, **object**, **exists**, şi **forall**. Deşi aceste simboluri rezervate pot fi folosite la numele unui slot prin schimbarea unei litere mici într-o litera mare, în general nu se recomandă aceasta. In următoarele exemple de fapte:

(test de inteligenta artificiala) (Test de inteligenta artificiala)

CLIPS-ul ne va informa că nu putem folosi ca nume de relație simbolul **test**, dar în al doilea caz se va accepta ca nume de relație simbolul **Test**, deoarece CLIPS-ul este **case-sensitive** şi vede cele două câmpuri ca fiind total diferite. Lista de elemente este formată din trei câmpuri: *de* , *inteligenta* , *artificială*. CLIPS-ul face deosebirea între un *ordered-fact* şi un *deftemplate-fact* extrăgând primul câmp din fapt (numele de relație) şi comparându-l apoi cu numele fiecărui deftemplate. Dacă la un fapt deftemplate, schimbând ordinea sloturilor se obținea acelaşi fapt, în schimb la un fact ordonat schimbând ordinea câmpurilor se obține un alt fapt.

(lista 2 3 4) *diferit de* (lista 4 3 2).

Uneori este mai avantajoasă folosirea unui fapt de tip deftempate în locul unui fapt ordonat, deoarece se specifică prin numele sloturilor semnificația fiecărui câmp. In cazul următor, dintre cele două fapte este de preferat un fapt deftemplate:

(adresa “Bistitei” 32 X1 A 12)

(adresa (strada “Bistritei 32”) (bloc X1) (scara A) (apartament 12))

Când un fapt deftemplate conține un singur slot (numele de slot ar putea fi sinonim cu numele de fapt) sau când faptul este folosit ca un flag sau ca un indicator, folosirea unui fapt ordonat este mai avantajoasă. De exemplu:

(conditie_satisfacuta_prematur) (timp 7:00)

1.5.3 Adăugarea şi ştergerea faptelor

Toate faptele cunoscute de CLIPS sunt păstrate în lista de fapte (**fact list**). Se pot folosi comenzile cum ar fi **assert** pentru adăugarea la listă şi **retract** pentru ştergerea din listă. Sintaxa comenzii assert este următoarea :

(assert <fact>+)

Prin <fact>+ se înțelege faptul că putem aserta un nou fapt sau mai multe fapte noi, deodată. Pentru a adăuga un fapt cu mai multe sloturi procedăm astfel:

```
CLIPS> (deftemplate  adresa
                (multislot  strada) (slot  bloc) (slot  scara) (slot  apartament))
CLIPS>(assert  (adresa  (strada Bistritei 32)
                        (bloc X1)  (scara A)  (apartament 12))
<Fact-0>
```

Funcția assert returnează o valoare: adresa de fapt. În cazul în care se asertează mai multe fapte deodată valoarea returnată va fi indexul ultimului fapt introdus. În cazul nostru s-a realizat introducerea unui singur fapt, *adresa*. După ce a fost introdus în lista de fapte, i-a fost asociat identificatorul 0. Parantezele unghiulare < > sunt folosite ca delimitatori pentru a încadra numele unui articol. La

fiecare nouă adăugare se va realiza automat o incrementare a numărului de fapte, această valoare fiind returnată de funcția assert.

Comanda **facts** este utilizată pentru afişarea faptelor aflate în lista de fapte. În cazul în care se utilizează comanda *(facts)* fără nici un parametru se va realiza afişarea tuturor faptelor din listă, dar dacă se doreşte o afişare a anumitor fapte dorite de utilizator, se pot folosi şi parametri adiționali ai acestei comenzi. Această facilitate se dovedeşte a fi foarte utilă atunci când avem un număr mare de fapte în listă. Sintaxa pentru comanda facts este:

(facts [<start> [<end> [<maximum>]]])

unde <start>, <end>, <maximum> sunt numere întregi pozitive. Toți parametrii sunt opționali şi deci comanda poate funcționa cu zero până la trei parametri.

Dacă apare primul parametru <start>, toate faptele cu indexul mai mare sau egal decât valoare lui <start> vor fi afişate; dacă apare al doilea parametru <end> atunci toate faptele cu indexul mai mic sau egal decât valoarea lui <end> vor fi afişate; dacă apare şi al treilea parametru <maximum>, atunci vor fi afişate un număr maxim de fapte. Dacă se va da comanda *(facts)* rezultatul va fi următorul:

```
CLIPS> (facts)
f-0     (adresa (strada Bistritei 32) (bloc X1) (scara A) (apartament 12))
For a total of 1 facts.
```

După cum se observă, comanda *(facts)* indică la sfârşit, numărul de fapte pe care le-a afişat pe monitor. De asemenea pentru fiecare fapt în parte va fi afişat şi identificatorul de fapt, în cazul nostru acesta fiind 'f-0'. Orice fapt inserat în listă are asociat de către CLIPS un identificator unic care începe cu litera f şi este urmat de un întreg numit şi **index** de fapt. CLIPS-ul nu acceptă introducerea unui duplicat în lista de fapte şi de aceea încercarea de asertare a unui fapt identic cu un altul anterior, duce la returnarea unui mesaj de eroare:

```
CLIPS>(assert  (adresa (strada Bistritei 32) (bloc X1) (scara A) (apartament 12))
FALSE
CLIPS>(assert  (adresa (strada Bistritei 47) (bloc X1) (scara A) (apartament 10))
<Fact-1>
```

Tot aşa cum un fapt poate fi asertat în lista de fapte, tot aşa poate fi şi şters. Operația de ştergere a unui fapt se mai numeşte şi retractare şi se realizează cu comanda *(retract)* cu sintaxa:

(retract <fact-index>+)

După cum se vede avem posibilitatea de a şterge unul sau mai multe fapte din listă. Încercarea de a şterge un fapt care nu există sau care a fost şters anterior duce de asemenea la afişarea unui mesaj de eroare: comanda *(retract)* nu returnează nici o valoare.

```
CLIPS> (retract 0)
CLIPS> (retract 1)
```

```
CLIPS> (retract 1)
[PRNTUTIL1] Unable to find fact f-1
```

Pentru a şterge faptele f-0 şi f-1 simultan, se putea folosi comanda *retact* astfel:

```
(retract 0 1)
```

In CLIPS mai există şi comanda **(clear)** al cărei efect este de a şterge toate faptele din listă, indexul de fapt începând după aceasta, din nou, cu valoarea 0.

```
CLIPS> (clear)
CLIPS> (assert  (a)   (b)   (c)  (d))
<Fact-3>
CLIPS>(facts 1 3 2)
f-1     (b)
f-2     (c)
For a total of 3 facts.
CLIPS>(assert (d))
FALSE
CLIPS>(retract 0 2 3)
CLIPS> (assert (d))
<Fact-4>          ;indexul de fapt este incrementat de unde a rămas
CLIPS> (facts 0)
f-1     (b)
f-4     (d)
For a total of 2 facts.
CLIPS> (retract 5 1)
[PRNTUTIL1] Unable to find fact f-5
CLIPS>(assert ( e ))
<Fact-5>
CLIPS>(facts)             ;se observă că f-1 a fost şters mai sus
f-4     (d)
f-5     (e)
For a total of 2 facts.
CLIPS> (retract *)
CLIPS> (facts)
CLIPS> (assert ( numar (+ 3 5)))
<Fact-6>
CLIPS> (facts)
f-6     (numar 8)
For a total of 1 facts.
```

Din exemplul de mai sus observăm că, deşi faptul *(d)* a fost şters şi apoi reintrodus, indexul de fapt a fost incrementat în continuare. CLIPS-ul nu se ocupă de o reactualizare a indexului faptelor din listă. Astfel, un fapt are întotdeauna acelaşi index din momentul asertării sale în listă, până în momentul ştergerii sale. Se mai observă de asemenea că la asertarea faptului *(e)* spațiile sunt ignorate şi că efectul comenzii *(retract 5 1)* este acela de a ne atenționa că faptul cu indexul 5 nu există, fără însă a se opri de la ştergerea faptului cu indexul 1. Efectul comenzii *(retract *)* este asemănător cu cel al comenzii *(clear)* , singura excepție fiind

indexul de fapte care nu este resetat pe valoarea 0, ci îşi continuă incrementarea de la valoarea rămasă. La asertarea faptului *numar* CLIPS-ul va realiza automat operația de adunare astfel încât atunci când se va da comanda *(facts)* vom avea în listă faptul *(numar 7)*. Se va vedea că ştergerea faptelor, duce şi la ştergerea regulilor aprinse de acele fapte, din agendă.

1.5.4. Modificarea şi duplicarea faptelor

Pentru modificarea sloturilor unui fapt avem la dispoziție comanda **modify** a cărei sintaxa este următoarea:

(modify <fact-index> <slolt-modifier>+)

<slot-modifier> ::= (<slot-name> <slot-value>)

După cum se observă avem posibilitatea modificării mai multor sloturi deodată. Această comandă practic şterge faptul pe care dorim să-l modificăm şi introduce un nou fapt care are valorile sloturilor modificate de utilizator.

Odată asertat noul fapt modificat are în mod evident şi un alt index de fapt. Trebuie reținut faptul că nu se poate modifica decât un singur fapt, iar acesta trebuie să fie de tip template, adică să conțină sloturi (un fapt ordonat nu conține sloturi şi deci nu poate fi modificat). Valoarea returnată de această funcție în caz de succes este adresa faptului nou, iar în caz de eroare (de exemplu exista deja un fapt identic cu cel modificat, sau faptul cu numărul de index specificat nu există în lista de fapte) se returnează simbolul FALSE.

```
CLIPS> (deftemplate student  (multislot  nume) (slot  nr_grupa) (slot  an) )
CLIPS> (assert (student  (nume Munteanu Dan) (nr_grupa 1406) (an IV))
               (student  (nume Chiriac Florin) (nr_grupa 1305) (an III)))
<Fact-1>
CLIPS> (modify 1 (nr_grupa 1406)  (an IV) )
<Facts-2>
CLIPS> (facts)
f-0     (student (nume Munteanu Dan) (nr_grupa 1406) (an IV))
f-2     (student (nume Chiriac Florin) (nr_grupa 1406) (an IV))
For a total of 2 facts.
CLIPS> (modify 1 (nume Violeta Dumea))
[PRNTUTIL 1] Unable to find fact f-1
CLIPS> (modify 2 (nume Munteanu Dan))
FALSE
CLIPS> (facts)
f-0     (student (nume Munteanu Dan) (nr_grupa 1406) (an IV))
For a total of 1 facts.
```

Remarcăm două erori: prima deoarece nu exista în listă un fapt cu numărul de index f-1, a doua deoarece faptul modificat nu este acceptat de CLIPS fiind identic cu un alt fapt f-0 deja existent. Funcția *modify* a returnat simbolul FALSE,

dar faptul f-2 ce se dorea a fi modificat a fost deja retractat. La a doua eroare practic s-a pierdut o înregistrare din lista de fapte şi de aceea se recomandă a se verifica mai întâi dacă nu exista deja faptul ce se doreşte a fi introdus.

Comanda **duplicate** oferă posibilitatea utilizatorului de a realiza o copie modificată a unui fapt din lista de fapte (nu se poate realiza o copie identică, deoarece CLIPS-ul nu acceptă acest lucru). Acțiunea acestei comenzi este aproape similară cu cea a comenzii *modify*, cu excepția faptului că de această dată nu este retractat faptul ce se modifică, în lista de fapte existând atât vechiul fapt cât şi faptul nou, modificat. Sintaxa comenzii *duplicate* este:

(duplicate <fact-index> <slolt-modifier>+)

<slot-modifier> ::= (<slot-name> <slot-value>)

De asemenea nu putem duplica decât un singur fapt, iar în cadrul acestuia putem modifica valorile câmpurilor unui slot sau a mai multor sloturi. Funcția returnează ca şi funcția *modify*, în caz de succes adresa de fapt a noului fapt, duplicat, iar în caz de eroare simbolul FALSE.

```
CLIPS> (duplicate 0 (an II) (nr_grupa 1207))
<Fact-3>
CLIPS> (facts)
f-0     (student (nume Munteanu Dan) (nr_grupa 1406) (an IV))
f-3     (student (nume Munteanu Dan) (nr_grupa 1207) (an II))
For a total of 2 facts.
CLIPS> (duplicate 0 (an II) (nr_grupa 1207)
FALSE
CLIPS> (duplicate 1 (an I))
[PRNTUTIL 1] Unable to find fact f-1
```

Şi de această dată s-a urmărit realizarea unor greşeli similare cu cele făcute la funcția *modify* , dar după cum era de aşteptat nu s-a mai pierdut informație, deoarece funcția *duplicate* nu retractează faptul ce se doreşte a fi duplicat.

1.5.5. Urmărirea faptelor. Comanda watch

CLIPS oferă o serie de comenzi pentru depanarea unui program. Cu ajutorul comenzii **watch** se pot urmări mai uşor faptele care sunt asertate sau retractate, fiind mai uşor de observat ce schimbări se produc în lista de fapte. Sintaxa generală a comenzii este:

(watch <watch-item>)

unde <watch-item> poate fi unul din simbolurile: **facts**, **rules**, **activations**, **statistics**, **complilations**, **focus** sau **all**. Dacă se foloseşte simbolul *all* atunci se validează toate watch-urile. Se pot realiza orice combinații de watch-uri.

La startarea CLIPS-ului este activat doar watch compilations, toate celelalte nefiind active. Se poate observa acest lucru la încărcarea şi compilarea unui program. In momentul de fața prezintă interes doar **watch facts**. Cu ajutorul acestei comenzi putem urmări faptele după cum urmează :

```
CLIPS> (clear)
CLIPS> (watch facts)
CLIPS> (assert (a)  (b))
==>     f-0     (a)
==>     f-1     (b)
<Fact-1>
CLIPS> (deftemplate copil  (slot  nume)  (slot  tata)  (slot  mama))
CLIPS> (assert (copil  (nume Iancu)  (tata Andrei) (mama Ana)))
==>     f-2     (copil (nume Iancu)  (tata Andrei) (mama Ana))
<Fact-2>
CLIPS> (modify 2 (nume Oana))
<==     f-2     (copil (nume Iancu)  (tata Andrei) (mama Ana))
==>     f-3     (copil (nume Oana)  (tata Andrei) (mama Ana))
<Fact-3>
CLIPS> (retract 3 1)
<==     f-3     (copil (nume Oana)  (tata Andrei) (mama Ana))
<==     f-1     (b)
```

Secvența de caractere ==> ne arată că faptul a fost asertat, intrând în lista de fapte; pe când secvența de caractere <== ne arată că faptul a fost retractat. Comanda *(watch facts)* ne arată lista de fapte într-un mod dinamic, utilizatorul poate observa ce fapte intră şi ce fapte ies din listă. Comanda *(facts)* însă ne prezintă într-un mod static lista de fapte. Pentru a opri efectul unui watch se foloseşte comanda *unwatch*. Sintaxa comenzii *unwatch* este următoarea:

(unwatch <watch-item>)

Evident că pentru a dezactiva urmărirea faptelor trebuie să dăm comanda *(unwatch facts)*. In varianta de CLIPS pentru Windows se pot urmări faptele din meniul Window, opțiunea Facts Window. De asemenea pot fi setate sau resetate din meniul Execution, opțiunea Watch (Ctrl+W) acele wach-uri pe care le dorim a fi active sau nu, la un moment dat.

1.5.6. Gruparea faptelor. Constructorul deffacts

Intr-un program este foarte avantajos ca după ce au fost declarate construcțiile deftemplate (dacă este necesară prezența lor) să poată fi asertate toate faptele ce reprezintă baza inițială de cunoaştere într-o singură aserțiune. Aceste fapte sunt cunoscute a fi adevărate încă înainte de rularea programului. Se poate defini un grup de fapte care reprezintă baza inițială de cunoaştere cu ajutorul construcției *deffacts*. Sintaxa comenzii este următoarea:

```
(deffacts   <deffacts name>   [<optional comment>]
                    <facts>* )
```

In cadrul construcției *deffacts* poate fi folosit ca nume orice simbol valid. După numele construcției urmează opțional un comentariu şi apoi faptele ce vor fi asertate în lista de fapte. Pentru a realiza asertarea faptelor dintr-o declarație *deffacts* trebuie obligatoriu utilizată comanda **reset**. Efectul acestei comenzi este acela că şterge toate faptele din baza de fapte (ca şi comanda *clear*) şi introduce faptele din declarațiile deffacts. Sintaxa comenzii este:

```
(reset)
```

Se va putea observa de asemenea că o dată cu comanda *reset* este introdus în lista de fapte un nou fapt generat implicit de aceasta, numit şi **inițial fact**. CLIPS-ul foloseşte acest fapt pentru a aprinde o regulă care nu are nici un element condițional în LHS (se va vedea în continuare). Introducerea acestui fapt se face automat de către CLIPS cu ajutorul a doua construcții :

```
(deftemplate initial-fact)
(deffacts initial-fact
        (initial-fact))
```

Identificatorul de fapt al lui *initial-fact* este întotdeauna f-0 , deoarece acesta este primul fapt ce va fi introdus de CLIPS în lista de fapte după ce a fost dată comanda *reset*. Dacă se doreşte o vizualizare a tuturor construcțiilor deffacts de poate utiliza comanda *get-deffacts-list*.

```
CLIPS> (clear)
CLIPS> (assert (a))
<Fact-0>
CLIPS> (facts)
f-0     (a)
For a total of 1 facts.
CLIPS> (get-deffacts-list)
(initial-fact)
CLIPS> (watch facts)
CLIPS> (deftemplate data  (slot zi)  (slot luna)   (slot an) )
CLIPS> (deftemplate angajat   (multislot nume)  (slot salariu) )
CLIPS> (deffacts date-initiale    “Baza de salarizare”
                (angajat (nume Cozma George) (salariu 1500000) )
                (angajat (nume Miron Liliana) (salariu 1650000) )
                (angajat (nume Mazilu Razvan) (salariu 1580000) )
                (data (zi 31) (luna 9 ) (an 1991) ) )
CLIPS> (deffacts salariu_maxim)
CLIPS>(get-deffacts-list)
(initial-fact  date-initiale  salariu_maxim)
CLIPS> (reset)
```

```
<==     f-0     (a)
==>     f-0     (initial-fact)
==>     f-1     (angajat (nume Cozma George) (salariu 1500000) )
==>     f-2     (angajat (nume Miron Liliana) (salariu 1650000) )
==>     f-3     (angajat (nume Mazilu Razvan) (salariu 1580000) )
==>     f-4     (data (zi 31) (luna 9) (an 1991) )
```

Se observă existența încă de la startarea CLIPS-ului a construcției deffacts *initial-fact*. Funcția *get-deffacts-list* ne informează de existența celor trei construcții deffacts, dar abia după ce se dă comanda *reset* în lista de fapte vor fi introduse faptele acestora. Ultima construcție deffacts *salariu_minim* nu are nici un fapt, existența ei fiind oarecum inutilă.

1.6. Definirea regulilor. Constructorul defrule

Una din metodele primare de reprezentare a cunoaşterii în CLIPS este regula. O regulă este o colecție de condiții şi de acțiuni ce urmează a fi executate dacă condițiile sunt îndeplinite. Un sistem expert trebuie să aibă atât reguli, cât şi fapte. Regulile executate sau 'aprinse' se bazează pe existența sau non-existența unor fapte sau instanțe ale unei clase definite de utilizator. Mecanismul (motorul de inferență) al CLIPS-ului aşteaptă potrivirea regulilor cu starea curentă a sistemului (reprezentată de lista de fapte şi lista de instanțe) şi apoi aplică acțiunile.

Regulile pot fi scrise direct în CLIPS sau pot fi încărcate dintr-un fişier cu reguli, creat cu un editor. Se poate folosi în acest scop editorul încorporat al CLIPS-ului sau oricare alt editor. Față de introducerea direct în **top level** a regulilor sau comenzilor, folosirea unui editor aduce posibilitatea corectării eventualelor erori, fără a mai fi necesară rescrierea din nou a regulii. Ne putem imagina cazul în care un calculator ce are introduse deja faptele şi regulile, avertizează operatorul ca în acea zi se dau salariile. Pseudocodul ar fi:

IF ziua curenta este 31
THEN răspunsul este "caseria trebuie să distribuie salariile"

Presupunem din exemplul anterior ca fiind deja asertat faptul *data*. In acest caz regula noastră ar putea arăta astfel:

```
(defrule actiunea_zilei   "Comentariu"
        (data (zi 31))
        =>
        (assert (actiune "caseria da banii…") ) )
```

Dacă s-a introdus corect regula şi nu s-a omis nici o paranteză, atunci prompter-ul "**CLIPS>**" va reapare. In caz contrar va apare un mesaj de eroare în care se indică cuvântul cheie greşit sau amplasarea greşită a parantezelor. Sintaxa comenzii defrule este:

```
(defrule <rule-name>  [<comment>]
        [<declaration>]                 ; Rule Properties
        <conditional-element>*          ;  Left-Hand  Side  (LHS)
        =>
        <action>* )                     ; Right-Hand Side  (RHS)
```

Redefinirea unei reguli deja existente, face ca regula anterioară cu acelaşi nume să fie ştearsă, chiar dacă regula actuală nou introdusă are erori. După cum se observă după numele regulii care poate fi orice simbol valid, după comentariu opțional care trebuie cuprins între ghilimele (comentariile pot fi introduse şi după punct şi virgula ' **;** ') urmează zero sau mai multe condiții (**pattern**-uri) şi apoi zero sau mai multe acțiuni. Practic o regulă este formată din două părți LHS şi RHS separate de secvența de caractere (=>) pe care o putem numi şi săgeată, formată dintr-un semn 'egal' şi un semn 'mai mare decât' . In comparația regulii cu un **IF-THEN** săgeata marchează începutul părții THEN. Partea stângă LHS este formată din zero sau mai multe elemente condiționale (CE) , care sunt legate între ele de un **and** implicit. RHS-ul este format dintr-o listă de acțiuni care sunt executate dacă şi numai dacă toate elementele condiționale (pattern-urile) sunt satisfăcute. Nu există nici o limită pentru numărul de condiții sau de reguli dintr-o regulă, singura limitare o constituie memoria actuală valabilă. CLIPS-ul aşteaptă îndeplinirea pattern-urilor unei reguli verificând faptele din lista de fapte. Atunci când toate pattern-urile se potrivesc cu faptele, regula este activată şi este pusă în **agenda**.

Agenda reprezintă în CLIPS o colecție de reguli activate. In agendă pot fi zero sau mai multe fapte, retractarea unui fapt din lista de fapte putând duce la stingerea unor reguli şi scoaterea lor automat din agendă. Dacă avem mai multe reguli care se pot aprinde CLIPS-ul este cel care hotărăşte care din reguli este cea mai apropiată şi le ordonează pe toate în agendă (CLIPS-ul ține cont şi de prioritatea pe care a oferit-o eventual utilizatorul regulii, numită şi **salience**).

Dacă o regulă nu are nici o condiție în LHS, atunci CLIPS-ul va adăuga automat pattern-ul special *initial-fact*. Orice regulă care nu are condiții se va aprinde automat şi va fi plasată în agendă, după ce va fi dată comanda *reset* şi faptul *initial-fact* va fi introdus în lista de fapte. Se poate întâmpla de asemenea ca o regulă să nu aibă nici o acțiune în RHS. Acest lucru nu este prea util, dar e bine de ştiut că este posibilă realizarea unei astfel de reguli.

Lansarea în execuție în CLIPS a unui program se face cu ajutorul comenzii *run* a cărei sintaxa este următoarea :

```
( run  [ <limit> ] )
```

Argumentul opțional <limit> este un număr întreg, care specifică numărul maxim de reguli ce vor fi aprinse. Dacă acest argument nu apare, atunci el are implicit valoarea –1 şi toate regulile vor fi aprinse până ce nu va mai rămâne nici una în agendă. In caz contrar execuția regulilor va înceta după ce un număr de <limit> reguli au fost aprinse. Următoarea comanda *run* va continua execuția din

locul de unde s-a oprit, astfel încât comanda **(run 1)** ar fi echivalentul unei rulări **pas cu pas** a programului.

Trebuie reținut faptul că o regulă o dată aprinsă, nu mai poate fi activată de obicei din nou. Este necesară ori 'reîmprospătarea' ei cu ajutorul comenzii **refresh**, ori apelarea comenzii **reset**, care şterge toate faptele şi reintroduce baza inițială de fapte din construcțiile *deffacts* de la care s-a plecat. Odată refăcută baza inițială de cunoaştere se poate da din nou comanda **run**. Exemple de reguli:

```
(defrule bariera_coborita                (defrule bariera_ridicata
        (tren_apropiat da)                       (tren_apropiat nu)
        =>                                       =>
        (actiune "Coboara bariera.")             (actiune "Ridica bariera.")
```

1.6.1. Afişarea agendei

După cum s-a mai spus agenda este o colecție de reguli activate, iar această listă poate fi afişată cu ajutorul comenzii **agenda** a cărei sintaxa este următoarea:

```
(agenda)
```

Dacă nu avem nici o regulă activată, atunci prompter-ul CLIPS-ului va apare imediat. Dacă în schimb avem una sau mai multe reguli în agendă, atunci acestea vor fi afişate în felul următor. Mai întâi este afişat un număr ce indică prioritatea regulii, aceasta având în mod implicit valoarea 0, dacă utilizatorul nu a asociat o prioritate regulii respective. După prioritate (salience) este afişat numele regulii urmat de identificatorii de fapt ce s-au potrivit cu pattern-urile (condițiile) regulii. In exemplul următor presupunem deja introduse construcția deffacts *date-initiale* şi regula *actiunea_zilei.*

```
CLIPS> (defrule find_person              "Die or life"
                (angajat (nume Cozma George))
                =>
                (assert (mesaj "Exista un angajat cu acest nume.")) )
CLIPS> (agenda)
0       find_person:  f-1
0       actiunea_zilei:  f-4
For a total of 2 activations.
CLIPS> (watch facts)
CLIPS> (run 1)
==>     f-5     (mesaj "Exista un angajat cu acest nume.")
CLIPS>(agenda)
0       actiunea_zilei:  f-4
For a total of 1 activation.
CLIPS> (run 1)
==>     f-6     (actiune "caseria da banii…")
CLIPS> (agenda)
```

In exemplul nostru la aprinderea unei reguli, se asertează în lista de fapte un fapt nou ce specifică o acțiune sau un mesaj. După cum se observă fiecare regulă aprinsă este scoasă din agendă, astfel încât la un moment dat nu se mai găseşte nici o regulă activată în agendă. Din acest moment comanda *run* nu mai are nici un efect.

In CLIPS o regulă nu se aprinde mai mult de o singură dată pentru un anumit set de fapte, această proprietate purtând şi numele de **refracție**. Fără această proprietate un sistem expert s-ar afla întotdeauna într-o buclă infinită, fiecare regulă fiind din nou şi din nou aprinsă de acelaşi fapt. Pentru a reaprinde regula, este necesară retractarea faptului ce a aprins-o (dacă sunt mai multe fapte, se alege unul la întâmplare) şi apoi asertarea acestuia din nou. Noul fapt va avea un nou identificator de fapt, iar regula va putea fi aprinsă din nou. Altă posibilitate este folosirea comenzii *refresh* a cărei sintaxa este următoarea:

(refresh <rule-name>)

Comanda *refresh* pune înapoi în agendă toate activările regulii, ce au fost deja aprinse (singura restricție este ca în lista de fapte să fie încă prezent faptul ce a dus la activarea acesteia).

```
CLIPS> (modify 1 (salariu 1700000))
<==     f-1     (angajat (nume Cozma George) (salariu 1500000) )
==>     f-7     (angajat (nume Cozma George) (salariu 1700000) )
<Fact-7>
CLIPS> (agenda)
0       find_person:  f-7
For a total of 1 activation.
CLIPS> (refresh actiunea_zilei)
CLIPS> (agenda)
0       actiunea_zilei:  f-4
0       find_person:  f-7
For a total of 2 activations.
CLIPS> (unwatch facts)
CLIPS> (retract 4 7)
CLIPS> (agenda)
CLIPS> (assert (angajat (nume George Cozma) ) )
CLIPS> (refresh actiunea_zilei)
CLIPS> (agenda)
```

Se observă faptul ca la ultima asertare s-au inversat în mod intenționat numele cu prenumele, din acest motiv CLIPS-ul vede cele două fapte ca fiind distincte, iar regula nu se mai aprinde. Dacă la asertarea unui fapt se omite asocierea unei valori unui slot, atunci CLIPS-ul va asocia implicit simbolul **nil** acelui slot. Ultimul *refresh* nu are nici un efect deoarece faptul care a aprins regula a fost retractat.

1.6.2. Urmărirea activărilor, a regulilor şi a statisticilor

La fel cum se putea urmări într-un mod dinamic asertarea şi retractarea faptelor, tot aşa cu ajutorul comenzii *watch* pot fi urmărite şi activările. Efectul este aproape similar cu cel al comenzii *agenda* putându-se observa faptele din agendă care sunt activate, sau care sunt dezactivate de retractarea sau modificarea vreunui fapt. In mod similar secvența de caractere <== specifică ştergerea unei activări din agendă, iar secvența de caractere ==> adăugarea unei activări la agendă. Se dă următorul exemplu în continuarea celui anterior:

```
CLIPS> (watch activations)
CLIPS> (reset)
==>     Activation 0     find_person:  f-1
==>     Activation 0   actiunea_zilei:  f-4
CLIPS> (retract 1)
 <==    Activation 0     find_person:  f-1
CLIPS> (run)
CLIPS> (unwatch activations)
CLIPS> (agenda)
```

Se observă că faptul *watch activations* nu afişează nici un mesaj atunci când o regulă este aprinsă (executată) după ce a fost dată comanda *run* ,deşi aceasta a fost ştearsă din agendă în momentul aprinderii sale. Dacă se urmăresc regulile atunci se va afişa un mesaj de fiecare dată când o regulă este aprinsă.

```
CLIPS> (reset)
CLIPS> (watch rules)
CLIPS> (run)
FIRE    1 actiunea_zilei:  f-4
FIRE    2 find_person:  f-1
CLIPS>(unwatch rules)
```

Numărul aflat după simbolul FIRE indică cât de multe reguli s-au aprins după ce a fost dată comanda *run*. Dacă se urmăresc statisticile vor fi afişate pe ecran o serie de mesaje informaționale după cum se va vedea:

```
CLIPS> (reset)
CLIPS> (watch statistics)
CLIPS> (run)
2 rules fired              Run time is 0.054 seconds.
36.4 rules per second.
6 mean number of facts (7 maximum)
1 mean number of instances (1 maximum)
2 mean number of activations (2 maximum)
CLIPS> (unwatch statistics)
```

1.6.3. Realizarea de breakpoint-uri

Pentru a ajunge la stadiul final al unui program utilizatorul trebuie sa observe efectele produse de fiecare regulă în parte. De obicei într-un program cu un număr mic de reguli se poate utiliza comanda **Step** (Ctrl+T) din meniul *Execution* al interfeței pe care o are CLIPS-ul în Windows sau comanda (**run 1**) care este echivalentă cu cea de mai sus. Intr-un program cu un număr mare de reguli utilizatorul poate să oprească execuția programului la o anumită regulă folosind un punct de oprire (breakpoint).

Un *breakpoint* este prin definiție în cazul CLIPS-ului o regulă a cărei execuție a fost oprită înainte de a fi aprinsă. Stabilirea unui *breakpoint* pentru o anumită regulă se face cu ajutorul comenzii **set-break** a cărei sintaxă este:

```
(set-break  <rule-name>)
```

Această funcție nu returnează nici o valoare. Fie exemplul următor care ilustrează folosirea *breakpoint*-urilor prin realizarea unor reguli de tip 'cascadă' (aprinderea unei reguli duce la aprinderea celeilalte şi în continuare tot aşa):

```
(defrule regula_1
        =>
        (assert (aprinde regula_2) ) )
(defrule regula_2
        (aprinde regula_2)
        =>
        (assert (aprinde regula_3) ) )
(defrule regula_3
        (aprinde regula_3)
        => )
```

Prima regulă nu are elemente condiționale în LHS, deci va fi aprinsă de construcția *initial-fact* (este necesară comanda *reset*), efectul execuției acesteia fiind asertarea unui fapt nou în lista de fapte, care va aprinde regula următoare. Ultima regulă nu are acțiuni în RHS, ea marcând sfârşitul cascadei.

```
CLIPS> (watch rules)
CLIPS> (reset)
CLIPS> (run)
FIRE    1 regula_1:     f-0
FIRE    2 regula_2:     f-1
FIRE    3 regula_3:     f-2
```

Tastând comenzile anterioare, se vede că, la comanda *run,* s-au aprins toate regulile, în ordine. In continuare se observă modul în care opresc execuția comenzile *set-break*:

```
CLIPS> (set-break regula_2)
CLIPS> (set-break regula_3)
CLIPS> (reset)                  ;se şterge agenda, fact-list şi regulile  pot
CLIPS> (run)                    ; sa se aprinda din nou la comanda run
FIRE    1 regula_1:     f-0
Breaking on rule regula_2
CLIPS> (run)
FIRE    1 regula_2:     f-1
Breaking on rule regula_3
CLIPS> (run)
FIRE    1 regula_3:     f-2
```

Trebuie reținut faptul că cel puțin o regulă trebuie să se aprindă la comanda *run* înainte ca execuţia să fie oprită de un *breakpoint* (în caz contrar comanda *run* nu are nici un efect). Pentru afişarea unei liste cu *breakpoint*-urile existente se foloseşte comanda **show-breaks** a cărei sintaxa este următoarea:

(show –breaks [<module-name>])

Până în acest moment nu a fost folosit decât modulul **MAIN** (modulul principal, implicit, al CLIPS-ului). Utilizatorul are posibilitatea grupării regulilor în module - în funcție de criteriile stabilite de el însuşi. Dacă nu este specificat numele modulului <module-name> se afişează toate *breakpoint*-urile din modulul curent (în cazul nostru MAIN), în caz contrar se afişează doar *breakpoint*-urile din modulul specificat. Folosirea ca argument a simbolului * are ca efect afişarea regulilor ce au *breakpoint*-uri din toate modulele.

Pentru ştergerea *breakpoint*-urilor se foloseşte comanda **remove-break**, care fără argument şterge toate *breakpoint*-urile; în caz contrar ştergând doar *breakpoint*-ul specificat. Sintaxa comenzii *remove-break* este următoarea:

(remove-break [<rule-name>])

1.7. Operații efectuate asupra construcțiilor din CLIPS

1.7.1. Incărcarea şi salvarea construcțiilor

Încărcarea în mediul CLIPS-ului a construcțiilor aflate într-un fişier se realizează cu ajutorul comenzii **load**. Sintaxa comenzii *load* este următoarea :

(load <file-name>)

Fişierul al cărui nume este specificat cu <file-name>, poate fi realizat cu ajutorul editorului de text al CLIPS-ului sau cu un alt editor. Marile avantaje pe care le prezintă folosirea unui fişier pentru stocarea construcțiilor se observă în momentul în care CLIPS-ul verifică sintaxa construcțiilor sau când se lansează în execuție programul. Dacă s-ar realiza introducerea construcțiilor direct de la

prompter în momentul în care ar fi fost semnalată o eroare, ar fi trebuit să reintroducem construcția respectivă.

Presupunând că avem într-un fişier *exemplu.clp* aflat pe discul C, construcții de tipul *deffacts, deftemplate, defrule*, încărcarea acestora se va face cu:

(load "C:exemplu.clp")

Dacă fişierul respectiv se află într-un director şi trebuie să specificăm calea, atunci pentru a rezolva problema *backslash*-ului care este interpretat de CLIPS ca fiind un caracter special, vor trebui folosite două *backslash*-uri pentru a crea un singur *backslash* (s-a văzut anterior). Specificarea căii se face astfel:

(load "C : \ \ temp \ \ programe \ \ exemplu.clp")

Comanda *load* returnează simbolul **TRUE** atunci când nu sunt erori în fişierul încărcat, respectiv simbolul **FALSE** în caz contrar. Dacă este activată **watch compilations** atunci va fi afişat un mesaj informațional (incluzând numele şi tipul construcției) pentru fiecare construcție încărcată.

Dacă nu este activată atunci pentru fiecare construcție va fi afişat un caracter ('*'pentru *defrule*; '$'pentru *deffacts*; '%'pentru *deftemplate*; '!'pentru *deffunctions*; ':'pentru *defglobal*; '^'pentru *defgeneric*; '&'pentru *defmethod*; '#'pentru *defclass*; '@'pentru *definstances*; "~" pentru *defmessage-handler*; "+"pentru *defmodule*). Nu este obligatorie punerea tuturor construcțiilor într-un singur fişier, dar în cazul folosirii mai multor fişiere trebuiesc încărcate toate fişierele pentru o bună execuție a programului. Dacă se folosesc construcții *deffacts* se dă comanda **reset** pentru a introduce faptele în baza de fapte, iar apoi comanda **run** pentru a lansa în execuție regulile şi funcțiile programului.

```
CLIPS> (watch compilations)
CLIPS> (load "exemplu.clp")
Defining deftemplate: template_1
Defining deffacts: initial_facts
Defining defrule: regula_1  +j
TRUE
```

In acest exemplu fictiv se observă mesajele afişate pentru fiecare construcție încărcată. Şirul "+j" afişat la sfârşitul lui "Defining defrule" este un mesaj intern al CLIPS-ului despre structura internă a regulii compilate. Pentru acelaşi exemplu fictiv dacă nu ar fi activată *watch compilations*, efectul încărcării fişierului ar fi următorul:

```
CLIPS> (clear)
CLIPS> (unwatch compilations)
CLIPS> (load "exemplu.clp")
%$*
TRUE
```

Pentru a salva toate construcţiile existente în CLIPS la un moment dat, se foloseşte comanda **save** a cărei sintaxa este următoarea:

(save <file-name>)

In general folosirea unui editor de către utilizator implică şi realizarea modificărilor şi salvarea construcţiilor cu acelaşi editor, din acest motiv folosirea comenzii *save* este mai rară (dezavantaj : nu se pot salva în fişier doar anumite construcţii). La fel ca şi la comanda *open* se poate realiza salvarea construcţiilor într-un fişier aflat într-un anumit director prin specificarea căii acestuia.

(save "C : \ \ temp \ \ programe \ \ exemplu.clp")

1.7.2. Afişarea listei membrilor unei construcţii specificate

Pentru afişarea listei regulilor aflate la un moment dat în CLIPS se utilizează comanda **list-defrules**. De asemenea comenzile **list-deftemplates** şi **list-deffacts** sunt utilizate pentru afişarea construcţiilor deftemplate, respectiv a construcţiilor deffacts. Sintaxa acestor comenzi este următoarea:

(list-defrules)
(list-deftemplates)
(list-deffacts)

```
CLIPS> (list-deffacts)
initial-fact
date-initiale
For a total of 2 deffacts.
CLIPS> (list-deftemplates)
Adresa
For a total of 1 deftemplates.
```

1.7.3. Afişarea textului unei construcţii specificate

Ca şi mai sus avem la dispoziţie trei comenzi **ppdefrule** (pretty-print defrule), **ppdeftemplate** şi **ppdeffacts** utilizate la afişarea textului (conţinutului) unei construcţii defrule, deftemplate sau deffacts. Sintaxa acestora este următoarea:

(ppdefrule <defrule-name>)
(ppdeftemplate <deftemplate-name>)
(ppdeffacts <deffacts-name>)

Singurul argument pe care îl acceptă aceste funcţii este numele construcţiei defrule, deftemplate sau deffacts. La afişare CLIPS-ul aranjează liniile şi spaţiile într-o manieră proprie, astfel încât textul să fie cât mai uşor de citit. De exemplu:

```
CLIPS> (ppdefrule actiunea_zilei)
(defrule MAIN::actiunea_zilei  "Comentariu"
    (data (zi 31))
    =>
    (assert (actiune "caseria da banii…")))

CLIPS>(ppdeftemplate data)
(deftemplate MAIN::data
    (slot zi)
    (slot luna)
    (slot an)))

CLIPS>(ppdeffacts date-initiale)
(deffacts MAIN::date-initiale  "Baza de salarizare"
    (angajat (nume Cozma George) (salariu 1500000) )
    (angajat (nume Miron Liliana) (salariu 1650000) )
    (angajat (nume Mazilu Razvan) (salariu 1580000) )
    (data (zi 31) (luna 9 ) (an 1991)))
CLIPS> (ppdeffacts initial-fact)
```

Simbolul MAIN:: care precede fiecare construcție, indică modulul în care au fost amplasate. Modulele sunt un mecanism de partiționare a bazei de cunoaştere, utilizatorul având posibilitatea să grupeze regulile în funcție de acțiunile realizate de acestea şi de preferințele sale în module. In cazul de față nu există decât modulul principal şi anume modulul MAIN. Se observă că la încercarea de afişare a construcției deffacts *inițial fact* nu se obține nici un rezultat, aceasta fiind creată automat de către CLIPS şi nu de utilizator.

1.7.4. Ştergerea membrilor unei construcții specificate

Comenzile **undefrule** , **undeftemplate** şi **undeffacts** sunt utilizate pentru ştergerea unei construcții defrule, deftemplate şi deffacts. Sintaxa acestora este:

```
(undefrule <defrule-name>)
(undeftemplate <deftemplate-name>)
(undeffacts <deffacts-name>)
```

După cum se observă comenzile nu acceptă decât un singur argument, iar acela este numele construcției ce se doreşte a fi ştearsă.

```
CLIPS> (undefrule actiunea_zilei)
CLIPS> (list-defrule)
find_person
For a total of 1 defrules.

CLIPS> (undeffacts date_initiale)
CLIPS> (list-deffacts)
initial-fact
For a total of 1 deffacts.
```

```
CLIPS> (undeffacts initial-fact)
CLIPS> (list deffacts)
```

Se observă faptul că putem şterge construcţiile deffacts *initial-fact* şi deftemplate *initial-fact* ca pe oricare altă construcţie definită de utilizator. Insă dacă se va da comanda *reset* de această dată *(initial-fact)* nu va mai fi adăugat la lista de fapte. In acest caz dacă avem reguli care nu au condiţii în LHS, acestea nu se mai aprind şi programul poate funcţiona incorect. Pentru a reintroduce acest fapt în lista de fapte este necesară execuţia comenzii *clear* şi reîncărcarea bazei de cunoştinţe.

Se poate folosi ca şi la comanda retract simbolul * ca argument, în acest caz fiind şterse toate construcţiile de acel tip. De exemplu (**undefrule ***) şterge toate construcţiile de tip defrule.

Trebuie reţinut şi faptul ca toate construcţiile ce sunt referite (cerute) de alte construcţii nu pot fi şterse. După cum se va vedea în următorul exemplu construcţia deftemplate *initial-fact* nu poate fi ştearsă până când construcţia deffacts *initial-fact* nu este ştearsă. De asemenea va trebui să fie ştearsă şi construcţia defrule *exemplu*, care utilizează faptul *initial-fact* în LHS pattern.

```
CLIPS> (defrule exemplu
         =>  )                          ;cea mai simplă regulă
CLIPS> (undeftemplate initial-fact)
Unable to delete deftemplate initial-fact while
outstanding references to it still exist.
CLIPS> (undeffacts initial-fact)
CLIPS> (undeftemplate initial-fact)
Unable to delete deftemplate initial-fact while
outstanding references to it still exist.
CLIPS> (undefrule exemplu)
CLIPS> (undeftemplate initial-fact)
```

Dacă mai există construcţii ce utilizează *initial-fact* (sau sunt cerute de alte construcţii) ar fi fost necesară mai întâi ştergerea acestora.

1.7.5. Ştergerea tuturor construcţiilor din mediul CLIPS

După cum s-a mai văzut comanda **clear** este cea care şterge toată informaţia din mediul CLIPS. Sunt şterse toate construcţiile şi toate faptele din lista de fapte. Sintaxa comenzii *clear* este următoarea :

```
(clear)
```

După ştergerea mediului CLIPS, comanda *clear* adaugă şi construcţia *initial-fact*:

```
CLIPS> (clear)
CLIPS> (list-deftemplates)
initial-fact
For a total of 1 deftemplates.
```

```
CLIPS> (list-deffacts)
Initial-fact
For a total of  1 deffacts.
```

1.8. Comanda printout

Comanda **printout** trimite spre un dispozitiv ataşat unui nume logic un şir de caractere. Numele logic trebuie în prealabil specificat pentru ca dispozitivul spre care se trimite să fie pregătit (de exemplu un fişier trebuie mai întâi deschis). Pentru a avea ieşirea spre **stdout** (ieşirea standard-în general monitorul) se foloseşte numele logic **t**. Sintaxa comenzii *printout* este următoarea :

(printout <logical-name> <expression>*)

Această comandă îşi dovedeşte utilitatea atunci când este amplasată în RHS-ul regulilor cu scopul de a se afişa mesaje. Orice număr de expresii poate fi amplasat într-o comandă *printout* pentru a fi afişate. Fiecare expresie este evaluată în vederea eliminării spațiilor adiționale aflate între expresiile ce se doresc a fi afişate. Simbolul **crlf** utilizat ca o <expresie> poate fi amplasat oriunde în lista de expresii, el forțând carriage return / new line (salt la început de linie nouă). Funcția *printout* îşi marchează şirurile pe care le afişează încadrându-le între ghilimele. Pot fi afişate de asemenea şi adresele de fapt, adresele de instanță şi adresele externe. Această funcție nu returnează nici o valoare. Dispozitivul spre care se trimite poate fi redefinit, astfel încât se pot transmite informații spre un modem sau o imprimantă. Următoarea regulă demonstrează utilitatea comenzii *printout* :

```
(defrule cartela_telefonica
        (card    yes)
        =>
        (printout  t  "Puteti forma numărul de telefon…"  crlf) )
```

In aceasta regulă prezența comenzii *printout* în RHS-ul regulii duce la afişarea spre terminal (standard output device) a unui şir de caractere. Singurul argument al funcției este şirul:

"Puteti forma numărul de telefon…"

care va fi afişat pe ecran fără ghilimele. Dacă s-ar fi dorit trimiterea şirului spre un fişier, ar fi trebuit mai întâi deschis fişierul cu funcția **open**.

1.9. Intrebări:

1. Care sunt tipurile de date fundamentale folosite în CLIPS ?
2. Clips-ul este **case-sensitive**? Face acesta diferență între simbolurile:
   abc aBc Abc ABC

3. Care sunt construcțiile de definiție pe care îi puteți folosi în CLIPS ?
4. De câte tipuri sunt faptele în CLIPS ? Dați exemple pentru fiecare tip.
5. Care este diferența între un slot şi un multislot?
6. Care este sintaxa constructorului **deftemplate**? Definiți o astfel de construcție.
7. Enumerați câteva simboluri rezervate pe care nu le putem folosi în CLIPS pentru numele unui fapt.
8. Cu ajutorul căror comenzi putem aserta şi şterge faptele. Dați un mic exemplu de utilizare a acestor funcții.
9. Ce face comanda **facts** şi care este semnificația fiecărui parametru al acesteia ?
10. Care sunt diferențele obținute la execuțiile comenzilor **modify** şi **duplicate**.
11. Operațiile de modificare sau de duplicare pot fi aplicate doar pentru fapte ordonate, fapte de tip şablon (deftemplate) sau pentru ambele tipuri? De ce?
12. Care sunt avantajele folosirii în cadrul unui program a construcției **deffacts**. În cazul în care avem definite astfel de construcții într-un program ce se întâmplă când este lansată în execuție comanda **reset** ?
13. Intr-un program putem defini doar o singură construcție deffacts sau putem defini mai multe astfel construcții?
14. Care este sintaxa folosită la definirea unei reguli? Exemplificați.
15. Care este efectul comenzii **refresh** într-un program? Când este utilă folosirea acestei comenzi?
16. Cu ajutorul cărei comenzi putem rula un program pas cu pas?
17. Cum putem opri execuția unui program la execuția unei reguli specificate?
18. Care sunt comenzile folosite pentru afişarea listei membrilor unei construcții? Dar cele de afişare a textului unui constructor? Cum putem şterge o construcție pe care am definit-o într-un program? Care este efectul folosirii asteriscului ***** ca parametru în cadrul acestor comenzi?
19. Care este efectul comenzii **clear** într-un program?
20. Care sunt avantajele pe care le oferă folosirea comenzilor **load** şi **save** , în comparație cu scrierea unui program de la prompter.
21. Folosind funcția **printout** afişați pe ecran mesajul :
    Va urez " Bafta multa ! " la examene.

1.10. Aplicații:

1. Convertiți următoarele propoziții în fapte, pe care apoi le veți grupa în construcții **deffacts**. Pentru fiecare grup de fapte definiți un **deftemplate** care să descrie relații cât mai generale pentru fiecare atribut.

Tatal lui Viorel este Iancu.
Mama lui Viorel este Maria.
Parinții lui Viorel sunt Iancu şi Maria.
Iancu este tata. Maria este mama. Viorel este fiu.
Iancu este bărbat. Maria este femeie. Viorel este bărbat.

2. Realizați pentru faptele de mai sus cu ajutorul construcției **defrule**, reguli care să verifice relațiile de rudenie între persoane sau dacă avem asociat un atribut pentru o persoană (de ex. atributul de fiu).

3. Construiți o bază de fapte despre colegi, sau despre cărțile dintr-o bibliotecă etc. cu un număr de minim 5 înregistrări după ce în prealabil a fost definită o construcție **deftemplate** care să conțină atributele pe care le vom asocia fiecărui fapt introdus. Realizați reguli care să verifice existența unei înregistrări.

4. Să se modifice construcția **deftemplate** de mai sus în sensul ca toate sloturile exceptând numele, vor fi înlocuite cu multisloturi. Avantajul care s-ar obține, ar putea fi simplificarea definiției construcției. De exemplu:

```
(deftemplate persoana
        (slot  nume)
        (slot  prenume)
        (slot  zi_nasterii)
        (slot  luna_nasterii)
        (slot  an_nasterii) )
```

```
(deftemplate persoana
        (multislot  nume)
        (multislot  data_nasterii) )
```

```
(persoana (nume Vlad)
        (prenume Irina)
        (zi_nasterii 21)
        (luna_nasterii 8)
        (an_nasterii 1975) )
```

```
(persoana (nume Vlad Irina)
        (data nasterii  21  8  1975 ) )
```