

# 1. CLIPS - Introducere

CLIPS - C Language Integrated Production System.

Este un mediu de dezvoltare al sistemelor expert realizat de către Departamentul Software ("Software Technology Branch, NASA/Lyndon B. Johnson Space Center") al NASA în anul 1986.

CLIPS permite dezvoltarea de software de modelare a cunoștințelor și expertizei umane. CLIPS este numit mediu de dezvoltare a sistemelor expert deoarece are facilități de creare și editare a bazelor de cunoștințe folosind un editor de text standard, de salvare a bazei de cunoștințe într-unul sau mai multe fișiere text și de încărcare și rulare a acesteia.

## Cuprins

1. CLIPS - Introducere .....	1
1.1 Ghid de programare în mediul CLIPS .....	9
1.2 Elementele de bază ale mediului CLIPS .....	12
1.2.1 Tipurile primitive de date .....	13
1.2.2 Funcții .....	17
1.2.3 Constructori .....	19
1.3 Abstractizarea datelor .....	20
1.3.1 Fapte .....	21
1.3.2 Obiecte .....	27
1.3.3 Variabile globale .....	29
1.4 Reprezentarea cunoștințelor procedurale .....	30
1.4.1 Reguli .....	30
1.4.2 Funcții .....	34
2. Funcții definite în mediul CLIPS .....	36
2.1 Funcții de interacțiune cu mediul .....	37
2.2 Funcții matematice .....	42
2.3 Structuri de control .....	53
2.4 Funcții trigonometrice .....	56
2.5 Funcții de intrare-ieșire .....	59
3. Reprezentarea cunoștințelor - Fapte .....	65
3.1 Fapte ordonate .....	70
3.1.1 Adăugarea de fapte .....	71
3.1.2 Afișarea faptelor .....	81
3.1.3 Ștergerea faptelor .....	84
3.1.4 Alte funcții pentru lucrul cu faptele .....	87
3.1.5 Urmărirea faptelor .....	89
3.2 Fapte neordonate .....	93
3.2.1 Adăugarea de fapte .....	104
3.2.2 Afișarea faptelor .....	108
3.2.3 Ștergerea faptelor .....	110
3.2.4 Alte funcții pentru lucrul cu faptele .....	111
4. Reprezentarea cunoștințelor - Reguli .....	116
4.1 Definirea regulilor .....	119

4.2	Declararea proprietăților unei reguli .....	124
4.3	Sintaxa părții condiționale ale unei reguli.....	129
4.3.1	Elementul condițional PATTERN .....	132
4.3.2	Elementul condițional TEST.....	148
4.3.3	Elementul condițional OR.....	150
4.3.4	Elementul condițional AND.....	154
4.3.5	Elementul condițional NOT .....	156
4.3.6	Elementul condițional EXISTS.....	159
4.3.7	Elementul condițional FORALL.....	163
4.3.8	Elementul condițional LOGICAL.....	166
4.4	Afișarea definiției unei reguli .....	173
5.	Variabile în mediul CLIPS.....	174
5.1	Variabile locale .....	176
5.2	Variabile multivaloare.....	186
5.2.1	Crearea de valori multicâmp .....	187
5.2.2	Specificarea unui element al unui multicâmp .....	189
5.2.3	Căutarea unui element al unui multicâmp.....	190
5.2.4	Ștergerea unui element al unui multicâmp.....	191
5.2.5	Extragerea unei secvențe de câmpuri.....	193
5.2.6	Înlocuirea unui câmp.....	194
5.2.7	Introducerea unui câmp.....	196
5.3	Variabile globale .....	198
6.	Strategii de rezolvare a conflictelor .....	203
6.1	Strategia depth.....	207
6.2	Strategia breadth .....	208
6.3	Strategia simplicity .....	208
6.4	Strategia complexity .....	209
6.5	Strategia lex.....	209
6.6	Strategia mea.....	210
6.7	Strategia random .....	212
7.	Funcții definite de utilizator în mediul CLIPS.....	213
8.	Aplicații realizate în mediul CLIPS .....	220
	Aplicația 1 .....	221
	Aplicația 2 .....	226

Aplicația 3 .....	231
Aplicație 4 .....	235
Aplicație 5 .....	239
Aplicație 6 .....	242
Aplicația 7 .....	249
Aplicația 8 .....	261
Probleme propuse .....	268

Datorită portabilității (Ms-Dos, Windows, Unix/Linux, Macintosh), popularității (codul sursă este distribuit gratuit) și performanțelor, mediul CLIPS a fost acceptat pe scară largă la implementarea tehnologiei sistemelor expert atât în sectoarele publice, cât și în cele private pentru o gamă largă de aplicații pe medii hardware diverse.

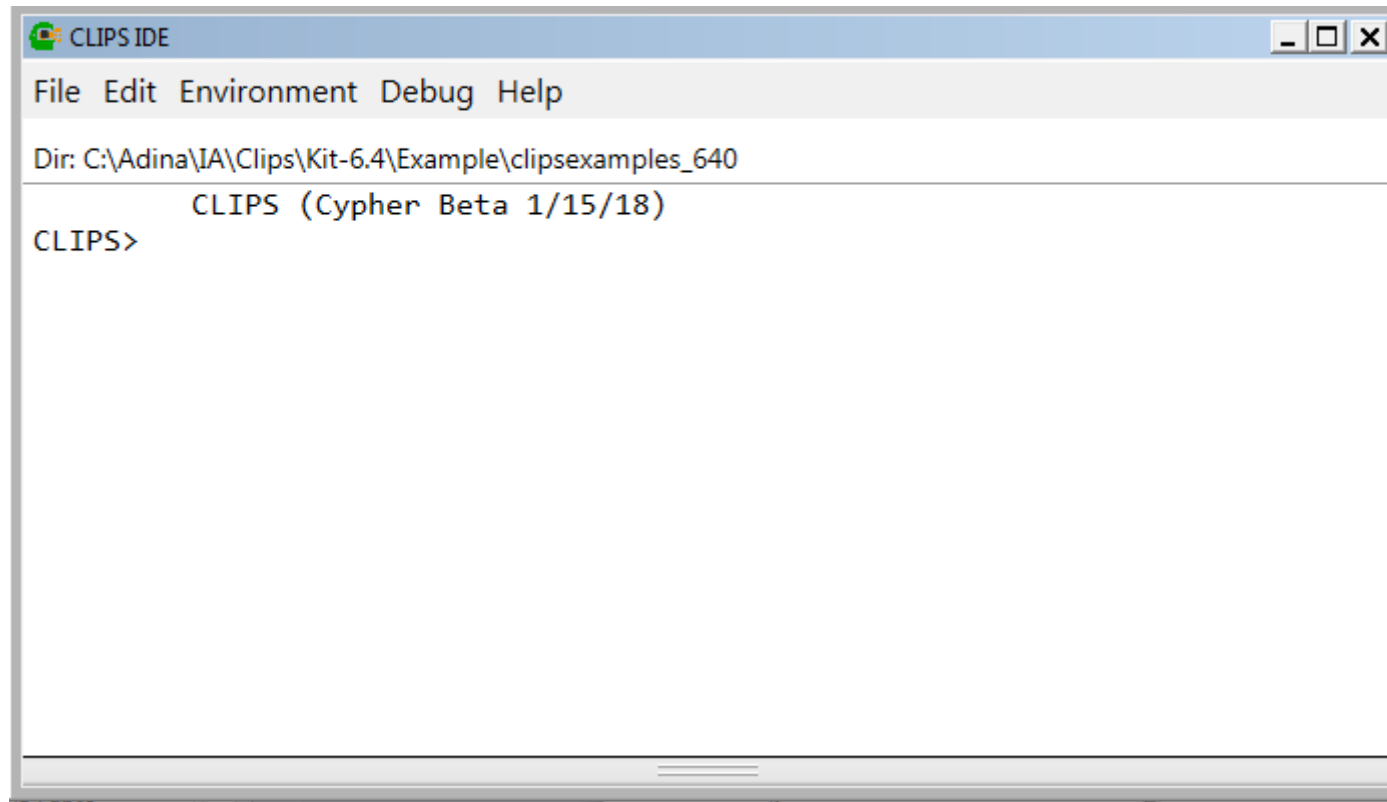
Un avantaj al acestui mediu de dezvoltare este că asigură integrarea cu limbaje de programare, cum ar fi C, Java, Visual și Ada.

Mediul CLIPS pune la dispoziția utilizatorului elementele de baza pentru dezvoltarea unui sistem expert:

- Baza de cunoștințe declarative:
  - liste de fapte
  - liste de instanțe;
- Baza de cunoștințe procedurale
  - baza de reguli;
- Motorul de inferență, care controlează execuția regulilor.

Un program în CLIPS este format din **reguli, fapte și obiecte**. Motorul de inferență decide care reguli vor fi executate și momentul când vor fi executate. Un sistem expert bazat pe reguli este un program în care faptele și eventual, obiectele, sunt datele care asigură execuția prin motorul de inferență.

Interfața mediului CLIPS este simplă, interactivă, permite introducerea de comenzi la prompt sau executarea comenzilor din meniurile puse la dispoziție. Prompt-ul mediului CLIPS este "CLIPS>". Acest prompt apare la lansarea în execuție a mediului CLIPS într-o fereastră de dialog.



Codul sursă al CLIPS este în limbajul C, astfel încât toate comenzile, procedurile și structurile sunt tratate ca funcții. Din acest motiv toate apelurile date la prompt trebuie incluse între paranteze. Mediul CLIPS este un limbaj funcțional.



## ***1.1 Ghid de programare în mediul CLIPS***

Un sistem expert realizat în CLIPS poate fi executat în trei moduri:

- interactiv, introducând comenzi de la prompt în mod text;
- folosind meniuri din fereastra grafică a Clips-ului;
- se proiectează un sistem expert încapsulat în care utilizatorul asigură un program principal și controlează execuția sistemului expert;

## Sintaxa

<string>	reprezintă o entitate simplă ce denumește o clasă sau un nume dat de utilizator. Aceste stringuri se numesc termeni sau simboluri ne-terminale.
*	când urmează un termen reprezintă zero sau mai multe entități de tipul respectiv. De exemplu: <string>* arată că se pot folosi mai multe entități <string> sau nici una.
+	când urmează un termen reprezintă una sau mai multe entități de tipul respectiv. De exemplu: <string>+ arată că se poate folosi cel puțin o entitate.
...	specifică mai multe entități.
	indică posibilitatea de a alege dintre mai mulți termeni
[<comment>]	un termen inclus între paranteze pătrate este opțional.

::=	este folosit pentru a arăta cu ce poate fi înlocuit un simbol neterminal. De exemplu următoarea sintaxă indică faptul că <lexeme> poate fi înlocuit cu <symbol> sau cu <string>: <lexeme>::=<symbol>   <string>
-----	---

## ***1.2 Elementele de bază ale mediului CLIPS***

- tipuri primitive de date;
- funcții pentru manipularea datelor;
- constructori pentru adăugarea la baza de cunoștințe;

## 1.2.1 Tipurile primitive de date

Tipuri primitive de date sunt:

1) *Integer*

2) *Float*

Un număr poate fi memorat ca float sau ca integer. Orice număr format din semn și cifre este memorat ca integer (reprezentat intern ca long integer din limbajul C).

3) *Simbol* – este alcătuit dintr-o secvență de caractere care începe cu un caracter ASCII printabil și este urmat de zero sau mai multe caractere ASCII printabile. Simbolul se termină la întâlnirea oricărui caracter printabil (spațiu, tab, return, line feed, ghilimele, paranteze (), &, |, <, ~, ;). Un simbol nu poate începe cu ? sau \$?, care sunt rezervate pentru variabile.

Observație: caracterul ; este utilizat și la începutul oricărui comentariu.

Clips este „**case sensitive**” (face distincție între literele mici si literele mari).

Exemplu:

buna	Hello	C32-GJ7	index_start
549D	653-03-439	% @=+-	3stari

4) *String* – este un set de caractere care este inclus între ghilimele ”,,. Pentru a include ,, într-un string acestea trebuie să fie precedate de backslash \.

Exemplu:

"buna"	"a and b"	"1 număr"	"a\"quote"
--------	-----------	-----------	------------

5) *Externaladdress* – este o adresă a unei structuri de date externă, returnată de o funcție. Acest tip de date poate fi creat doar prin apelul unei funcții. Reprezentarea în CLIPS a unei adrese externe este:

<Pointer-XXXX> unde XXXX este adresa externă.

6) *Factaddress* – specifică adresa unui fapt.

Un fapt este o listă de valori ale unor atomi (câmpuri) la care se face referință prin poziție (la faptele ordonate) sau prin nume (la faptele neordonate). Faptele sunt referite prin index sau adresă. Reprezentarea unei adrese a unui fapt este:

<Fact-XXXX> unde XXXX este indexul unui fapt.

7) *Instanța* – este o instanțiere a unui obiect al unei clase. În Clips obiectele pot fi definite ca float, integer, symbol, string, valori multicâmp (asemănătoare cu un vector ale cărui elemente pot avea

tipuri diferite), adrese externe, adrese de fapte sau instanțe ale unor clase definite de utilizator. Un nume al unei instanțe se formează prin includerea unei simbol în paranteze pătrate.

[stare-1]      [nume]      [+++]      [1473-390]

Adresa unei instanțe este de forma <Instance-XXXX> unde XXXX este numele unei instanțe.



## 1.2.2 Funcții

O funcție în Clips este o secvență de cod executabil identificată printr-un nume și care întoarce o valoare sau are ca rezultat un efect vizibil.

Tipuri de funcții:

- ale sistemului, scrise într-un alt limbaj (C) și introduse în mediul CLIPS (funcții interne ale mediului);
- definite de utilizator într-un alt limbaj (funcții externe) sau în mediul CLIPS cu ajutorul constructorului `deffunction`;
- generice definite prin constructorul `defgeneric` sau `defmethod`, și care permit executarea codului dependent de argumentele transmise spre funcțiile generice. Astfel se pot supradefini funcțiile;
- funcționale la care argumentele apar întotdeauna după numele funcției.

Exemplu: (+ 3 4 5) sau (+ 3 ( \* 4 8) 9)

O expresie este o funcție ale cărei argumente sunt specificate.

## 1.2.3 Constructori

**defmodule, defrule, deffacts, deftemplate, defglobal, deffunction, defclass, definstances, defmessagehandler, defgeneric, defmethod.**

La apelare toți constructorii trebuie să fie incluși în paranteze. Diferența dintre constructori și funcții este că apelul unei funcții nu modifică mediul CLIPS, pe când un constructor adaugă noi elemente în baza de cunoștințe a mediului CLIPS și nu returnează nici o valoare.

## **1.3 Abstractizarea datelor**

Pentru reprezentarea informațiilor în Clips se folosesc trei forme primare:

- faptele,
- obiectele
- variabilele globale.

Valorile și instanțele acestora la un moment dat formează memoria de lucru a mediului CLIPS.

## 1.3.1 Fapte

Faptele reprezintă forma de bază pentru reprezentarea cunoștințelor în Clips. Fiecare fapt reprezintă o piesă de informație și este plasat într-o listă de fapte. Faptele reprezintă unitatea fundamentală de date folosită de reguli.

Operațiunile ce se pot efectua asupra faptelor sunt:

- *adăugarea* de noi fapte la lista de fapte;
- *ștergerea* de fapte din lista de fapte;
- *modificarea* unui fapt;
- *duplicarea* unui fapt;

Toate aceste operații pot fi făcute direct, în linia de comandă a mediului sau printr-un program.

Numărul de fapte din lista de fapte și cantitatea de informație ce poate fi reținută de un fapt sunt limitate doar de capacitatea de memorie a computerului.

Dacă se încearcă introducerea unui fapt în lista de fapte și dacă acest fapt deja există în listă, atunci noul fapt introdus va fi ignorat. Există o constantă de mediu care permite introducerea unui nou fapt cu același conținut de informație, dar cu un nou index.

Un fapt poate fi specificat prin: adresă sau prin index.

Când un fapt este adăugat la lista de fapte sau modificat primește un index unic în lista de fapte. Indexul de fapte începe de la zero și se incrementează cu 1 la fiecare nou fapt adăugat sau modificat. Când se apelează comenzile reset sau clear indexul de fapte este pus la zero.

Adresa unui fapt se poate obține prin valoarea returnată la apelul unei comenzi assert, modify, duplicate sau printr-o variabilă legată cu adresa unui fapt care îndeplinește o condiție din partea stângă a unei reguli.

Faptele pot fi:

a) *ordonate*, constau într-un simbol urmat de o secvență de zero sau mai multe câmpuri separate de spații și delimitate de paranteze (). În general primul câmp al unui fapt ordonat specifică o relație între celelalte câmpuri ale faptului.

Exemple:

(numele este Mihai)

(altitudinea este 10000 metri)

(lista-mic-dejun paine lapte oua )

Câmpurile dintr-un fapt ordonat pot avea orice tip primitiv de date, cu excepția primului câmp care trebuie să fie un simbol. Faptele ordonate codifică informația după poziție. Pentru ca un utilizator să aibă acces la cunoștințe trebuie să cunoască nu numai ce date sunt memorate în acel fapt, dar și câmpul care conține datele.



b) *neordonate* constau dintr-o structură abstractă în care fiecare câmp are un nume. Pentru a crea fapte neordonate se folosește constructorul **deftemplate**. Primul câmp al unui fapt neordonat corespunde numelui faptului apoi este urmat de zero sau mai multe sloturi. Câmpurile dintr-un fapt neordonat se numesc sloturi și pentru ele se poate defini tipul de date pe care-l conțin, valoarea, și eventual domeniul de valori, și o valoare implicită. Sloturile dintr-un fapt neordonat pot avea orice ordine de așezare.

Exemplu:

```
(client (nume "Ion Popescu") (id 234652))  
(punct (abscisa 100) (ordonata -200))  
(grupa (profesor "Maria Ionescu") (numar-  
elevi 30) (sala "Y300"))  
(meniu (numar-produse 3) (produse paine lapte  
oua) )
```

c) *fapte inițiale* cu ajutorul constructorului **deffacts** se pot introduce în lista de fapte o colecție de fapte folosind comanda reset.

## 1.3.2 Obiecte

Obiecte sunt definite cu ajutorul tipurilor primitive de date simbol, string, numere întregi și reale, valori multicâmp, adrese externe sau instanțe ale unor clase definite de utilizator. Obiectele sunt descrise prin proprietăți și comportament. O clasă este un șablon pentru proprietăți și comportamentul comun al obiectelor ce sunt instanțe ale clasei.

În Clips obiectele fac parte din două categorii:

- tipuri primitive;
- instanțe ale unor clase definite de utilizator;

Aceste două categorii de obiecte diferă prin modul în care sunt referite, create și șterse.

Obiectele de tipul primitiv sunt referite simplu prin specificarea valorii și sunt create și șterse implicit de mediul CLIPS atunci când este nevoie. Aceste obiecte nu au nume și nici sloturi, iar clasele lor sunt predefinite în CLIPS.

O instanță a unei clase definită de utilizator este referită prin nume sau adresă și sunt create și șterse explicit prin mesaje și funcții speciale. Proprietățile unei astfel de instanțe sunt date printr-un set de sloturi care au nume și pot avea valori singulare sau multicâmp. Comportamentul unui obiect este specificat prin cod procedural numit „messagehandlers”, care este atașat obiectelor clasei. Toate instanțele unei clase au aceleași sloturi, dar fiecare instanță poate avea valori diferite ale acestor sloturi. Principala diferență dintre sloturile unui obiect și cele ale unui structuri definită cu ajutorul constructorului

deftemplate (fapt neordonat) este noțiunea de moștenire. În CLIPS se permite moștenirea multiplă.

### 1.3.3 Variabile globale

Variabile globale pot fi definite cu ajutorul constructorului **defglobal** și au avantajul că pot fi accesate oriunde în mediul CLIPS și valorile lor sunt independente de apelul altor constructori.

Există și constructori (cum ar fi defrule, deffunction) care permit definirea unor variabile locale care pot fi referite doar în constructorul respectiv. Variabilele globale din CLIPS sunt asemănătoare cu cele din limbajele procedurale C, Ada, dar spre deosebire de acestea pot conține valori care au mai multe tipuri de date.

## ***1.4 Reprezentarea cunoștințelor procedurale***

### **1.4.1 Reguli**

Regulile sunt folosite pentru a reprezenta cunoștințe euristice, asigurând realizarea unui set de acțiuni în condiții date, verificabile. Dezvoltarea unui sistem expert constă în definirea unui set de reguli care vor duce la rezolvarea problemei. O regulă seamănă cu secvența procedurală IF – THEN dintr-un limbaj de programare.

O regulă este alcătuită din:

- **antecedent** (parte condițională, LHS-left hand side – parte stângă, engl. if)
- **consecvent** (parte de acțiuni, RHS-right hand side – parte dreaptă, engl. then).

**Antecedentul** conține un set de condiții (elemente condiționale) care trebuie să fie satisfăcute pentru ca regula să fie activată (să fie aplicabilă). În CLIPS, partea condițională a unei reguli este satisfăcută dacă există sau nu anumite fapte în lista de fapte sau o anumită instanță a unei clase definite de utilizator sau este îndeplinită o condiție testată explicit. O condiție din antecedentul unei reguli se numește „pattern”. Procesul prin care se verifică faptele și instanțele care satisfac o anumită condiție (pattern) se numește „**pattern-matching**” (verificare). Mediul CLIPS furnizează un mecanism, numit motor de inferență, care compară șabloanele cu starea curentă a listei de fapte și a listei de instanțe, în mod automat, pentru a determina regulile care au partea condițională satisfăcută și deci, sunt aplicabile și care sunt trecute într-o listă numită *agendă*. În agendă pot fi zero sau mai multe reguli activate la un moment dat.

**Consecventul** unei reguli este un set de acțiuni care sunt executate la aplicarea regulii. Acțiunile sunt realizate atunci când motorul de inferență execută o regulă din cele aplicabile. Dacă sunt aplicabile mai multe reguli la un moment dat, motorul de inferență folosește o *strategie de rezolvare* a conflictelor care alege o singură regulă. Acest proces se termină atunci când nu mai există nici o regulă în agendă sau când s-a specificat explicit, printr-o comandă, oprirea inferenței (agenda conține toate regulile aplicabile la un moment dat).



Procesul inferențial prin care motorul de inferență caută regulile aplicabile și le execută este partea cea mai importantă a unui sistem expert (comanda de apel a motorului inferențial al mediului CLIPS este (run)). Procesul funcționează ciclic astfel:

1. se determină regulile aplicabile (care au partea condițională satisfăcută, adică toate elementele condiționale sunt adevărate);
2. dacă nu există nici o astfel de regulă (aplicabilă), atunci procesul de oprește;
3. dacă există mai multe reguli aplicabile, motorul de inferență folosește o strategie de rezolvare a conflictelor, pentru a selecta regula a cărei acțiuni vor fi executate;
4. execută acțiunile corespunzătoare regulii selectate;
5. reia pasul 1.

## 1.4.2 Funcții

Pot include structuri de control: if...then...else, while, loop-for-count, progn, progn\$, return, breack, switch.

Tipuri de funcții:

- **deffunction** – permite declararea unor noi funcții. Corpul unei funcții este un set de acțiuni executate secvențial la apelul funcției. Valoarea returnată de funcție este valoarea ultimei expresii evaluate în corpul funcției.
- **funcții generice** sunt similare cu primele, dar au avantajul că pot fi supraîncărcate.
- **mesajele atașate obiectelor** – comportamentul obiectelor este dat de o parte de cod procedural numit „messagehandlers”. Obiectele sunt manipulate cu ajutorul mesajelor, care sunt asemănătoare funcțiilor din CLIPS și care returnează o valoare sau au un anumit efect.

- **defmodules** – permit modularizarea bazei de cunoștințe. Fiecare constructor trebuie să facă parte dintr-un modul. Vizibilitatea unui constructor definit într-un modul pentru un alt modul poate fi setată de utilizator. La fel se poate seta și vizibilitatea faptelor și a instanțelor. Modulele pot fi folosite și pentru controlul execuției regulilor.

## 2. Funcții definite în mediul CLIPS

În mediul CLIPS apelul unei funcții se face în notația prefixată – argumentele funcției (separate de spații) sunt trecute întotdeauna după numele funcției. Argumentele unei funcții pot fi tipuri primitive de date, variabile, sau alte funcții.

## ***2.1 Funcții de interacțiune cu mediul***

- **load**

`(load <nume-fișier>)`

Încarcă constructori din fișierul specificat ca argument (împreună cu calea de directoare).

- **bload**

`(bload <nume-fișier>)`

Încarcă un fișier binar, produs anterior cu comanda `bsave`.

- **batch**

`(batch <nume-fișier>)`

Execută comenzi din fișierul specificat ca argument. Dacă reușește, întoarce valoarea ultimei expresii din fișier.

- **agenda**

(agenda)

Afișează o listă de reguli aplicabile la un moment dat. Dacă nu se specifică nici un argument, sunt afișate activările din modulul curent. Dacă se specifică un nume de modul, se afișează numai activările din acel modul. Dacă se specifică “\*”, sunt afișate toate activările.

- **clear**

(clear)

Șterge mediul CLIPS.

- **exit**

(exit)

Oprește și închide mediul CLIPS.

- **halt**

`(halt)`

Oprește execuția unei reguli. Nu are efect dacă nu este apelată în consecventul unei reguli.

- **reset**

`(reset)`

Șterge toate faptele din lista de fapte, toate regulile aplicabile din agendă, instanțele. Apoi introduce faptul inițial (`initial-fact`) în lista de fapte, toate faptele definite cu ajutorul constructorului `deffacts`, introduce instanțele definite de constructorul `definstance`, și (dacă proprietatea `set-reset-globals` este `TRUE`) inițializează la valorile declarate inițial toate variabilele globale.

- **run**

(run [`<întreg>`])

Pornește motorul de inferență. Dacă nu este specificat nici un argument, atunci el are implicit valoarea `-1`, și CLIPS rulează până când nu mai rămân activări sau până când întâlnește o comandă `halt`. Dacă se specifică un argument, acesta va da numărul maxim de reguli de executat înainte de oprire. Următoarea comanda `run` va continua execuția din locul de unde s-a oprit anterior. Comanda `(run 1)` este echivalentă cu rularea `pas` cu `pas`.



- **set-reset-globals**

```
(set-reset-globals (TRUE | FALSE | nil))
```

Schimbă setarea curentă a constantei de mediu cu referire la variabilele globale la apelul comenzii reset. Dacă constanta este setată TRUE, comanda reset reinițializează valorile variabilelor globale la valorile lor inițiale. Dacă proprietatea este setată FALSE sau nil, comanda nu va afecta valorile variabilelor globale.

## 2.2 Funcții matematice

- +

(+ <expresie-numerică> <expresie-numerică>+)

Întoarce suma primului argument cu următoarele argumente.

- -

(- <expresie-numerică> <expresie-numerică>+)

Întoarce primul argument minus argumentele următoare.

- \*

(\* <expresie-numerică> <expresie-numerică>+)

Întoarce produsul argumentelor.

- /  
(/ <expresie-numerică> <expresie-numerică>+)  
Întoarce primul argument împărțit la argumentele următoare.
- \*\*  
(\* \* <expresie-numerică> <expresie-numerică>)  
Ridică primul argument la puterea celui de-al doilea argument.
- <, >  
(< <expresie-numerică> <expresie-numerică>+)  
Întoarce TRUE dacă fiecare argument este mai mic (mai mare) în valoare decât argumentul următor. Altfel, întoarce FALSE.

- `<=, =>`

`(<= <expresie-numerică> <expresie-numerică>+)`

Întoarce TRUE dacă valoarea fiecărui argument este mai mică (mai mare) sau egală cu valoarea argumentului următor. Altfel, întoarce FALSE.

- `<>`

`(<> <expresie-numerică> <expresie-numerică>+)`

Întoarce TRUE dacă valoarea primului argument nu este egală cu valorile argumentelor următoare. În caz contrar, întoarce FALSE. Se aplică numai pentru compararea expresiilor numerice.

- **=**

(= <expresie-numerică> <expresie-numerică>+)

Întoarce TRUE dacă valoarea primului argument este egală cu valorile argumentelor următoare. Altfel întoarce FALSE. Se aplică numai pentru compararea expresiilor numerice.

- **abs**

(abs <expresie-numerică>)

Întoarce valoarea absolută a argumentului său.

- **div**

(div <expresie-numerică> <expresie-numerică>+)

Întoarce primul argument împărțit la argumentele următoare folosind împărțirea întreagă. Câtul valorilor celor două expresii numerice este rotunjit la cel mai apropiat întreg.

- **exp**

(exp <expresie-numerică>)

Ridică numărul *e* la puterea indicată de argument.

- **float**

(float <expresie-numerică>)

Convertește argumentul într-un număr real de tip float.

- **integer**

`(integer <expresie-numerică>)`

Convertește argumentul într-un număr întreg. Trunchiază orice parte fracționară a valorii expresiei numerice date și întoarce partea întreagă.

- **log**

`(log <expresie-numerică>)`

Întoarce logaritmul natural al argumentului.

- **log10**

`(log10 <expresie-numerică>)`

Întoarce logaritmul zecimal al argumentului.

- **max**

(max <expresie-numerică>+)

Întoarce valoarea celui mai mare argument numeric.

- **min**

(min <expresie-numerică>+)

Întoarce valoarea celui mai mic argument numeric.

- **mod**

(mod <expresie-numerică> <expresie-numerică>)

Întoarce restul împărțirii primului argument la cel de-al doilea.

- **pi**

(pi)

Întoarce constanta *pi*.



- **random**

(random)

Întoarce aleator un întreg cu o valoare în intervalul 0 și 65536.

- **round**

(round <expresie-numerică>)

Rotunjește argumentul către cel mai apropiat întreg.

- **sqrt**

(sqrt <expresie-numerică>)

Întoarce rădăcina pătrată a argumentului.

- **and**

(and <expresie>+)

Întoarce TRUE dacă toata argumentele sunt evaluate la o valoare non-FALSE.

- **or**

(or <expresie>+)

Întoarce TRUE dacă oricare dintre argumente este evaluat la o valoare non-FALSE.

- **eq**

(eq <expresie> <expresie>+)

Întoarce TRUE dacă primul argument este egal ca tip și valoare cu toate celelalte argumente.

- **eq\***

(eq\* <expresie> <expresie>+)

Întoarce TRUE dacă primul argument este echivalent cu toate celelalte argumente.

- **neq**

(neq <expresie> <expresie>+)

Întoarce TRUE dacă primul argument nu este egal ca tip și valoare cu argumentele următoare.

- **lexemep**

(lexemep <expresie>)

Întoarce TRUE dacă expresia este de tip simbol sau string.

- **Numberp (symbolp, stringp, integerp, floatp)**

(numberp <expresie>)

Întoarce TRUE dacă expresia este de tip integer sau float (simbol, string, integer, float).

## 2.3 Structuri de control

- **if**

```
(if <expresie> then <acțiune>* [else  
<acțiune>*])
```

Permite execuția condițională a unui grup de acțiuni. Se evaluează expresia. Dacă nu este evaluată ca FALSE, se execută prima listă de acțiuni (corespunzătoare ramurii then). Dacă este evaluată FALSE, se execută cea de-a doua listă de acțiuni (corespunzătoare ramurii else – dacă aceasta este specificată). Se returnează valoarea ultimei expresii evaluate.

- **return**

`(return [<expresie>])`

Dacă este întâlnită într-o funcție definită cu ajutorul constructorului `deffunction`, atunci întoarce valoarea dată, și încheie execuția acesteia. Dacă este definită în partea dreaptă a unei reguli, atunci termină imediat execuția regulii și scoate modulul de curent din stiva de focus.

- **while**

`(while <expresie> [do] <acțiune>*)`

Permite ciclarea condițională. Evaluează expresia în mod repetat. Atâta timp cât aceasta nu este `FALSE`, se execută acțiunea. Returnează valoarea ultimei expresii evaluate.

- **loop-for-count**

```
(loop-for-count <contor> [do] <acțiune>*)  
<contor> ::= <index-de-sfârșit> |  
              (<variabilă-contor>  
               [<index-de-început> <index-de-sfârșit>])  
<index-de-început> ::= <expresie-integer>  
<index-de-sfârșit> ::= <expresie-integer>
```

Permite ciclarea cu contor. Returnează valoarea ultimei expresii evaluate.

## 2.4 Funcții trigonometrice

acos	arccosinus
acot	arccotangent
acsc	arccosecant
asec	arcsecant
asin	arcsine
atan	arctangent
cos	cosinus
cot	cotangent
csc	cosecant
sec	secant
sin	sinus
tan	tangent

acosh	arccosinus hiperbolic
acoth	arccotangent hiperbolic
acsch	arccosecant hiperbolic
asech	arcsecant hiperbolic
asinh	arcsinus hiperbolic
atanh	arctangent hiperbolic
cosh	cosinus hiperbolic
coth	cotangent hiperbolic
csch	cosecant hiperbolic
sech	secant hiperbolic
sinh	sinus hiperbolic
tanh	tangent hiperbolic



## Exemplu: de utilizare a diverselor funcțiilor ale mediului CLIPS

```
CLIPS> (+ 3 4 )
```

```
7
```

```
CLIPS> (* 10 8 )
```

```
80
```

```
CLIPS> (* 5 (+ 3 7) (+ (** 2 3) (/ 24 2)))
```

```
1000.0
```

```
CLIPS> (eq test alt-test)
```

```
FALSE
```

```
CLIPS> (neq test alt-test)
```

```
TRUE
```

```
CLIPS> (* 5 7.8 2 )
```

```
78.0
```

```
CLIPS> (* 7 (+ 5 (* 3 4 6 ) 9  
) (+ 10 3 ))
```

```
7826
```

```
CLIPS> (acos 0.5)
1.0471975511966
CLIPS> (log10 10)
1.0
CLIPS> (>= 5 7 )
FALSE
CLIPS> (max 3 6 10 1 9 )
10
CLIPS> (sqrt 81)
9.0
```

Se observă în acest exemplu că funcțiile matematice se pot apela cu mai multe argumente. Evaluarea funcțiilor se face din interior către exterior, în cazul în care sunt prezente paranteze.

## ***2.5 Funcții de intrare-ieșire***

Funcțiile de intrare-ieșire folosesc nume de dispozitive de intrare-ieșire sau nume logice de fișiere, unde vor fi efectuate operațiile de culegere sau afișare de date. Aceste dispozitive logice pot fi implicite:

stdin – este implicit pentru toate dispozitivele standard de intrare (tastatura);

stdout – este implicit pentru toate dispozitivele standard de ieșire (de exemplu monitorul, dacă se specifică „t”);

sau explicite, definite de utilizator, asociate cu nume de fișiere.

- **printout**

`(printout <nume-logic> <expresie>* )`

Afișează expresia la dispozitivul standard de ieșire (monitorul) dacă se specifică „t” la nume-logic, sau scrie într-un fișier asociat cu numele logic printr-o operație de deschidere de fișier. Simbolul „crlf” poate fi utilizat ca o expresie și poate fi amplasat oriunde în lista de expresii, el forțând carriage return / new line (face saltul la un nou rând). Pentru a afișa un șir de caractere cu `(printout)` acesta trebuie trecut între ghilimele. Expresiile care nu sunt scrise între ghilimele sunt evaluate și rezultatul este afișat. Se pot afișa și adrese de fapte, de instanțe sau adrese externe. Această comandă este folosită de obicei, în partea de acțiune a unei reguli pentru afișarea unui mesaj pe monitor și poate conține orice număr de expresii de afișat.

- **format**

```
(format      <      nil|nume-logic>      <sablon>  
<expresie>* )
```

Formatează o expresie pentru a o scrie în fișierul sau dispozitivul asociat cu numele logic sau dacă nu se dorește scrierea se specifică „nil”. Această funcție este asemănătoare funcției „printf” din limbajul de programare C.

- **read**

```
(read [<nume-dispozitiv-logic>] )
```

Citește o valoare de la dispozitivul logic dacă este specificat.

- **readline**

`(readline [<nume-dispozitiv-logic>] )`

Citește un șir de caractere până la întâlnirea caracterului de trecere la linie nouă, sau EOF, de la dispozitivul logic, dacă acesta este specificat.

- **open**

`(open <nume-fisier> <nume-logic> [<mod>])`

`<mod> ::= "r" | "w" | "r+" | "a" | "wb"`

Deschide un fișier specificat prin nume (împreună cu calea de directoare) și îl asociază cu numele logic, care va fi folosit de alte funcții pentru a avea acces la fișier. Modul de utilizare al fișierului este specificat opțional și poate fi numai pentru citire („r” - implicit), numai pentru scriere („w”), pentru citire și pentru scriere („r+”), pentru

adăugare („a”), pentru scriere în binar („wb”). Această funcție este folosită, în general, în consecventul unei reguli.

- **close**

```
(close [<nume-logic>] )
```

Închide un fișier specificat prin nume logic, deschis anterior cu funcția open. Dacă funcția este apelată fără nici un argument, atunci sunt închise toate fișierele deschise.

Exemplu: de utilizare a funcțiilor de lucru cu fișiere

```
CLIPS>(open "test1.clp" fis1 "w")
```

```
TRUE
```

```
CLIPS>(printout t "Voi scrie in linie comanda"  
crlf)
```

```
Voi scrie in linie comanda
```

```
CLIPS>(printout fis1 "date fisier")
CLIPS>(close fis1)
TRUE
CLIPS>(open "C:\\Ex-clips\\test1.clp" fis2)
TRUE
CLIPS>(read fis2)
date
CLIPS>(read fis2)
fisier
CLIPS>(read fis2)
EOF
CLIPS>(close)
TRUE
CLIPS>(close)
FALSE
CLIPS>
```



Fișierul cu numele test1.clp a fost deschis pentru scriere și asociat cu numele logic fis1, apoi se scrie un mesaj pe ecran și sunt introduse date în fișier. După ce se închide fișierul, se redeschide pentru citire.

### **3. Reprezentarea cunoștințelor - Fapte**

Faptele sunt forma principală de abstractizare, în mediul CLIPS, a cunoștințelor declarative, structurate ca piese de cunoaștere. Pentru a se putea lucra cu piese de cunoaștere, acestea trebuie formalizate într-un anumit formalism, care apoi să permită și raționamentul cu aceste cunoștințe. Cel mai des întâlnite metode de formalizare a cunoștințelor declarative sunt logica propozițiilor și logica predicatelor de ordin unu, care au un fundament matematic bine definit.

Definirea unui element de informație se face folosind tripletul:

- Obiect – la care face referire piesa de cunoaștere.
- Atribut (proprietate) – este o funcție (caracteristică) care poate modifica valoarea sau un ansamblu de valori ale obiectului.
- Valoare – corespunde unui predicat, adică o mulțime de referințe legate de atribut.

Exemplu: de reprezentare a unui element de informație

(animal mamifer elefant)

(patrat latura 10)

(persoana (nume Popescu) (varsta 30)) – structură de date cu mai multe perechi atribut-valoare

Fiecare fapt este plasat în lista curentă de fapte (engl.: fact-list), care face parte din memoria de lucru a sistemului. Memoria de lucru a sistemului mai cuprinde lista de instanțe, lista de variabile împreună cu valorile asociate și modulele de cunoștințe.

Operațiunile care se pot efectua asupra faptelor sunt:

- introducerea de noi fapte la lista de fapte,
- ștergerea din lista de fapte,
- modificarea sau multiplicarea unui fapt existent.

Numărul de fapte din lista de fapte este nelimitat, singura limitare fiind cea impusă de capacitățile hard ale calculatorului, adică de dimensiunea memoriei.

În mediul CLIPS faptele pot fi de două tipuri:

- Ordonate
- Neordonate

Faptele pot avea unul sau mai multe câmpuri, care sunt separate între ele de delimitatori (blanchspace, tabs, carriage return – pentru faptele ordonate), (paranteze – pentru faptele neordonate).

Câmpurile unui fapt pot avea următoarele tipuri:

- float
- integer
- symbol (caracter sau șir de caractere începând cu un caracter printabil)
- string (șir de caractere)
- externaladdress
- factaddress

- instancename
- instanceaddress

Caracteristic mediului CLIPS este posibilitatea de a introduce un fapt inițial, numit "initial-fact". Acesta este adăugat în lista de fapte în urma comenzii (`reset`), care, în primul rând, șterge toate faptele din lista de fapte și apoi introduce faptul inițial cu indicele 0. Faptul inițial este folosit pentru activarea unor reguli care nu au o condiție precisă de activare.

### **3.1 Fapte ordonate**

Faptele ordonate memorează obiecte, attribute sau valori ale pieselor de cunoaștere sub forma câmpurilor. Ordinea câmpurilor este importantă, deoarece este singurul criteriu de identificare. Este bine ca ordinea dintre câmpuri să exprime o relație între acestea. De obicei, primul câmp exprimă obiectul, iar câmpurile următoare memorează attributele și/sau valorile obiectului. De exemplu, faptul (animal este zebra) nu este același cu faptul (zebra este animal). La faptele ordonate utilizatorul trebuie să cunoască ce cunoștințe sunt memorate în fapt și poziția câmpurilor care conțin acele cunoștințe.

Tipul fiecărui câmp este dat de tipul valorii memorata în acel câmp. La faptele ordonate tipul valorii unui câmp este determinat implicit.

### 3.1.1 Adăugarea de fapte

Comanda cu care se introduce un nou fapt la lista de fapte este (assert). Sintaxa comenzii este următoarea:

```
(assert <fapt>+)
```

și are ca efect adăugarea imediată a tuturor faptelor scrise între paranteze, după numele comenzii, în lista de fapte și întoarcerea indexului ultimului fapt introdus sau FALSE dacă nu s-au introdus cu succes faptele. Comanda (assert) poate fi folosită în mediul CLIPS în linia de comandă, sau în partea de acțiune a unei reguli.

Folosind constructorul (deffacts) se poate defini o listă de fapte, care va fi introdusă în lista de fapte a mediului CLIPS la execuția comenzii (reset). Sintaxa comenzii este:

```
(deffacts <nume-lista-fapte> [<comment>])
```

`<fapt-ordonat>*)`

Această comandă poate fi scrisă în linie de comandă a mediului CLIPS, sau poate fi scrisă într-un fișier CLIPS (cu extensia clp) ca o comandă de sine stătătoare. Faptele introduse prin acest constructor au același comportament cu al faptelor introduse prin comanda (assert).

O altă comandă care permite introducerea unui fapt în lista de fapte are următoarea sintaxă:

`(assert-string <expresie-șir>)`

și are ca efect convertirea expresiei șir într-un fapt, urmată de introducerea în listă. Scopul funcției este de a analiza șirul ca pe un fapt, și dacă este corect, afișează indicele faptului introdus.

Observație: primul câmp al unui fapt trebuie să fie de tip simbol sau string și este folosit de obicei, pentru a exprima o relație între celelalte



câmpuri utilizate pentru valori, asigurând astfel și documentație pentru utilizator.

### Exemplu:

```
CLIPS> (assert (masura) )
```

```
<Fact-1>
```

```
CLIPS> (assert (a 45) (test fapt) (laborator-  
clips tematica fapte reguli variabile) )
```

```
<Fact-4>
```

```
CLIPS> (assert-string "(numar real 5.6) ")
```

```
<Fact-5>
```

În acest exemplu au fost introduse în listă următoarele fapte: (masura) – care are un singur câmp, (a 45), (test fapt) – aceste fapte au două câmpuri, primul poate fi interpretat ca numele faptului, iar al doilea ca atribut, (numar real 5.6) – care are trei câmpuri și faptul (laborator-clips tematica fapt reguli variabile) care are cinci câmpuri. La ultimele

două fapte, primul câmp se poate interpreta ca fiind obiectul asupra căruia se face referire, al doilea câmp atributul obiectului, iar câmpurile următoare pot fi considerate valorile asociate atributului. De observat că se folosește o singură dată comanda (assert) pentru a introduce mai multe fapte.

În mediul CLIPS fiecare fapt primește un identificador (numit și index sau indice) unic, de forma fact-0, fact-1, fact-23, etc.

Nu este permis ca un fapt ordonat să includă un alt fapt.

Exemplu:

```
(dreptunghi lungime 40 latime 20)
```

Următorul fapt nu va fi acceptat

```
(dreptunghi (lungime 40 latime 20))
```

Toate numerele în CLIPS sunt tratate ca numere întregi sau reale în dublă precizie. Numerele fără zecimale sunt considerate a fi întregi, iar cele cu virgulă reale. Comanda (clear) este folosită pentru a îndepărta toate faptele din memorie și pentru a forța indicele de fapte să înceapă cu f-0.

Exemplu de folosire a numerelor pentru valorile unor câmpuri ale faptelor:

```
CLIPS> (clear)
```

```
CLIPS> (assert (numar 1)      (x 1.5)      (y -1.0)  
(z 65) )
```

```
<Fact-3>
```

```
CLIPS> (assert (distanța 3.5e5)      (coordonate  
1 2 3)      (coordonate 1 3 2) )
```

```
<Fact-6>
```

Lista de fapte va arăta astfel:

f-0 (numar 1) ;al doilea câmp are valoare întreagă  
f-1 (x 1.5) ;al doilea câmp are valoare reală  
f-2 (y -1.0) ;al doilea câmp are valoare reală negativă  
f-3 (z 65) ;al doilea câmp are valoare întreagă  
f-4 (distanța 350000.0) ;al doilea câmp are valoare  
reală  
f-5 (coordonate 1 2 3) ;câmpurile de pe pozițiile 2, 3 și 4  
au valori întregi  
f-6 (coordonate 1 3 2) ;este diferit de faptul anterior  
datorită valorilor diferite ale câmpurilor de pe  
poziția 3 și 4.  
For a total of 7 facts.

Câmpurile dintr-un fapt ordonat pot fi separate de delimitatori, care pot fi de spații, tabs, Enter.

Exemplu: următorul fapt va fi introdus o singura dată pentru că este același:

```
CLIPS> (clear)
```

```
CLIPS> (assert (Animal este "Zebra".))
```

```
<Fact-0>
```

```
CLIPS> (assert (Animal este "Zebra".  
) )
```

```
FALSE
```

```
CLIPS> (assert (Animal  
este  
"Zebra".) )
```

```
FALSE
```

```
CLIPS> (assert (Animal  
este  
"Zebra  
".) )
```

<Fact-1>

Lista de fapte va conține următoarele două fapte:

f-0            (Animal este "Zebra".)

f-1            (Animal este "Zebra  
")

Mediul CLIPS nu permite folosirea spațiilor în cadrul unui fapt, decât pentru delimitarea câmpurilor. Pentru a utiliza totuși spațiile acestea trebuie incluse între ".

## Exemplu: de folosire a spațiilor în interiorul unui câmp de tip string

```
CLIPS> (clear)
```

```
CLIPS> (assert (animal-este "cal")      (animal-  
este "cal ")      (animal-este " cal")      (animal-  
este " cal ")))
```

```
<Fact-3>
```

Lista de fapte va conține următoarele fapte:

```
f-0      (animal-este "cal")  
f-1      (animal-este "cal ")  
f-2      (animal-este " cal")  
f-3      (animal-este " cal ")
```

Dacă se încearcă inserarea unui fapt care deja există în lista de fapte atunci noua inserare va fi ignorată.

### Exemplu:

```
CLIPS>(assert (lungime 50) (latime 40)
(lungime 30))
CLISP>(assert (lungime 50))
FALSE
```

Lista de fapte va conține următoarele fapte:

```
f-0 (lungime 50)
f-1 (latime 40)
f-2 (lungime 30)
```

Există o opțiune a mediului CLIPS care permite adăugarea unui același fapt cu un nou indice, prin următoarea setare:

```
(set-fact-duplication <expresie-booleeana>)
```



### 3.1.2 Afișarea faptelor

Pentru a afișa lista de fapte se folosește comanda:

```
(facts [<start-integer> [<end-integer> [<max-integer>]])
```

start-integer este o valoare întreagă care exprimă indicele faptului cu care începe afișarea, end-integer arată indicele unui fapt la care să se oprească afișarea, max-integer fixează numărul maxim de fapte care pot fi afișate.

Exemplu: în care sunt introduse patru fapte ordonate în lista de fapte și apoi sunt urmărite diverse utilități ale comenzii (facts).

```
CLIPS> (clear)
```

```
CLIPS> (assert (a) (b) (c) (d) )
```

```
<Fact-3>
```

```
CLIPS> (facts)
```

```
f-0      (a)
f-1      (b)
f-2      (c)
f-3      (d)
For a total of 4 facts.
CLIPS> (facts 2)
f-2      (c)
f-3      (d)
For a total of 2 facts.
CLIPS> (facts 1 3)
f-1      (b)
f-2      (c)
f-3      (d)
For a total of 3 facts.
CLIPS> (facts 0 3 2)
f-0      (a)
```

f-1 (b)

For a total of 2 facts.

Afişarea liste de fapte se poate realiza în mediul CLIPS şi într-o fereastră separată, care poate fi deschisă cu opţiunea Facts Window din meniul Window.

### 3.1.3 Ștergerea faptelor

Pentru a șterge fapte din lista de fapte se folosește comanda  
(`retract <indice-fapt> | <adresa-fapt > + | *`)

Ca argument al comenzii se dă indicele faptului din listă sau adresa unui fapt (care poate fi obținută în partea condițională a unei reguli). Pentru a șterge toate faptele din lista de fapte se dă comanda (`clear`). Prin comanda `clear` mai sunt șterse toate regulile, variabilele globale, funcțiile, metodele și instanțele. Pentru a șterge toate faptele din lista de fapte și pentru a introduce faptul inițial se folosește comanda (`reset`).

Exemplu: în care sunt introduse patru fapte ordonate în lista de fapte și apoi sunt șterse folosind comanda (reset).

```
CLIPS> (clear)
```

```
CLIPS> (assert (a) (b) (c) (d) )
```

```
<Fact-3>
```

Pentru a șterge faptul cu indicele 2 se apelează comanda retract astfel:

```
CLIPS> (retract 2)
```

```
CLIPS> (facts)
```

```
f-0      (a)
```

```
f-1      (b)
```

```
f-3      (d)
```

```
For a total of 3 facts.
```

Dacă se încearcă ștergerea unui fapt care a fost deja șters sau a unui fapt inexistent se primește următorul mesaj:

```
CLIPS> (retract 2)
```

```
[PRNTUTIL1] Unable to find fact f-2.
```

Se pot șterge și mai multe fapte odată:

```
CLIPS> (retract 1 3)
```

```
CLIPS> (facts)
```

```
f-0      (a)
```

For a total of 1 fact.

Pentru a șterge toate faptele existente se folosește comanda (retract \*):

```
CLIPS> (retract *)
```

```
CLIPS>
```

După apelul comenzii (retract \*) indexul de fapte nu revine la 0, deoarece incrementarea următoarelor fapte se face pornind de la indicele ultimului fapt șters.

### 3.1.4 Alte funcții pentru lucrul cu faptele

- **load-facts**

`(load-facts <nume-fișier>)`

Introduce în lista de fapte a mediului CLIPS faptele încărcate dintr-un fișier primit ca argument. Argumentul trebuie să fie numele unui fișier, împreună cu calea de directoare completă, care conține o listă de fapte (nu constructori deffacts și nici alte comenzi sau constructori) separate prin delimitatorii admiși de mediul CLIPS.

Exemplu: fie fișierul test.txt care are următorul conținut:

(a)    (b)    (c)  
(d)

Pentru a încărca aceste fapte în mediul CLIPS se dă comanda  
(load-facts “C:/Clips-Exemple/ test.txt”)

- **save-facts**

`(save-facts <nume-fișier>)`

Salvează fapte într-un fișier specificat prin nume și calea completă de directoare. Încearcă să deschidă fișierul pentru a scrie, și apoi scrie lista faptelor mediului CLIPS în fișier.



### 3.1.5 Urmărirea faptelor

CLIPS asigură o serie de comenzi care ajută utilizatorul să depaneze programe. Una din aceste comenzi, (watch facts), permite urmărirea faptelor care sunt introduse sau șterse din lista de fapte. Este mai utila decât tastarea repetată a comenzii (facts).

Exemplu: de folosire a comenzii (watch facts)

```
CLIPS> (clear)
```

```
CLIPS> (watch facts)
```

```
CLIPS> (assert (Marti M1 curs "Inteligenta  
artificiala"))
```

```
==> f-0          (Marti M1 curs "Inteligenta  
artificiala")
```

```
<Fact-0>
```

Simbolul ==> arată că s-a introdus un fapt în memorie, iar simbolul <== arată că s-a șters un fapt din lista de fapte.

```
CLIPS> (reset)
<== f-0          (Marti M1 curs "Inteligenta
artificiala")
==> f-0          (initial-fact)
CLIPS> (assert (Marti M2 laborator "Clips"))
==> f-1          (Marti M2 laborator "Clips")
<Fact-1>
CLIPS> (retract 1)
<== f-1          (Marti M2 laborator "Clips")
CLIPS> (facts)
f-0          (initial-fact)
For a total of 1 fact.
```

Comanda (watch facts) asigură urmărirea dinamică a stării listei de fapte, spre deosebire de comanda (facts) care dă starea statică a listei în momentul apelării comenzii.

Pentru a înceta urmărirea faptelor se folosește comanda (unwatch facts). Pe lângă posibilitatea de urmărire a faptelor, CLIPS asigură opțiuni ale comenzii (watch) pentru urmărirea și a altor obiecte, după cum urmează:

(watch instances)	;folosite pentru instanțe ale obiectelor
(watch slots)	;folosite pentru obiecte
(watch rules)	;folosite pentru reguli
(watch activations)	;urmărește activarea regulilor
(watch messages)	;folosite pentru obiecte
(watch message-handlers)	;folosite pentru obiecte
(watch generic-functions)	;pentru funcții generice
(watch methods);	;folosite pentru metodele obiectelor

(watch deffunctions)	;urmărirea funcțiilor definite de utilizator
(watch compilations)	;activat implicit
(watch statistics)	
(watch globals)	;pentru variabile globale
(watch focus)	;urmărirea modulelor
(watch all)	;urmărește toate opțiunile enumerate anterior

Aceste opțiuni de urmărire pot fi selectate și din meniul Execution prin Watch Options sau din meniul Windows, prin bifarea ferestrei corespunzătoare, care permite urmărirea obiectelor în ferestre separate.

## ***3.2 Fapte neordonate***

Faptele neordonate permit introducerea unei structuri abstracte, în care numele faptului specifică un obiect. Obiectul poate avea mai multe câmpuri (numite sloturi), fiecare câmp din fapt având un nume și una sau mai multe valori asociate (multi-valoare). Ordinea sloturilor faptului nu are importanță, acesta fiind motivul pentru care se numesc fapte neordonate.

Definirea faptelor neordonate se face cu ajutorul constructorului (deftemplate) care are următoarea sintaxă:

```
(deftemplate <nume-deftemplate> [<comentariu>]
  <definitie-slot>*)
<slot-definition> ::=
  <definitie-slot-simplu> |
  <definitie-slot-multiplu>
<definitie-slot-simplu> ::= (slot <nume-slot>
  <atribut>*)
<definitie-slot-multiplu> ::=
  (multislot <nume-slot> <atribut>*)
<atribut> ::= <atribut-implicit> și/sau
  <atribut-explicit>
<atribut-implicit> ::=
  (default ?DERIVE | ?NONE | <expresie>*) |
  (default-dynamic <expresie>*)
```

Acest constructor este folosit pentru a crea șabloane în care câmpurile sunt accesibile prin numele lor (este asemănătoare definirii unei structuri în limbaj C). Constructorul (`deftemplate`) permite definirea unui fapt neordonat cu nici unul sau mai multe câmpuri (numite slot sau multislot). Sloturile care au asociate mai multe valori se numesc multi-slot. Spre deosebire de faptele ordonate, sloturilor li se poate specifica tipul, valoarea și valoarea implicită. Pentru a introduce un fapt neordonat definit cu constructorul (`deftemplate`) în lista de fapte se folosește comanda (`assert`).

Numele unui fapt neordonat și numele sloturilor, definite cu constructorul (`deftemplate`) trebuie să fie de tipul string. La faptele ordonate, pentru a avea acces la informații, utilizatorul trebuie să cunoască nu numai ce date sunt memorate în faptul respectiv, dar și poziția câmpurilor care conțin datele respective. Spre deosebire de

acestea, pentru a avea acces la informații în faptele neordonate este suficient să se cunoască numele atributului.

Un fapt definit prin constructorul (deftemplate) poate avea oricâte sloturi simple sau sloturi multiple. Atributele implicite (default sau default-dynamic), specifică valori care vor fi asociate cu un slot atunci când, la introducerea faptului în lista de fapte nu se dă nici o valoare slotului respectiv.

Valoarea *implicită* a unui slot poate fi statică (default) sau dinamică (default-dynamic). Valoarea implicită *statică* este evaluată o singură dată, atunci când este definit faptul, iar rezultatul este memorat în slotul corespunzător.



?DERIVE - valoarea implicită e dată de tipul valorii impuse slotului (de obicei este NIL), dacă nu este specificată o valoare explicită.

?NONE - valoarea slotului trebuie specificată explicit atunci când se introduce faptul în lista de fapte; dacă nu se atribuie o valoare explicită, mediul CLIPS va semnaliza eroare.

<expresie> specifică o expresie evaluată o singură dată la introducerea faptului în lista de fapte.

Valoarea implicită *dinamică* specifică o expresie care este evaluată de fiecare dată când este introdus faptul în lista de fapte, iar rezultatul este atribuit slotului corespunzător.

## Exemplu:

```
CLIPS>(clear)
```

```
CLIPS> (deftemplate cerc
        (slot x_centru (default ?NONE))
        (slot y_centru (default 0))
        (slot raza (default ?DERIVE)))
```

```
CLIPS>(assert (cerc))
```

[TMPLTRHS1] Slot x\_centru requires a value because of its (default ?NONE) attribute.

```
CLIPS>(assert (cerc (x_centru 20) (y_centru 30))
```

```
(cerc (x_centru 50) ))
```

```
<Fact-1>
```

```
CLIPS>(facts)
```

```
f-0      (cerc (x_centru 20) (y_centru 30) (raza
nil))
```

```
f-1      (cerc (x_centru 50) (y_centru 0) (raza  
nil))
```

For a total of 2 fact.

În exemplu (raza nil) semnifică faptul că pentru slotul raza nu are asociată nici o valoare. Valoarea "nil" este folosită pentru a indica existența unui câmp, chiar dacă acesta nu are nici o valoare. Dacă această valoare nu ar fi trecută, atunci slotul respectiv ar avea specificat doar numele, fără nici o valoare, ceea ce ar duce la erori.

Atributele *explicite* ale unui slot, permit specificarea tipului de date care poate fi memorat cu slotul respectiv și pot avea următoarea sintaxă:

```
<attribute_explicite> ::= (type <tip-de-date-  
permis>) |
```

(allowed-values ?<variabila> )

<tip-de-date-permis> specifica tipul de date pe care îl poate avea valoarea slotului respectiv: INTEGER, FLOAT, SYMBOL, STRING, EXTERNAL-ADDRESS, FACT-ADDRESS, INSTANCE-NAME, INSTANCE-ADDRESS.

<variabila> permite memorarea oricărui tip de date.

Exemplu: de utilizare a unui fapt neordonat, cu specificarea tipului de date al sloturilor (simbol pentru slotul nume, număr întreg pentru slotul vârstă, număr real pentru slotul id, valorile „m” și „f” pentru slotul sex, mai multe valori de tip simbol pentru multislottul adresă):

```
CLIPS> (deftemplate persoana
  (slot nume (type SYMBOL) (default ?DERIVE))
  (slot varsta (type INTEGER) (default ?NONE))
  (slot id (type FLOAT) (default 0.0))
  (slot sex (allowed-values f m ) (default m))
```

```
(multislot adresa (type SYMBOL) (default  
?DERIVE)))
```

```
CLIPS> (assert  
(persoana (nume Ionescu) (varsta 30) (id 12.3)  
(adresa str Domaneasca, nr.10))  
(persoana (nume Popescu) (varsta 41))  
(persoana (nume Florescu) (adresa str Brailei,  
nr.54) (sex f) (varsta 32) (id 45.2))  
(persoana (varsta 27)))
```

**Lista de fapte va conține următoarele fapte:**

```
f-1 (persoana (nume Ionescu) (varsta 30) (id  
12.3)  
(sex m) (adresa str Domaneasca, nr.10))  
f-2 (persoana (nume Popescu) (varsta 41) (id  
0.0)  
(sex m) (adresa))
```

```
f-3 (persoana (nume Florescu) (varsta 32) (id
45.2)
(sex f) (adresa str Brailei, nr.54))
f-4 (persoana (nume nil) (varsta 27) (id 0.0)
(sex m) (adresa))
```

La introducerea unui fapt persoana în lista de fapte, este obligatoriu să se precizeze o valoare pentru slotul varsta, deoarece are specificat ca valoare implicită ?NONE.

Redefinirea unui fapt neordonat duce la ignorarea definiției anterioare. Un fapt poate fi redefinit cu ajutorul constructorului deftemplate, numai dacă nu e folosit în momentul respectiv (dacă nu există nici un astfel de fapt în lista de fapte), în caz contrar mediul CLIPS afișează eroare.

Un fapt neordonat definit cu constructorul (deftemplate) poate avea oricâte sloturi sau multisloturi, în orice combinație. Facem observația

că un slot nu poate avea mai multe valori, în acest caz trebuie să se definească un multislot.

La introducerea unui fapt neordonat în lista de fapte nu are importanță ordinea în care sunt specificate sloturile.

### 3.2.1 Adăugarea de fapte

Comanda cu care se introduce un nou fapt la lista de fapte este (assert). Sintaxa comenzii este următoarea:

```
(assert <fapt>+)
```

Exemple ale folosirii acestei comenzi au fost prezentate anterior. Comanda (assert) poate fi folosită în mediul CLIPS în linia de comandă, sau în partea de acțiune a unei reguli.

Cu ajutorul constructorul (deffacts) se poate defini o listă de fapte, care va fi introdusă în lista de fapte a mediului CLIPS la execuția comenzii (reset). Sintaxa comenzii este:

```
(deffacts <nume-lista-fapte> [<comment>]  
  <fapt-neordonat>*)
```



Această comandă poate fi scrisă în linie de comandă a mediului CLIPS, sau poate fi scrisă într-un fișier CLIPS (cu extensia clp) ca o comandă de sine stătătoare, fiind utilă pentru introducerea bazei inițiale de cunoștințe. Faptele introduse prin acest constructor au același comportament cu al faptelor introduse prin comanda (assert).

Exemplu: de folosire a constructorului (deffacts) pentru introducerea în lista de fapte a mediului CLIPS a unor fapte ordonate si neordonate. Se salvează un fișier cu extensia .clp, care va fi încărcat în mediul CLIPS prin comanda (load), sau dacă se lucrează cu WinClips, se poate edita acest fișier cu editorul CLIPS-ului și apoi va fi încărcat prin comanda LoadBuffer, din meniul Buffer.

```
(deftemplate persoana
  (slot nume (type SYMBOL) (default
?DERIVE) )
  (slot varsta (type INTEGER) (default
?NONE) )
  (slot id (type FLOAT) (default 0.0))
  (slot sex (allowed-values f m ) (default
m) )
  (multislot adresa (type SYMBOL) (default
?DERIVE) ) )
```

```
(deffacts lista-fapte-initiale
;fapt ordonat
  (pers nume Ionescu varsta 30 id 12.3 adresa
str Domneasca nr.10)
;fapt neordonat
  (persoana (nume Ionescu) (varsta 30) (id 12.3)
(adresa str Domaneasca, nr.10)))
```

Observații: constructorul (deftemplate) pentru definirea șablonului unui fapt neordonat, trebuie să fie trecut înaintea constructorului (deffacts); primul câmp al faptului ordonat trebuie să fie denumit diferit față de numele faptului ordonat, pentru a nu declanșa erori.

### 3.2.2 Afişarea faptelor

Pentru a afişa lista de fapte se foloseşte comanda:

```
(facts [<start-integer> [<end-integer> [<max-integer>]])
```

pe care am prezentat-o la faptele ordonate, cap. 3.1.2.

Comanda cu care se poate urmări lista faptelor neordonate este:

(list-deftemplates),  
cu observaţia că afişează şi faptele ordonate.

Exemplu: pentru fișierul prezentat mai sus, vor fi afișate următoarele:

```
CLIPS> (reset)
```

```
CLIPS> (facts)
```

```
f-0 (initial-fact)
```

```
f-1 (pers nume Ionescu varsta 30 id 12.3  
adresa str Domneasca nr.10)
```

```
f-2 (persoana (nume Ionescu) (varsta 30) (id  
12.3) (sex m) (adresa str Domaneasca, nr.10))
```

```
For a total of 3 facts.
```

```
CLIPS> (list-deftemplates)
```

```
initial-fact
```

```
pers
```

```
persoana
```

```
For a total of 3 deftemplates.
```

### 3.2.3 Ștergerea faptelor

Pentru a șterge fapte din lista de fapte se folosește comanda  
(`retract <indice-fapt> | <adresa-fapt > + | *`)

Ca argument al comenzii se dă indicele faptului din listă sau adresa unui fapt (care poate fi obținută în partea condițională a unei reguli). Pentru a șterge toate faptele din lista de fapte se dă comanda (`clear`). Pentru a șterge toate faptele neordonate din lista de fapte care au fost introduse prin comanda (`assert`) și pentru a introduce faptul inițial se folosește comanda (`reset`). Efectul comenzii (`reset`) este de a reintroduce în lista de fapte, acele fapte definite prin constructorul (`def facts`), dacă nu a fost apelată anterior comanda (`clear`).

### 3.2.4 Alte funcții pentru lucrul cu faptele

Valoarea unui slot al unui fapt neordonat introdus cu (deftemplate) poate fi redefinită, cu condiția ca faptul respectiv să nu fie activ în acel moment (să nu existe nici un fapt cu acel nume în lista de fapte).

```
(modify <specificator-fapt> (<nume-slot>  
<valoare>)+)
```

<specificator-fapt> este indicele unui fapt sau adresa unui fapt (obținută în partea condițională a unei reguli). Fiecare modificare trebuie să conțină numele slotului ce urmează a fi modificat și noua valoare care se asignează slotului. În caz de succes al modificării, faptul modificat va fi șters din lista de fapte și un nou fapt se introduce în lista de fapte, similar cu faptul inițial, dar având sloturile specificate înlocuite cu valori noi. Efectul comenzii este de a întoarce indexul

faptului nou. Cu ajutorul acestei comenzi se pot modifica mai multe sloturi odată, dar nu se poate modifica decât un singur fapt, al cărui specificator este dat ca argument în funcția (modify).

Exemplu: pentru faptele neordonate definite în exemplu anterior:

```
CLIPS>(modify 2 (nume POPESCU) )
```

```
<Fact-3>
```

```
CLIPS>(facts 3)
```

```
f-2 (persoana (nume POPESCU) (varsta 30) (id  
12.3) (sex m) (adresa str Domaneasca, nr.10))
```

```
For a total of 1 fact.
```

Comanda de copiere a unui fapt neordonat, într-un nou fapt are sintaxa:

```
(duplicate <specificator-fapt> (<nume-slot>  
<valoare>)+)
```

Face o copie a faptului. Apelul comenzii trebuie să conțină indexul sau adresa faptului care se dorește a fi copiat, urmat, de o listă de sloturi a



căror valori se doresc a fi modificate în noul fapt și noua valoare care se asociază slotului. Efectul comenzii este introducerea unui nou fapt în lista de fapte, similar cu faptul copiat, dar având valorile sloturilor specificate înlocuite cu valori noi și afișarea indexului faptului nou.

Exemplu: în continuarea operațiilor exemplificate anterior:

```
CLIPS>(duplicate 3 (nume Georgescu))
```

```
<Fact-4>
```

```
CLIPS>(facts 3 4)
```

```
f-3 (persoana (nume POPESCU) (varsta 30) (id  
12.3) (sex m) (adresa str Domaneasca, nr.10))
```

```
f-4 (persoana (nume Georgescu) (varsta 30) (id  
12.3) (sex m) (adresa str Domaneasca, nr.10))
```

```
For a total of 2 facts.
```

Afișarea valorilor unui slot al unui fapt neordonat se face prin comanda:

```
(fact-slot-value <specificator-fapt> <nume-  
slot>)
```

Întoarce valoarea slotului specificat prin nume.

Afișarea numelor sloturilor unui fapt neordonat se face prin comanda:

```
(fact-slot-names <specificator-fapt>)
```

Întoarce o listă cu numele sloturilor faptului specificat prin index sau adresă.

Exemplu: de utilizare al ultimelor două comenzi:

```
CLIPS>(fact-slot-value 4 nume)
```

```
Georgescu
```

```
CLIPS>(fact-slot-names 4)
```

```
(nume varsta id sex adresa)
```

Funcțiile de urmărire, salvare în fișier, încărcare de fapte dintr-un fișier, pentru faptele neordonate sunt identice cu cele prezentate la faptele ordonate.

## **4. Reprezentarea cunoștințelor - Reguli**

Sursa de inspirație pentru sistemele bazate pe reguli de producție este o simplificare a modelului cognitiv uman. Regulile sunt principala formă de reprezentare a cunoștințelor în mediul CLIPS, prin specificarea unui set de acțiuni care urmează a fi realizate într-o anumită situație. Dezvoltarea unui sistem expert constă în definirea unui set de reguli care să rezolve problema dată. Regulile sunt folosite pentru a selecta setul de acțiuni ce trebuie executate, în cazul îndeplinirii unei condiții.

Procesul inferențial poate fi cu înlănțuire înainte (engl. forward chaining) sau cu înlănțuire înapoi (engl. backward chaining). Mediul CLIPS pune la dispoziție numai **inferență cu înlănțuire înainte**.

Un program scris în CLIPS își termină execuția când nu mai există nici o regula în agendă.

Dacă după ce o regulă a fost executată se dă comanda (run) din nou, înainte de a fi dată comanda (reset), cu toate că partea ei condițională este satisfăcută ea nu va mai fi activată în agendă. CLIPS ține cont de următoarele principii: o regulă este activată dacă partea condițională se potrivește:

- cu un nou șablon care nu exista înainte de activarea anterioară a regulii;
- cu un șablon care a existat înainte, dar a fost șters și reintrodus și astfel este definită o nouă entitate (un nou index).

## 4.1 Definirea regulilor

Regulile sunt definite folosind constructorul (defrule), care are următoare sintaxă:

```
(defrule <nume-regula> [<comentariu>]
; Left-Hand Side (LHS)
  [<declarație-proprietate>]
  <element-condițional>*
=>
; Right-Hand Side (RHS)
  <acțiune>*)
```

Întregul corp al regulii trebuie să fie inclus între paranteze, la fel și fiecare element condițional și fiecare acțiune. Prin sintaxa de definire a unei reguli partea condițională (LHS) este despărțită de partea de

acțiune (RHS) prin „ $\Rightarrow$ ” (egal, mai mare). Partea condițională este formată din zero sau mai multe elemente condiționale, care trebuie să fie adevărate pentru ca regula să fie aplicabilă. Fiecare șablon constă din unul sau mai multe câmpuri. Partea condițională a unei reguli mai poate conține declarații cu privire la proprietățile regulii, imediat după numele regulii și comentariu. O acțiune poate realiza introducerea sau ștergerea unui fapt sau a unei instanțe sau poate apela o secvență procedurală sau o funcție care, în general, nu întoarce nici o valoare, dar realizează ceva util pentru program (de exemplu afișarea unui mesaj). Nu există o limită a numărului de condiții sau de acțiuni pe care o regulă le poate avea (alta decât limitarea impusă de memoria disponibilă). Acțiunile sunt executate secvențial dacă și numai dacă, toate condițiile din antecedent sunt satisfăcute.



Redefinirea unei reguli deja existente, cauzează îndepărtarea vechii reguli (ștergerea ei), chiar dacă noua regulă este declarată eronat. La un moment dat în CLIPS nu pot exista două reguli cu același nume.

Dacă în antecedentul unei reguli nu se află nici o condiție, atunci este utilizat automat (initial-fact) sau (initial-object). Dacă nu sunt acțiuni în consecventul unei reguli, regula poate fi activată și executată, dar execuția ei nu va avea nici un efect.

Regulile pot fi scrise direct în mediul CLIPS sau pot fi preluate dintr-un fișier cu extensia „clp”, pentru a putea corecta mai ușor eventualele erori. Pentru ușurința editării regulilor, se face mai întâi trecerea de la limbajul natural la pseudocod, iar apoi se folosesc comenzi ale mediului CLIPS pentru reguli.

Exemplu: de regula

DACA unghi=90grade

ATUNCI triunghiul este dreptunghic

Regula poate fi introdusă pe o singura linie, dar poate fi introdusă și pe linii separate pentru a fi mai ușor de editat și de citit.

```
CLIPS> (clear)
```

```
CLIPS> (assert (unghi 90-grade) )
```

```
<Fact-0>
```

```
CLIPS> (defrule r1
```

```
  (unghi 90-grade) ;șablon  
=> (assert (triunghiul-este dreptunghic) )  
    ;actiune
```

```
CLIPS>
```

Dacă regula a fost introdusă corect prompt-ul mediului CLIPS va reapare. Altfel va fi afișat un mesaj de eroare.

Observație: numărul de paranteze deschise trebuie să fie egal cu cel al parantezelor închise (una din cele mai frecvente erori apare datorită parantezelor neperechi).

Dacă se apelează comanda (reset) memoria de lucru a mediului CLIPS este ștearsă și se introduce în lista de fapte faptul inițial (initial-fact). În urma acestei comenzi nu sunt șterse și regulile, ci doar agenda cu regulile activate.

## 4.2 Declararea proprietăților unei reguli

Proprietățile (caracteristicile) unei reguli se pot declara în antecedent, folosind comanda **declare**. O regulă nu poate avea decât un **declare**, care trebuie să fie situat înaintea de definirea elementelor condiționale. Sintaxa prin care se definesc proprietățile unei reguli este:

```
<declaratie-proprietate> ::=  
    (declare <proprietate-regula>+ )  
<proprietate-regula> ::=  
    (salience <expresie-integer> ) |  
    (auto-focus < TRUE | FALSE > )
```

Proprietatea salience permite utilizatorului să introducă un grad de încredere unei reguli. Gradul de încredere are o valoare întreagă și

poate fi cuprins în intervalul -10000 și 10000. Dacă gradul de încredere al unei reguli nu este declarat explicit, atunci mediul CLIPS asigură implicit valoarea "0".

Proprietatea auto-focus, dacă simbolul boolean este TRUE, permite ca o comandă focus să se execute ori de câte ori regula este activată.

## Exemplu: de definire a unor reguli cu valori specificate ale încrederii

```
(defrule test-1
  (declare (salience 99) )
  (fapt test-1)
=>
  (printout t "Regula test-1 este executata."
crlf ))
```

```
(defrule test-2
  (declare (salience 10 ) )
  (fapt test-1)
=>
  (printout t "Regula test-2 este executata."
crlf ))
```

Cele două reguli sunt activate în același moment, dar regula test-1 va fi executată înaintea regulii test-2, deoarece are valoarea încrederii asociate mai mare.

Mediul CLIPS execută întotdeauna regula cu cea mai mare prioritate din agenda, cea care este trecută prima în listă. În momentul în care acțiunea unei reguli a fost executată, regula este ștearsă din agendă, (altfel s-ar ajunge la bucle infinite) și se execută următoarea regulă. Acest proces continuă până când nu mai există nici o regulă în agendă sau până când se întâlnește o comandă de stop. Dacă sunt două reguli care au același grad de încredere se folosește o strategie de rezolvare a conflictelor care decide care regulă va fi activată prima.

Comanda cu care se poate vizualiza conținutul agendei este (agenda).

### Exemplu:

```
CLIPS> (agenda)
0                r1: f-0
For a total of 1 activation.
CLIPS>
```

Primul număr "0" reprezintă gradul de încredere al regulii, iar f-0 este indicele faptului care a contribuit la activarea regulii.

Regulile introduse în memoria mediului CLIPS, la un moment dat, pot fi listate cu comanda (rules).

### Exemplu:

```
CLIPS> (rules)
r1
For a total of 1 defrule. CLIPS>
```



## ***4.3 Sintaxa părții condiționale ale unei reguli***

Partea condițională a unei reguli este constituit din zero sau mai multe elemente condiționale (prescurtat CE) conjunctive.

Exista opt tipuri de elemente condiționale:

- **pattern**
- **test**
- **and**
- **or**
- **not**
- **exists**
- **forall**
- **logical**

Elementul condițional de tip **pattern** este cel mai întâlnit și cel mai utilizat. Acesta conține condiții (constrângeri) care sunt folosite pentru a afla dacă există fapte sau instanțe ce satisfac pattern-ul (șablonul). Elementul condițional de tip **test** se folosește pentru a evalua expresiile în procesul de pattern-matching (potrivire de șabloane). Elementul condițional **and** este folosit pentru a specifica dacă un întreg grup de elemente condiționale trebuie să fie îndeplinite simultan. Elementul condițional **or** se folosește atunci când se dorește să se specifice că este suficient să poate fi îndeplinit doar unul dintr-un grup de elemente condiționale.

Sintaxa pentru elementele condiționale specificate mai sus este:

$\langle \text{element-conditional} \rangle ::=$

$\langle \text{pattern-CE} \rangle \mid$

$\langle \text{not-CE} \rangle \mid$

$\langle \text{and-CE} \rangle \mid$

$\langle \text{or-CE} \rangle \mid$

$\langle \text{logical-CE} \rangle \mid$

$\langle \text{test-CE} \rangle \mid$

$\langle \text{exists-CE} \rangle \mid$

$\langle \text{forall-CE} \rangle$

### 4.3.1 Elementul condițional PATTERN

Elementele condiționale de tip pattern constau dintr-o serie de constrângeri de câmp și valori, eventual asociate unor variabile, care sunt folosite pentru a condiționa un set de fapte care se potrivesc șablonului. Constrângerile sunt folosite pentru a testa un câmp al unui fapt ordonat sau un slot al unui fapt neordonat. Ele pot conține doar o constrângere literală, dar totuși, pot fi compuse și din mai multe constrângeri conectate între ele. Pe lângă constrângerile literale, mediul CLIPS furnizează alte trei tipuri de constrângeri: conective, predicative, și respectiv, de valoare returnată.

## Constrângeri literale

Acest tip de condiție poate fi folosită în cadrul unui element condițional de tip **pattern** prin indicarea valorii exacte care corespunde cu valoarea câmpului. Se numește constrângere literală, deoarece este compusă în totalitate din constante de tip simbol, șir de caractere, număr întreg sau real, nume de instanțe. Această constrângere nu trebuie să conțină variabile.

Sintaxa unui astfel de element condițional este:

pentru un fapt ordonat, care conține doar literali:

(<constanta-1>...<constanta-n>)

pentru un fapt neordonat:

(<nume-slot> <constanta>)

## Exemplu: de reguli care au elemente condiționale de tip pattern

```
(deftemplate persoana
  (slot nume (type SYMBOL) (default
?DERIVE))
  (slot varsta (type INTEGER) (default
?NONE)))

(deffacts fapte-de-start
  (unghi 90 grade)
  (unghi 45 grade)
  (persoana (nume Ion) (varsta 30))
  (persoana (nume Ion) (varsta 45))
  (persoana (nume Vasile) (varsta 62)))

(defrule determina-tip-triunghi
  (unghi 90 grade)
=>
```

```
(printout t "Triunghiul este dreptunghic"
crlf))
```

```
(defrule cauta-Ion
(persoana (nume Ion) )
=>)
```

Se introduc aceste două reguli și faptele definite prin constructorul (deffacts) într-un fișier deschis în editorul mediului CLIPS (meniul File comanda New), salvat (Meniul File comanda Save), care apoi este încărcat în mediul CLIPS (fie cu funcția (load) din meniul File sau tastată la promter, fie cu comanda LoadBuffer din meniul Buffer, apelată din fișierul în care se face editarea). În continuare se apelează următoarea secvență de comenzi:

```
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
```

```
f-1    (unghi 90 grade)
f-2    (unghi 45 grade)
f-3    (persoana (nume Ion) (varsta 30))
f-4    (persoana (nume Ion) (varsta 45))
f-5    (persoana (nume Vasile) (varsta 62))
For a total of 6 facts.
```

```
CLIPS> (agenda)
0  cauta-Ion:    f-4
0  cauta-Ion:    f-3
0  determinare-tip-triunghi:  f-1
For a total of 3 activations.
CLIPS>
```

Se observă că este un singur fapt, cel cu indice f-1 care satisface constrângerea literală impusă în elementul condițional al regulei determinare-tip-triunghi, și există două fapte neordonate persoana, care



au slotul nume cu valoarea Ion, al căror indici sunt f-3 și f-4 care satisfac regula cauta-Ion. Activările acestor reguli se pot observa în agendă, prin comanda (agenda), de vizualizare a listei de reguli activate.

## Constrângeri conective

Sunt disponibile trei constrângeri conective, care pot fi combinate oricum și în orice număr. Acestea sunt următoarele:

- & (and)-dacă toate elementele condiționale conectate sunt adevărate;
- | (or)-dacă cel puțin un element condițional dintre cele conectate este satisfăcut;
- ~ (not)-dacă elementul condițional căruia îi este aplicat nu este adevărat.

Constrângerile conective au următoarea sintaxă:

<expresie-1>&<expresie-2>...&<expresie-n>

<expresie-1>|<expresie-2>...|<expresie-n>

~<expresie>

unde <expresie> poate fi variabilă, constantă sau constrângere conectivă.

Prioritatea constrângerilor conective este: ~, &, |.

Exemplu: de folosire a elementelor condiționale care conțin constrângeri conective. Următoarea secvență de cod va fi salvată într-un fișier cu extensia clp, care apoi va fi încărcat folosind funcția (load). În continuare se apelează funcția (reset) pentru a introduce în lista de fapte, faptele definite prin constructorul (deffacts) și funcția (run) pentru ca motorul de inferență să execute regulile definite. Explicațiile fiecărei reguli se găsesc în consecventul acesteia, unde se face afișare.

```
(deffacts fapte
  (fapt_1 aaa)
  (fapt_1 bbb)
  (fapt_1 ccc)
  (fapt_2 uuu)
  (fapt_2 vvv)
(fapt_2 www)
(fapt_3 3)
```

```
(fapt_3 4)
(fapt_3 5) )
```

```
(defrule r1
  (fapt_1 ~aaa)
=>
  (printout t "regula 1: IF not(aaa)"
  crlf))
```

```
(defrule r1-0
  (fapt_1 ~a)
;dacă în loc de valoarea „aaa” se trece „a”,
nu se considera a ;fi o eroare, deoarece se
încearcă găsirea unui fapt_1 care să ;nu aibă
valoarea „a” pe al doilea câmp
=>
```

```

        (printout t "regula 1-0: IF not(a) "
crlf))

(defrule r2
    (fapt_1 ?x&~aaa&~bbb)
=>
    (printout t "regula 2: IF (not(aaa) AND
not(bbb) )
        THEN: "?x crlf))

(defrule r3
    (fapt_1 aaa|ccc)
=>
    (printout t "regula 3: IF (aaa OR ccc) "
crlf))

```

```

(defrule r4
  (fapt_1 ?x)
  (fapt_2 ?y&~uuu&~vvv)
=>
  (printout t "regula 4: IF [fapt_1 AND
    (not(uuu) AND not(vvv))] THEN: fapt_1="?x "
    fapt_2=" ?y crlf))

(defrule r5
  (fapt_1 ?x&~aaa)
  (fapt_2 uuu|vvv)
=>
  (printout t "regula 5: IF [(not(aaa) AND
    (uuu OR vvv)] THEN: fapt_1="?x crlf))

```

Observație: variabilele se notează cu ? înaintea numelui, iar domeniul de vizibilitate este constructorul în care sunt apelate prima dată. Am notat cu ?x variabila cu numele x. Variabilele vor fi prezentate într-un alt capitol.

## Constrângeri predicative

Uneori este necesar să se verifice dacă valoarea unui câmp îndeplinește o anumită condiție impusă de valoarea de adevăr booleană a unei expresii date. Constrângerile predicative permit ca o funcție predicativă, care întoarce valori FALSE sau TRUE, să fie apelată și evaluată în timpul procesului de pattern-matching. De cele mai multe ori, constrângerile predicative sunt folosite în combinație cu constrângeri conective.

Sintaxa pentru constrângerile predicative este:

```
<termen> ::= <constanta> |  
           <variabila> |  
           :<apel-functie>
```



Exemplu: de folosire a constrângerilor predicative combinate cu constrângeri conective, în continuarea exemplului anterior

```
(defrule r6
  (fapt_3 ?y)
  (fapt_3 ?x&: (> ?x ?y) )
  =>
  (printout t ?x " > " ?y crlf))
```

Agenda va conține și următoarele activări

```
0  r6:      f-7 ,  f-9
0  r6:      f-8 ,  f-9
0  r6:      f-7 ,  f-8
```

## **Constrângeri ale valorii returnate**

Este posibil să se compare valoarea returnată de o funcție externă cu valoarea unui câmp. Constrângerea de valoare returnată, permite utilizatorului să apeleze și să evalueze funcții externe, din interiorul unui șablon din partea condițională a unei reguli. Valoarea returnată trebuie să fie una din tipurile primitive de date.

Sintaxa pentru constrângerile de valoare returnată este:

```
<termen> ::= <constanta> |  
            <variabila> |  
            :<apel-functie> |  
            =<apel-functie>
```

Exemplu: la fișierul anterior se mai adaugă regula

```
(defrule r7
  (fapt_3 ?y)
  (fapt_3 ?x & = (- (* ?y 2) 3) )
  =>
  (printout t ?x " = " ?y "*2-3" crlf) )
```

Agenda va cuprinde și activările:

```
0  r7:  f-8,  f-9
0  r7:  f-7,  f-7
```

## 4.3.2 Elementul condițional TEST

Elementul condițional de tip **test** este satisfăcut dacă funcția pe care o apelează este evaluată la o valoare TRUE și nesatisfăcut dacă funcția este evaluată la o valoare FALSE. Funcțiile apelate pot fi comparații între variabile și/sau constante, comparații logice sau pot fi apelate și testate funcții ale mediului CLIPS sau funcții externe definite de utilizator.

Sintaxa elementului condițional de tip test este:

```
<test-CE> ::= (test <apel-functie>)
```

Exemplu: se definește regula următoare care verifică dacă diferența a două numere este mai mare sau egală cu valoarea 2, folosindu-se faptele definite în exemplul anterior

```
(defrule r8
  (fapt_3 ?x)
  (fapt_3 ?y)
  (test (>= (abs (- ?y ?x) ) 2) )
=>
  (printout t "Regula 8" crlf))
```

Agenda va cuprinde activările

```
0  r8: f-9, f-7
0  r8: f-7, f-9
```

### 4.3.3 Elementul condițional OR

Acest tip de element condițional permite combinarea mai multor elemente condiționale, astfel încât să fie suficient să fie adevărat unul singur pentru a îndeplini o condiție. Dacă oricare dintre elementele condiționale dintr-un element condițional compus de tip **or** este satisfăcut, atunci elementul compus este satisfăcut. Dacă toate celelalte elemente condiționale din antecedentul regulei sunt satisfăcute, regula va fi activată. În interiorul unui element condițional or pot fi plasate oricâte elemente condiționale.

Sintaxa pentru elementul condițional or este:

$$\langle \text{or-CE} \rangle ::= (\text{or } \langle \text{element-conditional} \rangle^+ )$$

Exemplu: de utilizare a elementelor condiționale or. Se definește regula poligon-regulat, care este satisfăcută dacă exista un fapt (figura necunoscuta) și macar unul din faptele din elementul condițional or.

```
(defrule poligon-regulat
  (figura necunoscuta)
  (or (triunghi echilateral)
       (patrat)
       (hexagon regulat) )
```

=>

```
(printout t "Proprietati: toate unghiurile
si laturile congruente" crlf) )
```

Regula de mai sus este echivalentă cu următoarele trei reguli:

```
(defrule regulat-3
  (figura necunoscut)
  (triunghi echilateral)
```

=>

```
(printout t "Proprietati: toate unghiurile  
si laturile congruente" crlf) )
```

```
(defrule regulat-4  
(figura necunoscut)  
(patrat)
```

=>

```
(printout t "Proprietati: toate unghiurile  
si laturile congruente" crlf) )
```

```
(defrule regulat-6  
(figura necunoscut)  
(hexagon regulat)
```

=>



```
(printout t "Proprietati: toate unghiurile  
si laturile congruente" crlf) )
```

### 4.3.4 Elementul condițional AND

Mediul CLIPS consideră ca toate regulile au un elementele condiționale separate de AND implicit. Aceasta înseamnă ca toate elementele condiționale din antecedentul regulei trebuie să fie satisfăcute pentru ca regula să fie activată. Dar, mediul CLIPS pune la dispoziție și un and explicit, pentru a permite combinații complexe care să conțină and și or. Elementul condițional de tip and este satisfăcut dacă toate elementele condiționale incluse în acesta sunt satisfăcute. Într-un element condițional de tip and pot fi incluse un număr oricât de mare de elemente condiționale simple. Sintaxa elementului condițional de tip and este:

$$\langle \text{and-CE} \rangle ::= (\text{and } \langle \text{element-conditional} \rangle^+ )$$

Exemplu: de regulă care combină elemente condiționale de tip or cu elemente condiționale de tip and.

```
(defrule gripa
(stare boala)
(or (and (temperatura mare) (ochi inrositi))
    (and (temperatura mica) (dureri gat) (curge
nasul))
=>
(printout t "Bolnavul este suspect de gripa"
crlf ))
```

### 4.3.5 Elementul condițional NOT

Elementul condițional not este satisfăcut doar dacă acele condiții care îl compun nu sunt îndeplinite (sunt evaluate la o valoare de adevăr FALSE).

Sintaxa elementului condițional not este:

$$\text{<not-CE>} ::= (\text{not } \text{<element-conditional> } )$$

La un moment dat, doar un singur element condițional poate fi negat. Combinarea lui NOT cu AND și OR trebuie făcută cu multă atenție, deoarece rezultatele nu sunt întotdeauna clare.

Exemplu: de utilizare a elementului not, care verifică dacă nu există faptul (stare boala).

```
(defrule gripa
  (not (stare boala))
  (and (temperatura mare) (ochi inrositi))
  =>
  (printout t "Posibil stare de oboseala" crlf
  ) )
```

Un element condițional de tip NOT care conține un singur TEST este convertit astfel încât elementul condițional TEST să fie inclus într-un AND și să fie precedat de (initial-fact) sau (initial-object). De exemplu, următoarea condiție (val-1 și val-2 sunt numele unor variabile):

```
(not (test (> ?val-1 ?val-2) ) )
```

este convertită în:

```
(not (and (initial-fact)
          (test (> ?val-1 ?val-2) ) ) )
```

Totuși, ar fi mult mai simplu să convertim condiția dată ca exemplu, în forma următoare:

```
(test (not (> ?val-1 ?val-2) ) )
```

## 4.3.6 Elementul condițional EXISTS

Acest tip de element condițional este folosit pentru a determina dacă un grup de elemente condiționale este îndeplinit de măcar un set de date.

Sintaxa elementului condițional exist este:

$$\langle \text{exists-CE} \rangle ::= (\text{exists} \quad \langle \text{element-conditional} \rangle +)$$

Elementul condițional exists este implementat folosind două elemente condiționale de tip NOT. De exemplu, regula de mai jos:

```
(defrule exemplu
  (exists (a ?x) (b ?x) )
  =>)
```

este echivalentă cu următoarea:

```
(defrule exemplu
  (not (not (and (a ?x) (b ?x) ) ) )
  => )
```

Exemplu: de utilizare a elementului condițional exists. Regula exemplu-exists-1 va fi activată și executată de 2 ori deoarece exista cel puțin un fapt date al cărui slot latura să aibă valoarea romb și faptul (unghi ...), este prezent de 2 ori în lista de fapte (cu valorile 45 și 90). Regula exemplu-exists-2 va fi activată o singură dată, deoarece elementul condițional exists verifică dacă există cel puțin un fapt al cărui slot latura cu valoarea romb.

```
(deftemplate date
  (slot valoare)
```



```
(slot latura (default romb))  
(def facts fapte-start  
  (date (valoare 15))  
  (date (valoare 25))  
  (date (valoare 35))  
  (unghi 90)  
  (unghi 45))  
  
(defrule exemplu-exists-1  
  (unghi ?v)  
  (exists (date (latura romb))))  
=>  
  (printout t "Exista un romb cu unghi de "?v  
    crlf))  
  
(defrule exemplu-exists-2
```

```
(exists (date (latura romb)))  
=>  
(printout t "Exista un romb " crlf))
```

### 4.3.7 Elementul condițional FORALL

Elementul condițional FORALL are rolul de a determina dacă un număr de elemente condiționale dintr-un grup sunt îndeplinite pentru fiecare caz al unui alt element condițional, precizat explicit.

Sintaxa elementului condițional forall este:

```
<forall-CE> ::= (forall <element-conditional>  
                  <element-conditional>+ )
```

Acest tip de element condițional este echivalent cu două elemente condiționale: unul NOT și unul AND. De exemplu, regula de mai jos:

```
(defrule exemplu  
  (forall (a ?x) (b ?x) (c ?x) )  
=> )
```

este echivalentă cu următoarea regulă:

```
(defrule exemplu
  (not (and (a ?x) (not (and (b ?x) (c ?x) ) )
  ) )
=> )
```

Exemplu: următoarea regulă determină dacă toți studenții au absolvit examenele la Inteligența-artificială, Baze-de-date, Rețele-calculatoare, folosind elementul condițional forall. Singurul student care a trecut la toate examenele este Pop, iar regula va fi activată o singură dată.

```
(defacts fapte-start
  (student Pop)
  (Inteligența-artificială Pop)
  (Baze-de-date Pop)
  (Rețele-calculatoare Pop))
```

```
(defrule exemplu-forall
  (forall (student ?v)
    (Inteligenta-artificiala ?v)
    (Baze-de-date ?v)
    (Retele-calculatoare ?v))
=>
  (printout t "Studentul este absolvent" crlf))
```

### 4.3.8 Elementul condițional LOGICAL

Elementul condițional LOGICAL este folosit pentru a crea un grup de dependențe logice între șabloanele de tip logical din partea stângă a unei reguli cu entități (fapte sau instanțe) create în partea de dreaptă a regulii. Condiția logical grupează șabloanele împreună, exact ca și un AND explicit.

```
<logical-CE> ::= (logical <element-  
conditional>+ )
```

Poate fi folosit în combinații cu elemente condiționale de tip AND, OR, sau NOT. Trebuie menționat faptul că doar șabloanele logical ale unei reguli trebuie să fie poziționate înaintea celorlalte șabloane. De exemplu, regula de mai jos este corectă:

```
(defrule exemplu-logical-corect
```

```
(logical (a) )  
(logical (b) )  
(c)  
=>  
(assert (d) ) )
```

în timp ce următoarele reguli nu sunt corecte:

```
(defrule exemplu-logical-incorect-1  
  (logical (a) )  
  (b)  
  (logical (c) )  
=>  
  (assert (d) ) )
```

```
(defrule exemplu-logical-incorect-2  
  (a)
```

```
(logical (b) )  
(logical (c) )  
=>  
(assert (d) ) )
```

```
(defrule exemplu-logical-incorect-3  
  (or      (a)    (logical (b) ) )  
  (logical (c) )  
=>  
(assert (d) ) )
```



## Exemplu: de utilizare a dependențelor logice

```
CLIPS> (clear)
```

```
CLIPS>
```

```
(defrule regula1 ;dependenta logica intre a  
și x,y  
  (logical (a))  
  (b)  
=>  
  (assert (x) (y)))
```

```
CLIPS>
```

```
(defrule regula2 ;dependenta logica intre c  
și x,y  
  (logical (c))  
  (d)  
=>  
  (assert (x) (y)))
```

CLIPS>

Pentru a vedea cum funcționează dependențele logice se vor urmări faptele și activările de reguli și regulile.

```
CLIPS> (watch facts)
```

```
CLIPS> (watch activations)
```

```
CLIPS> (watch rules)
```

```
CLIPS> (assert (a) (b) (c) (d) )
```

```
==> f-0      (a)
```

```
==> f-1      (b)
```

```
==> Activation 0      regula1: f-0, f-1
```

```
==> f-2      (c)
```

```
==> f-3      (d)
```

```
==> Activation 0      regula2: f-2, f-3
```

```
<Fact-4>
```

```
CLIPS> (run)
```

```

FIRE      1 regula2: f-2, f-3
; regula2 introduce dependenta logica
==> f-4      (x)
==> f-5      (y)
FIRE      2 regula1: f-0, f-1
; regula1 introduce dependenta logica, dar nu mai sunt introduse încă
odata faptele (x) și (y) în lista de fapte, deoarece ele există deja.
CLIPS> (retract 0)
<== f-0      (a)
; se șterge dependenta logica dintre (a) și (x), (y)
CLIPS> (retract 2)
; se șterge dependenta logica dintre (c) și (x), (y), astfel (x) și (y) nu
mai au suport logic și vor fi șterse automat din lista de fapte.
<== f-2      (c)
<== f-4      (x)
<== f-5      (y)

```

```
CLIPS> (unwatch all)
```

```
CLIPS>
```

## ***4.4 Afişarea definiţiei unei reguli***

Pentru a tipări o regulă se foloseşte comanda (ppdefrule) urmată de numele regulii dorite.

Exemplu:

```
CLIPS> (ppdefrule r1)
  (defrule MAIN::r1
    (latura1 ?val1)
    (latura2 ?val2)
    =>
      (assert (figura dreptunghi))
      (printout t "Figura este dreptunghi"))
CLIPS>
```

## 5. Variabile în mediul CLIPS

Variabilele sunt folosite pentru a stoca valori. Spre deosebire de fapte, ale căror câmpuri au valori care sunt statice și pe parcursul unui program nu se schimbă, valoarea variabilelor este dinamică. În plus, odată ce un fapt a fost introdus în lista de fapte, câmpurile sale pot fi modificate doar prin ștergerea faptului respectiv și introducerea sa ca un nou fapt, cu valorile câmpurilor schimbate, și care are un nou indice.

Variabilele sunt de două tipuri:

- locale – sunt vizibile doar în constructorul în care au fost definite;
- globale – sunt vizibile din momentul în care au fost definite până la sfârșitul fișierului.

Valorile variabilelor pot fi unul din tipurile de date primitive (float, integer, symbol, string, external address, fact address, instance name și instance address). Spre deosebire de limbajele de programare unde, pentru a declara o variabilă este necesar să i se specifice tipul și eventual, domeniul de valori, mediul CLIPS evaluează variabila și îi asociază automat unul din tipurile primitive de date.

Variabilele pot avea valori:

- simple
- multivaloare

Numele unei variabilei trebuie să fie un simbol, cu observația că trebuie să înceapă cu o literă (în caz contrar deși nu se semnalează eroare, variabila este inexistentă).

## 5.1 Variabile locale

Forma de definire și de folosire a variabilelor locale este:

```
?<nume-variabila>
```

Înainte ca o variabilă să fie folosită trebuie să i se dea o valoare. În caz contrar, la apelul variabilei respective va apare un mesaj de eroare. Variabilele locale au domeniul de vizibilitate doar în constructorul în care au fost folosite.

Exemplu: de eroare la folosirea unei variabile

```
CLIPS> (clear)
CLIPS> (defrule test
  (initial-fact)
=>
  (printout t ?x crlf) )
```



```
[PRCCODE3] Undefined variable x referred in RHS  
of defrule.
```

```
ERROR:
```

```
(defrule MAIN::test  
  (initial-fact)  
  =>  
  (printout t ?x crlf) )
```

```
CLIPS>
```

Variabila x din partea dreaptă a regulii nu a fost definită și CLIPS afișează un mesaj de eroare pentru că nu poate asigura nici o valoare variabilei respective.

Variabilele locale sunt folosite, în principal, pentru a **memora valorile** folosite în elementele condiționale într-un anumit șablon din antecedentul regulilor și pentru a le folosi în consecventul regulilor.

### Exemplu:

```
(defrule determinare-tag  
  (tag ?nume)  
  =>
```

```
  (assert (cauta-tag-pereche ?nume) ) )
```

Dacă se introduce faptul (tag html) și apoi se dă comanda (run), după care se verifică lista de fapte, atunci noul fapt introdus în lista de fapte este (cauta-tag-pereche html), deoarece variabila nume are asociată de valoarea "html" obținută prin compararea variabilei cu valoarea celui de-al doilea câmp al șablonului din antecedentul regulii.

Exemplu: preluat din capitolul anterior, în care se asociază valoarea posibilă unui câmp cu două variabile x și y, aceste valori urmând a fi comparate.

```
(defrule r6
  (fapt_3 ?y)
  (fapt_3 ?x&: (> ?x ?y) )
  => )
```

De asemenea variabilele pot fi folosite pentru a **tipări** valorile asociate cu unele câmpuri.

Exemplu: de afișare a valorii celui de-al doilea câmp al șablonului din antecedentul regulii, asociat cu variabila nume. Variabila poate fi folosită de mai mult ori în cadrul constructorului în care a primit valoarea, dar utilizarea ei în cadrul oricărui alt constructor este eronată,

deoarece domeniul de valabilitate este numai în constructorul în care a primit prima oară valoarea. În exemplul de mai jos existența variabilei este legată de regula determinare-tag, iar după execuția regulei variabila este ștearsă, pierzându-și valoarea.

```
(defrule determinare-tag
  (tag ?nume)
  =>
  (printout t "tag-ul cautat este " ?nume
    crlf)
  (assert (pereche-tag _nume)))
```

Exemplu: preluat din capitolul anterior

```
(defrule r2
  (fapt_1 ?x&~aaa&~bbb)
  =>
  (printout t "regula 2: IF (not(aaa) AND
    not(bbb)) THEN: "?x ))
```

Se asociază valoarea celui de-al doilea câmp al faptului `fapt_1` cu variabila `x`, se compară această valoare cu `~aaa` și `~bbb` și apoi se afișează valoarea variabilei `x` într-un mesaj în funcția (`printout`).

În constructori diferiți se pot folosi aceleași variabile, fără ca valorile acestora să se influențeze.

O variabilă mai poate fi folosită pentru a **memora adresa un fapt** în partea dreaptă a unei reguli, faptul putând fi șters, modificat, duplicat. Pentru aceasta, se leagă adresa faptului de o variabilă în partea stângă a regulii. Deci variabila memorează adresa unei fapt, de tip `fact-address` și nu faptul. Adresa unui fapt este specificată prin operatorul "`<-`" (engl. pattern binding operator).

Exemplu:

```
CLIPS> (clear)
```

```
CLIPS> (assert (latura 4))
<Fact-0>
CLIPS> (facts)
f-0      (latura 4)
For a total of 1 fact.
CLIPS> (defrule latura
  ?l <- (latura ?val)
=>
  (retract ?l)
  (printout t "Latura are valoarea " ?val "
memorata in faptul cu indice " ?l crlf) )
CLIPS> (run)
Latura este <Fact-0>
CLIPS> (facts)
CLIPS>
```

Se observa că prin comanda printout este afișat atât valoarea asociată laturii memorată în variabila „val”, cât și indexul faptului memorat în variabila „l”, iar la apelul comenzii (facts) nu mai este afișat nici un fapt, deoarece faptul latura a fost șters în partea dreaptă a regulii latura. Chiar dacă faptul latura a fost șters, valoarea celui de-al doilea câmp mai este disponibilă până la sfârșitul regulii, acesta fiind memorată în variabila val. De notat că, atât regula, cât și faptul au același nume latura, mediul CLIPS acceptând pentru construcții ale unor cunoștințe diferite (reguli, fapte, funcții, variabile) același nume.

Exemplu: de utilizare a variabilelor pentru memorarea valorii unui câmp și a adresei unui fapt. Se observă că regula grupa va fi executată de trei ori, deoarece este activată de cele trei fapte, iar la fiecare execuție a regulii variabilele primesc alte valori.

```
CLIPS> (clear)
```

```
CLIPS> (defrule grupa
        ?id_stud <- (student ?nume)
=>
        (printout t ?nume " este student al acestei
grupe." crlf)
        (retract ?id_stud))
CLIPS> (deffacts studenti
        (student Adrian)
        (student Dan)
        (student Stefan))
CLIPS> (reset)
CLIPS> (run)
Adrian este student al acestei grupe.
Dan este student al acestei grupe.
Stefan este student al acestei grupe.
CLIPS>
```



O variabilă poate fi folosită și la citirea unei valori de la tastatură, care să poată fi folosită în diverse operații.

Exemplu: de citire a unei valori de la tastatură folosind funcția (read), asocierea acestei valori cu variabila val utilizând funcția (bind). Variabila este folosită apoi pentru afișare și introducerea unui nou fapt. Pentru a executa această regulă este suficient să se dea comanda (reset), urmată de (run).

```
(defrule introduce-valoare  
=>  
  (bind ?val (read) )  
  (printout t "valoarea = " ?val crlf)  
  (assert (valoare ?val)))
```

## ***5.2 Variabile multivaloare***

Pe lângă variabilele cu o singură valoare, mediul CLIPS pune la dispoziția utilizatorilor variabile multivaloare, care pot memora mai multe tipuri de date în același timp. Variabilele multivaloare sunt apelate cu următoarea sintaxă:

`$?<nume-variabila>`

## 5.2.1 Crearea de valori multicâmp

Această funcție grupează orice număr de câmpuri pentru a crea o valoare multicâmp. Sintaxa comenzii este:

```
(create$ <expresie>*)
```

Valoarea returnată de funcția create\$ este o valoare multicâmp cu un număr oarecare de date de un anumit tip. Dacă se apelează funcția fără argumente, atunci este creată o valoare cu lungime zero.

### Exemplu: de utilizare al funcției create

```
CLIPS (create$ ion vasile popescu iordachescu)
      (ion vasile popescu iordachescu)
CLIPS> (create$ (+ 3 4) (* 2 3) (/ 8 4))
      (7 6 2)
CLIPS>
```

## 5.2.2 Specificarea unui element al unui multicâmp

Se realizează cu funcția `nth$` care returnează un câmp al unei valori multicâmp. Sintaxa este următoarea:

```
(nth$      <expresie-integer>      <expresie-  
multicamp>)
```

primul argument este un număr întreg care specifică poziția câmpului care se dorește a fi returnat. Dacă acest întreg este mai mare decât numărul de campuri a valorii multicâmp, atunci se returnează nil.

Exemplu:

```
CLIPS> (nth$ 3 (create$ a b c d e f g) )  
c  
CLIPS>
```

### 5.2.3 Căutarea unui element al unui multicâmp

Funcția `member$` verifică dacă o valoare simplă se regăsește într-o valoare multicâmp. Sintaxa este:

```
(member$ <expresie> <expresie-multicamp>)
```

Funcția returnează poziția expresiei dacă aceasta se regăsește printre valorile expresie multicâmp, și FALSE în rest.

#### Exemplu:

```
CLIPS> (member$ galben (create$ rosu 3 "text"  
8.7 galben) )
```

```
5
```

```
CLIPS> (member$ 4 (create$ rosu 3 "text" 8.7  
galben) )
```

```
FALSE
```

CLIPS>

## 5.2.4 Ștergerea unui element al unui multicâmp

Funcția pentru ștergerea unor elemente ale unei variabile multicâmp are sintaxa:

```
(delete$ <expresie-multicamp> <begin-integer>  
<end-integer>)
```

Valoarea întoarsă de această funcție este valoarea multicâmp modificată prin ștergerea câmpurilor cu poziția începând la begin-integer și terminându-se la end-integer.

### Exemplu:

```
CLIPS> (delete$ (create$ camp1 camp2 camp3  
camp4 camp5) 3 4)  
(camp1 camp2 camp5)  
CLIPS> (delete$ (create$ computer imprimanta  
hard-disk) 1 1)  
(imprimanta hard-disk)  
CLIPS>
```



## 5.2.5 Extragerea unei secvențe de câmpuri

Funcția creează o nouă valoare multicâmp care conține câmpurile din valoarea inițială specificate prin poziția de început și sfârșit, cu următoarea sintaxă:

```
(subseq$ <valoare-multicamp> <begin-integer>  
<end-integer>)
```

### Exemplu:

```
CLIPS> (subseq$ (create$ camp1 camp2 camp3  
camp4) 3 4)  
(camp3 camp4)  
CLIPS> (subseq$ (create$ 1 "abc" def "ghi" 2)  
1 1)  
(1)
```

## 5.2.6 Înlocuirea unui câmp

Se realizează cu funcția `replace$`, care are sintaxa:

```
(replace$ <expresie-multicamp> <begin-integer>  
<end-integer> <expresie-sipla-sau-multicamp>+)
```

Scopul acestei funcții este de a înlocui cu o valoare simplă sau multicâmp specificată explicit prin al patrulea argument al funcției într-o expresie multicâmp între pozițiile specificate cu `begin-integer` și `end-integer`.

## Exemplu:

```
CLIPS> (replace$ (create$ camp1 camp2 camp3)
3 3 macheta)
(camp1 camp2 macheta)
CLIPS> (replace$ (create$ a b c d) 2 3 x y
(create$ q r s))
(a x y q r s d)
CLIPS>
```

## 5.2.7 Introducerea unui câmp

Funcția `insert$` introduce o valoare simplă sau multicâmp pe poziția specificată explicit printr-un număr întreg, după următoarea sintaxă:

```
(insert$ <expresie-multicamp>  
  <integer> <expresie-sipla-sau-multicamp>+)
```

### Exemplu:

```
CLIPS> (insert$ (create$ a b c d) 1 x)  
(x a b c d)  
CLIPS> (insert$ (create$ a b c d) 4 y z)  
(a b c y z d)  
CLIPS> (insert$ (create$ a b c d) 5 (create$  
q r) )  
(a b c d q r)
```

CLIPS>

Mai există și alte funcții pentru lucrul cu valori multicâmp (first\$, rest\$, implode\$, explode\$) care pot fi căutate în help-ul pus la dispoziție de mediul CLIPS.

## 5.3 Variabile globale

Pe lângă variabilele locale mai există și variabile globale, care sunt vizibile în toți constructorii fișierul în care au fost definite și în mediul CLIPS. Variabilele globale pot fi folosite într-un șablon al părții stângi ale unei reguli, dar schimbarea valorii asociate nu influențează procesul de potrivire al șabloanelor (pattern-matching). Pentru a putea fi folosite, variabilele globale se definesc cu constructorul:

```
(defglobal [<expresie>*)
```

Numele variabilei globale trebuie să fie de tip simbol, iar specificarea numelui modulului din care face parte este opțională (implicit este modulul MAIN). Expresia poate fi o valoare simplă sau o valoare multicâmp.

Observație: între numele variabilei, semnul = și expresie, trebuie să fie blankspace, altfel, mediul CLIPS afișează eroare.

Cu ajutorul unui constructor defglobal se pot defini mai multe variabile globale la un moment dat. Dacă o variabilă globală a mai fost definită, atunci vechea valoare este înlocuită cu valoarea specificată de noul constructor defglobal.

**Atribuirea** unei valori unei variabile globale se face cu funcția (bind).

**Restabilirea valorii inițiale** a unei variabile globale se realizează la apelul comenzii (reset) sau cu funcția (bind) fără argumente.

**Ștergerea** unei variabile globale se face folosind comanda (clear) sau comanda (undefglobal).

**Afișarea valorii** unei variabile se face prin apelul numelui variabilei.

Vizualizarea tuturor variabilelor globale și a valorilor acestora se poate face în fereastra Globals Window, din meniul Window.

**Afişarea definiţiei** unei variabile globale se face prin comanda (ppdefglobal).

Observaţie: argumentele comenzilor (undefglobal) şi (ppdefglobal) nu trebuie să conţină ? sau \*, se specifică doar numele variabilei.

Variabilele globale nu pot fi folosite pentru memorarea unei valori în partea stângă a unei reguli, şi nici ca argument al unei funcţii definite de utilizator.



## Exemple: de utilizare a variabilelor globale.

```
CLIPS> (defglobal
    ?*x* = 3
    ?*y* = ?*x*
    ?*z* = (+ ?*x* ?*y*)
    ?*q* = (create$ a b c) )
CLIPS> (ppdefglobal x)
(defglobal MAIN ?*x* = 3)
CLIPS> (bind ?*x* 10)
10
CLIPS> ?*x*
10
CLIPS> (reset)
CLIPS> ?*x*
3
CLIPS> ?*q*
```

```
(a b c)
CLIPS>(undefglobal x)
CLIPS>?*x*
[GLOBALDEF1] Global variable ?*x* is unbound.
FALSE
CLIPS>
```

## 6. Strategii de rezolvare a conflictelor

Odată ce baza de reguli a fost construită prin definirea regulilor, iar lista de fapte și lista de instanțe conțin entități, mediul CLIPS lansează procesul de potrivire a șabloanelor regulilor (pattern matchig) cu datele din liste pentru a determina regulile satisfăcute, pe care le trece în agendă, după care poate începe execuția regulilor activate. Agenda se comportă asemănător cu o stivă (prima regulă din agendă este cea care se va executa mai întâi). Într-un program scris într-un limbaj de programare, punctul de start, punctul de oprire și secvența de operații sunt definite explicit de către programator. În cazul mediului CLIPS, nu este nevoie de acest lucru, fiind suficientă comanda (run) pentru a executa regulile din agendă. Cunoștințele euristice (regulile) și datele (faptele și instanțele) sunt separate, iar motorul de inferență este

furnizat de către CLIPS și folosit pentru a căuta regulile activabile și pentru a le activa.

Ciclul de bază pentru execuția unei reguli este următorul:

- a) Se selectează pentru execuție prima regulă din agendă. Dacă nu există nici o regulă activată, atunci motorul de inferență se oprește.
- b) Pentru regula selectată sunt executate acțiunile din consecventul regulii. Se incrementează numărul regulilor care au fost executate pentru a putea fi folosit în cazul în care se specifică explicit numărul de reguli de executat.
- c) În urma pasului anterior se poate modifica lista de fapte și lista de instanțe, ceea ce determină modificarea agendei, prin activarea sau

dezactivarea unor reguli. Regulile nou activate (cele ale căror condiții sunt satisfăcute), sunt plasate în agendă. Locul unde sunt plasate este determinat de gradul de încredere al regulii și de strategia de rezolvare a conflictelor folosită. Regulile dezactivate sunt îndepărtate din agendă.

d) Dacă se folosește atribuirea dinamică a gradelor de încredere, atunci sunt reevaluate valorile gradelor de încredere pentru toate regulile din agendă. Deci, în urma acestei reevaluări sunt rearanjate regulile în agendă.

Motorul de inferență poate executa doar o regula la un moment dat. Acest lucru ar însemna că este nevoie de o strategie de rezolvare a conflictelor, care să determine o anumită aranjare a regulilor în agendă, aranjare care influențează ordinea de prioritate în care sunt executate

regulile. În agendă regulile sunt aranjate în primul rând după gradul de încredere asociat. Dacă toate regulile au același grad de încredere, atunci pentru departajarea acestora intervine strategia de rezolvare a conflictelor.

CLIPS pune la dispoziție următoarele strategii de rezolvare a conflictelor:

- depth - este strategia considerată implicit;
- breadth;
- lex;
- mea;
- complexity;
- simplicity;
- random;

În funcție de aplicație se alege una din aceste strategii, folosind comanda (set-strategy) cu următoarea sintaxă:

```
(set-strategy <strategie>)
```

Strategia de rezolvare a conflictelor poate fi schimbată în partea de acțiune a unei reguli, astfel încât regulile ulterioare vor fi trecute în agendă după noua strategie.

## ***6.1 Strategia depth***

Noile reguli activate sunt plasate deasupra tuturor regulilor cu același grad de încredere. Regulile activate datorită unor fapte care au un index superior sunt ordonate înaintea unor reguli activate de fapte de un index de ordin mai mic.

## **6.2 *Strategia breadth***

Noile reguli activate sunt plasate sub toate regulile cu același grad de încredere. Regulile activate datorită unor fapte care au un index inferior sunt ordonate înaintea unor reguli activate de fapte de un index de ordin mai mare.

## **6.3 *Strategia simplicity***

Printre reguli cu același grad de încredere, noile reguli activate sunt plasate deasupra tuturor regulilor cu specificitate egala sau mai mare. Specificitatea unei reguli este determinată de numărul de comparații ce trebuiesc executate în partea condițională a regulii. Fiecare comparație adaugă o unitate la specificitate. Deci, regulile mai „simple” sunt trecute în agendă înaintea celorlalte reguli.



## **6.4 *Strategia complexity***

Printre reguli ce au același grad de încredere, noile reguli activate sunt plasate deasupra regulilor cu specificitate egală sau mai mică.

## **6.5 *Strategia lex***

Printre reguli cu același grad de încredere, noile reguli activate sunt plasate folosind strategia OPS5. Această strategie folosește „noutatea” șabloanelor (pattern) care au activat regula. Fiecare fapt, respectiv fiecare instanță, este marcată intern cu o „etichetă de timp” ce indică noutatea relativă a acestuia față de fiecare alt fapt, respectiv instanță din sistem. Șabloanele asociate cu fiecare activare a unei reguli sunt sortate în ordine descrescătoare. O activare care conține șabloane mai recente este plasată înaintea activărilor cu șabloane mai vechi. Dacă o regulă activată are mai multe șabloane decât o altă regulă activată și

etichetele de timp sunt identice, atunci activarea cu mai multe etichete de timp va fi plasată în agendă înaintea celeilalte activări. Dacă două activări au același grad de încredere și aceeași noutate, activarea cu o specificitate mai mare este plasată în agendă înaintea activării cu o specificitate mai mică.

## **6.6 *Strategia mea***

Printre reguli cu același grad de încredere, noile reguli activate sunt plasate folosind strategia OPS5 cu același nume. Mai întâi, eticheta de timp a primului șablon este folosită pentru a determina unde trebuie plasată regula în agendă. O regulă care are eticheta de timp a primului șablon mai mare decât eticheta de timp a primului șablon corespunzător altei activări, este plasată în agendă înainte de cea de-a doua activare. Dacă ambele activări au aceeași etichetă de timp

asociată primului șablon, atunci este folosită strategia lex pentru a determina ordinea activărilor.

## **6.7 *Strategia random***

Fiecărei activări îi este atribuit un număr aleator care este folosit pentru a determina plasarea acesteia printre activările cu grad de încredere egal. Acest număr este memorat atunci când strategia este schimbată, pentru ca atunci când este reselectată strategia random să se furnizeze aceeași ordonare (pentru activările care erau deja în agendă când strategia a fost schimbată pentru prima oară).

## 7. Funcții definite de utilizator în mediul CLIPS

Funcțiile definite de utilizator sunt definite cu ajutorul constructorului (deffunction) și pot fi apelate apoi, ca orice altă funcție predefinită a mediului CLIPS. Funcțiile astfel definite, nu trebuie compilate, ele sunt interpretate de mediul CLIPS. Sintaxa de definire a unei funcții este:

```
(deffunction <nume> [<comentariu>]
  (<variabila-simpla>*          [<variabila-
multicamp>])
  <actiune>*)
```

Pentru a defini o funcție trebuie să i se precizeze numele, de tip simbol, un comentariu care este opțional, o listă de zero sau mai multe variabile simple sau multicâmp care formează argumentele funcției și o secvență

de operații sau expresii care vor fi executate secvențial la apelul funcției. Numele funcției trebuie să fie unic, și nu trebuie să corespundă cu numele unor funcții sistem. Definirea unei funcții trebuie să se facă înaintea apelării acesteia, excepție făcând funcțiile recursive. O funcție poate accepta un număr exact sau mai mic de argumente, depinzând de utilizarea sau nu a variabilelor multicâmp. Dacă se utilizează numai variabile simple, atunci numărul de argumente la apelul funcției trebuie să fie identic cu numărul de argumente de la definirea funcției. În cazul utilizării variabilelor multicâmp se pot folosi în interiorul funcției definite, comenzi pentru lucrul cu aceste variabile.

Exemplu: de funcție cu două argumente ale căror valori le afișează la ecran.

```
CLIPS> (clear)
```

```
CLIPS>
```

```
(deffunction afisare-argumente (?a ?b )  
  (printout t ?a " " ?b crlf))
```

```
CLIPS> (afisare-argumente 1 2)
```

```
1 2
```

```
CLIPS>
```

```
(deffunction afisare (?a ?b $?c)  
  (printout t ?a" "?b" si "?c" de lungime: "  
  (length ?c)                                crlf))
```

```
CLIPS> (afisare 1 2)
```

```
1 2 si () de lungime: 0
```

```
CLIPS> (afisare a b c d)
```

```
a b si (c d) de lungime: 2
```

CLIPS>

Valoarea întoarsă de o funcție este evaluarea ultimei expresii. Dacă o funcție nu are acțiuni de îndeplinit, atunci valoarea returnată este FALSE.

Funcțiile definite cu constructorul `deffunction` pot fi recursive, adică se pot apela în ele însele, sau în alte funcții.

Exemplu: de funcție care se apelează în partea sa de acțiune.

```
(deffunction factorial (?a)
  (if (or (not (integerp ?a)) (< ?a 0)) then
    (printout t "Factorial Imposibil!" crlf)
  else
    (if (= ?a 0) then
      1
    else
```



```
(* ?a (factorial (- ?a 1)))
```

Apelul funcției factorial se face astfel:

```
CLIPS>(factorial w)
Factorial Imposibil!
CLIPS>(factorial 4)
24
CLIPS>
```

Se observă că în funcția factorial au fost folosite funcții procedurale obișnuite în limbajele de programare, care au fost prezentate într-un capitol anterior.

Exemplu: de funcție recursivă care se apelează în altă funcție.

```
CLIPS>(deffunction f-apelata ()
  (printout t "Funcție apelată" crlf))
CLIPS>(deffunction f-apelanta ()
```

```
(printout t "Se apeleaza" crlf)
(f-apelata) )
CLIPS> (f-apelanta)
Se apeleaza
Functia apelata
CLIPS>
```

Exemplu: de funcție care afișează o întrebare corespunzătoare argumentului întreb și citește de la tastatură răspunsuri pe care le transformă în șiruri de caractere mici, pe care apoi le compară cu valori-permise, transmise ca argument al funcției. Această secvență se repetă până când răspunsurile introduse de la tastatură fac parte din lista de valori permise. Valorile permise sunt declarate ca valori multicâmp create folosind funcția create\$ la apelul funcției întrebare.

```

CLIPS> (deffunction intrebare (?intreb ?valori-permise)
  "raspunsul la o intrebare cu valori permise ale
raspunsului"
  (printout t ?intreb " " ?valori-permise " ? ")
  (bind ?raspuns (read))
  (if (lexemep ?raspuns) then (bind ?raspuns (lowercase
?raspuns))))
  (while (not (member ?raspuns ?valori-permise)) do
    (printout t ?intreb " " ?valori-permise " ? ")
    (bind ?raspuns (read))
    (if (lexemep ?raspuns) then (bind ?raspuns (lowercase
?raspuns)))))
  ?raspuns)
CLIPS> (intrebare "Raspundeti cu " (create$ da nu))

```

## **8. Aplicații realizate în mediul CLIPS**

### **Aplicația vreme**

Fișierele vreme1.clp, vreme2.clp, vreme3.clp

În fișierul vreme1 faptele inițiale sunt introduse prin regula initiala, care este înlocuită de constructorul deffacts în următoarele variante.

În fișierul vreme2, regulile ploua-da și ploua-nu sunt înlocuite în fișierul vreme3 cu regula ploua care are în partea dreaptă.

### **Aplicația ecgr2.clp**

Rezolvă prin diverse cazuri o ecuație de gradul 2.

### **Aplicația telefoaneFapte.clp**

Definește faptul neordonat telefon și caută cu o regulă două telefoane cu același nume, același model și două caracteristici diferite.

## **Aplicația arbore.clp**

Definește o structură arborescentă de fapte și demonstrează raționamentul.

## **Aplicația CE.clp, CE-forall.clp**

Exemplifică elementul condițional exists și forall.

## **Aplicația regulaPattern.clp**

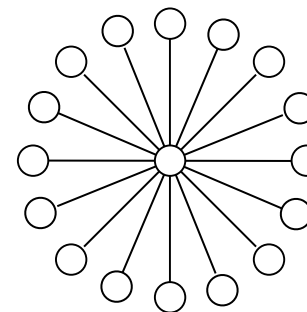
Exemplifică elemente condiționale simple.

## **Aplicația ordineReguli.clp**

Arată cum influențează încrederea unei reguli ordinea în care este trecută în agendă (regulile activate la un moment dat).

## Aplicația cerc.clp

Se dă următoarea figură. Să se așeze în cele 19 căsuțe toate numerele întregi de la 1 la 19, astfel ca suma numerelor din oricare trei căsuțe aflate pe aceeași dreaptă să fie egală cu 30.



Rezolvare: pentru a reduce spațiul de căutare se observă că în căsuța din centru trebuie să se așeze numărul 10, iar celelalte numere se împart în două grupe: grupa 1 cu numere de la 1 la 9, care urmează să fie așezate la un capăt al dreptei, grupa 2 cu numere de la 11 la 19, care vor fi așezate în capătul opus. Regula r1 introduce fapte în lista de fapte, iar regula r2 caută din cele două grupe perechi de fapte ale căror valori însumate să dea 20.

```
(defrule r1  
=>  
  (assert    (grupal 1)  (grupal 2))
```

```
(grupa1 3) (grupa1 4)
(grupa1 5) (grupa1 6)
(grupa1 7) (grupa1 8) (grupa1 9)
(grupa2 11) (grupa2 12)
(grupa2 13) (grupa2 14)
(grupa2 15) (grupa2 16)
(grupa2 17) (grupa2 18) (grupa2 19))
```

```
(defrule r2
  (grupa1 ?val1)
  (grupa2 ?val2)
  (test (eq (+ 10 ?val1 ?val2) 30))
=>
  (assert (pereche 10 ?val1 ?val2))
  (printout t "perechile care dau suma 30: 10
"?val1" "
```



```
?val2" "crlf))
```

## Aplicația numerelmpare.clp

Să se obțină numărul 20 adunând opt numere impare (se admite repetarea aceluiasi număr). Să se găsească toate soluțiile problemei.

Rezolvare: se observă că pentru a obține numărul 20 prin adunarea a opt numere impare, numărul 1 se repetă de patru ori. Celelalte numere impare se împart în patru grupe, soluția găsindu-se prin adunarea unor numere care fac parte din grupe diferite. Regula r1 are elemente condiționale care obligă ca soluțiile obținute să aibă numerele în ordine crescătoare, astfel încât să se evite repetarea unei soluții.

```
(def facts fapte-initiale
  (a 1) (a 3)
  (b 1) (b 3) (b 5)
  (c 1) (c 3) (c 5) (c 7)
  (d 5) (d 7) (d 11) (d 13))
( defrule r1
```

```

    (a ?val-a) (b ?val-b) (c ?val-c) (d ?val-
d)
    ( test(eq (+ ?val-a ?val-b ?val-c ?val-d)
16) )
    (test (<= ?val-a ?val-b) )
    (test (<= ?val-b ?val-c) )
    (test (<= ?val-c ?val-d) )
    =>
    (assert (per 1 1 1 1 ?val-a ?val-b ?val-c
?val-d) )
    (printout t "8 numere impare ce dau suma
20: 1 1 1 1 "?val-a" "?val-b" "?val-c" "?val-
d" " crlf) )

```

## Rezolvare 2:

```
(def facts initial
```

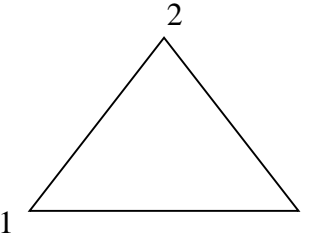
```
(impar 1)
(impar 3)
(impar 5)
(impar 7)
(impar 9)
(impar 11)
(impar 13)
(impar 15)
(impar 17)
(impar 19)
)
```

```
(defrule cauta
  (impar ?v1)
  (impar ?v2)
  (impar ?v3)
```

```
(impar ?v4)
(impar ?v5)
(impar ?v6)
(impar ?v7)
(impar ?v8)
(test (<= ?v1 ?v2) )
(test (<= ?v2 ?v3) )
(test (<= ?v3 ?v4) )
(test (<= ?v4 ?v5) )
(test (<= ?v5 ?v6) )
(test (<= ?v6 ?v7) )
(test (<= ?v7 ?v8) )
(test (eq 20 (+ ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7
?v8) ) )
=>
```

```
(assert (solutie ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7  
?v8) )  
)
```

## Aplicația tringhi.clp

Se consideră următoarea  figură. Să se realizeze un sistem care să așeze numerele 4, 5, 6, 7, 8, 9, pe laturile triunghiului, astfel încât suma numerelor pe fiecare latură să fie egală cu 17.

Rezolvare 1: deoarece sunt 6 numere se realizează 6 grupe cu aceste numere, repetate în fiecare grupă. Sumele pe fiecare latură trebuie să fie 14, 13 și 12, după ce s-au scăzut capetele. Se verifică fiecare număr cu celelalte pentru a fi diferite, și să fie în ordine crescătoare pentru a evita repetarea aceleiași soluții.

```
(def facts fapte-start
```

```
  (a 4) (a 5) (a 6) (a 7) (a 8) (a 9)
```

```
  (b 4) (b 5) (b 6) (b 7) (b 8) (b 9)
```

```
  (c 4) (c 5) (c 6) (c 7) (c 8) (c 9)
```

```
  (d 4) (d 5) (d 6) (d 7) (d 8) (d 9)
```

```

      (e 4) (e 5) (e 6) (e 7) (e 8) (e 9)
      (f 4) (f 5) (f 6) (f 7) (f 8) (f 9) )
)
(defrule r1
  (a ?val-a) (b ?val-b) (c ?val-c)
  (d ?val-d) (e ?val-e) (f ?val-f)
  (test (eq (+ ? val-a ?val-b) 14))
  (test (eq (+ ?val-c ?val-d) 12))
  (test (eq (+ ?val-e ?val-f) 13))
  (test (<> ?val-a ?val-b))
  (test (<> ?val-a ?val-c))
  (test (<> ?val-a ?val-d))
  (test (<> ?val-a ?val-e))
  (test (<> ?val-a ?val-f))
  (test (<> ?val-b ?val-c))
  (test (<> ?val-b ?val-d))

```



```

(test (<> ?val-b ?val-e) )
(test (<> ?val-b ?val-f) )
(test (<> ?val-c ?val-d) )
(test (<> ?val-c ?val-e) )
(test (<> ?val-c ?val-f) )
(test (<> ?val-d ?val-e) )
(test (<> ?val-d ?val-f) )
(test (<> ?val-e ?val-f) )
(test (< ?val-a ?val-b) )
(test (< ?val-c ?val-d) )
(test (< ?val-e ?val-f) )
=>
(assert (l1 ?val-a ?val-b) )
(assert (l2 ?val-c ?val-d) )
(assert (l3 ?val-e ?val-f) )
(printout t "                2" crlf)

```

```

(printout t "          /      "\" " crlf)
(printout t "          /      "\" " crlf)
(printout t "          " ?val-a "          " ?val-
c crlf)
(printout t "          /      "\" " crlf)
(printout t "          /      "\" " crlf)
(printout t "          " ?vb "          " ?val-
d crlf)
(printout t "          /      "\" " crlf)
(printout t "          /      "\" " crlf)
(printout t "1-----" ?val-e "-----" ?val-
f "-----3"crlf crlf crlf)
)

```

## Aplicație 4 robotLabirint.clp

Presupunem ca din punctul notat cu \* pleacă un robot. Să se realizeze un sistem care să asigure parcurgerea de către robot a tuturor căsuțelor cu excepția celor hașurate. Robotul nu trebuie să treacă decât o singură dată printr-o căsuță și să ajungă în punctul din care a plecat.

	1	2	3
1			*
2			
3			
4			

Rezolvare: se notează căsuțele de la stânga la dreapta și de sus în jos, ca în figură. Se definesc ca fapte, toate mutările posibile și poziția inițială și cea finală. Regula `r_init` verifică dacă există o mutare din poziția inițială. Dacă da, șterge faptul poziție inițială, pentru a nu mai fi parcurs încă o dată, iar noua poziție o etichetează ca fiind parcursă\_nu. Regula `r_mutare` verifică dacă din poziția curentă, există o mutare care să ducă într-o poziție parcursă\_nu. Dacă da, atunci poziția curentă devine parcursă\_da, iar noua poziție devine poziția curentă cu starea parcursă\_nu. Această regulă se repetă atâta timp cât

există mutări de făcut în poziții parcurse\_nu. Regula r\_final verifică dacă poziția curentă corespunde cu poziția finala, caz în care afișează că drumul s-a încheiat și s-a ajuns la destinație.

```
(def facts mutari
  (mutare 3 1 2 1) (mutare 3 1 3 2)
  (mutare 2 1 1 1) (mutare 2 1 3 1)
  (mutare 3 2 3 1) (mutare 3 2 3 3)
  (mutare 1 1 2 1) (mutare 1 1 1 2)
  (mutare 3 3 3 2) (mutare 3 3 2 3)
  (mutare 1 2 1 1) (mutare 1 2 1 3)
  (mutare 2 3 3 3) (mutare 2 3 1 3)
  (mutare 2 3 2 4) (mutare 1 3 1 2)
  (mutare 1 3 2 3) (mutare 1 3 1 4)
  (mutare 2 4 1 4) (mutare 2 4 2 3)
  (mutare 1 4 1 3) (mutare 1 4 2 4) )
```

```
(def facts pozitii
  (poz_init 3 1) (poz_finala 3 1))
```

```
(defrule r_init
  ?p <- (poz_init ?v1 ?v2)
  (mutare ?m1 ?m2 ?m3 ?m4)
  (test (= ?v1 ?m1))
  (test (= ?v2 ?m2)))
```

=>

```
(retract ?p)
(assert (poz ?m3 ?m4 parcursa_nu))
(printout t "Drum " ?m3 " " ?m4 crlf)
)
```

```
(defrule r_mutare
```

```

?p <- (poz ?v1 ?v2 parcursa_nu)
?m <- (mutare ?v1 ?v2 ?m3 ?m4)
(not (poz ?m3 ?m4 parcursa_da))
=>
(retract ?p)
(assert (poz ?v1 ?v2 parcursa_da))
(assert (poz ?m3 ?m4 parcursa_nu))
(printout t "Drum " ?m3 " " ?m4 crlf)
)

(defrule r_final
?p <- (poz ?v1 ?v2 parcursa_nu)
(poz_finala ?v1 ?v2)
=>
(printout t "S-a ajuns" crlf)
)

```

## Aplicație cadranCeas.clp

Să se execute un sistem care să împartă, cu ajutorul a două drepte cadranul unui ceas în trei părți, astfel încât adunând numerele din fiecare parte să se obțină aceeași sumă.

Rezolvare: pentru a rezolva această problemă se face observația că împărțind cele 12 ore de pe cadranul unui ceas cu două drepte, se obțin două serii de 4 numere consecutive, celelalte 4 numere fiind obținute prin excludere. Căutare celor două serii de numere consecutive se face considerând că fiecare număr poate fi începutul unei serii, aceasta fiind motivul pentru care se introduc faptele cifre la lista de fapte. Regula r1 verifică dacă există 4 numere consecutive a căror sumă să fie egală cu 26 ( $1+...+12=78$ ,  $78/3=26$ ).

```
(def facts ceas
```

```
(cifre 1 2 3 4 5 6 7 8 9 10 11 12)
(cifre 2 3 4 5 6 7 8 9 10 11 12 1)
(cifre 3 4 5 6 7 8 9 10 11 12 1 2)
(cifre 4 5 6 7 8 9 10 11 12 1 2 3)
(cifre 5 6 7 8 9 10 11 12 1 2 3 4)
(cifre 6 7 8 9 10 11 12 1 2 3 4 5)
(cifre 7 8 9 10 11 12 1 2 3 4 5 6)
(cifre 8 9 10 11 12 1 2 3 4 5 6 7)
(cifre 9 10 11 12 1 2 3 4 5 6 7 8)
(cifre 10 11 12 1 2 3 4 5 6 7 8 9)
(cifre 11 12 1 2 3 4 5 6 7 8 9 10)
(cifre 12 1 2 3 4 5 6 7 8 9 10 11)
)
```

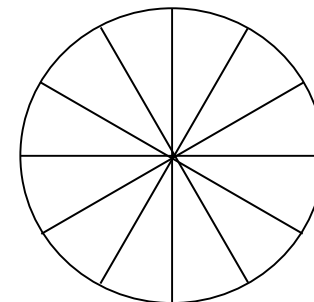
```
(defrule r1
```



```
(cifre ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
?v10 ?v11 ?v12)
(test (eq (+ ?v1 ?v2 ?v3 ?v4) 26))
=>
(assert (pereche ?v1 ?v2 ?v3 ?v4))
(assert (rest ?v5 ?v6 ?v7 ?v8 ?v9 ?v10 ?v11
?v12))
(printout t "Pereche: " ?v1 " " ?v2 " " ?v3 "
" ?v4 crlf)
)
```

## Aplicația diametre.clp

La capetele a cinci diametre trebuie așezate toate numerele întregi de la 1 până la 10, astfel încât suma oricăror două numere învecinate să fie egală cu suma celor două numere diametral opuse, corespunzătoare.



Rezolvare: numerele de la 1 la 10 se împart în 5 perechi cu diferențe egale, astfel:  $A+B=a+b$  sau  $A-a=b-B$  ( $A$  și  $B$  sunt numere învecinate, iar  $a$  și  $b$  sunt numerele diametral opuse). Rezultatele diferențelor pot fi doar 1 și 5. Astfel, avem două grupe de numere. Regula r1 caută perechi de două numere a căror diferență este egală cu 1. regula r11 aranjează cele 5 perechi de numere găsite de regula r1 astfel încât să îndeplinească toate condițiile impuse. Regulile r2 și r21 sunt pentru cazul în care diferența numerelor de pe un diametru este egală cu 5.

```
(def facts grupe
  (numere1 1) (numere1 2)
```

```
(numere1 3) (numere1 4)
(numere1 5) (numere1 6)
(numere1 7) (numere1 8)
(numere1 9) (numere1 10)
(numere2 1) (numere2 2)
(numere2 3) (numere2 4)
(numere2 5) (numere2 6)
(numere2 7) (numere2 8)
(numere2 9) (numere2 10) )
```

```
(defrule r1
  ?n1 <- (numere1 ?v1)
  ?n2 <- (numere1 ?v2)
  (test (eq (- ?v1 ?v2) 1 ) )
=>
  (retract ?n1 ?n2)
```

```

(assert (pereche1 ?v1 ?v2 ))
(printout t "Pereche1: " ?v1 " " ?v2  crlf))

(defrule r11
?p1 <- (pereche1 ?v1 ?v2)
?p2 <- (pereche1 ?v3 ?v4)
?p3 <- (pereche1 ?v5 ?v6)
?p4 <- (pereche1 ?v7 ?v8)
?p5 <- (pereche1 ?v9 ?v10)
(test (eq (+ ?v1 ?v4) (+ ?v2 ?v3)))
(test (<> ?v1 ?v3))
(test (<> ?v1 ?v5))
(test (<> ?v1 ?v7))
(test (<> ?v1 ?v9))
(test (<> ?v3 ?v5))
(test (<> ?v3 ?v7))

```

```

(test (<> ?v3 ?v9))
(test (<> ?v5 ?v7))
(test (<> ?v5 ?v9))
(test (<> ?v7 ?v9))
(test (eq (+ ?v3 ?v6) (+ ?v4 ?v5)))
(test (eq (+ ?v5 ?v8) (+ ?v6 ?v7)))
(test (eq (+ ?v7 ?v10) (+ ?v8 ?v9)))
(test (eq (+ ?v9 ?v2) (+ ?v10 ?v1)))
=>
(retract ?p1 ?p2 ?p3 ?p4 ?p5)
(assert (ordine1 ?v1 ?v4 ?v5 ?v8 ?v9 ?v2 ?v3
?v6 ?v7 ?v10))
(printout t "Ordine: " ?v1 " " ?v4 " " ?v5 "
" ?v8 " " ?v9 " " ?v2 " " ?v3 " " ?v6 " " ?v7
" " ?v10 crlf))

```

```

(defrule r2
  ?n1 <- (numere2 ?v1)
  ?n2 <- (numere2 ?v2)
  (test (eq (- ?v1 ?v2) 5 ))
=>
  (retract ?n1 ?n2)
  (assert (pereche2 ?v1 ?v2 ))
  (printout t "Pereche2: " ?v1 " " ?v2 crlf))

```

```

(defrule r21
  ?p1 <- (pereche2 ?v1 ?v2)
  ?p2 <- (pereche2 ?v3 ?v4)
  ?p3 <- (pereche2 ?v5 ?v6)
  ?p4 <- (pereche2 ?v7 ?v8)
  ?p5 <- (pereche2 ?v9 ?v10)
  (test (eq (+ ?v1 ?v4) (+ ?v2 ?v3)))

```

```
(test (<> ?v1 ?v3) )
(test (<> ?v1 ?v5) )
(test (<> ?v1 ?v7) )
(test (<> ?v1 ?v9) )
(test (<> ?v3 ?v5) )
(test (<> ?v3 ?v7) )
(test (<> ?v3 ?v9) )
(test (<> ?v5 ?v7) )
(test (<> ?v5 ?v9) )
(test (<> ?v7 ?v9) )
(test (eq (+ ?v3 ?v6) (+ ?v4 ?v5) ) )
(test (eq (+ ?v5 ?v8) (+ ?v6 ?v7) ) )
(test (eq (+ ?v7 ?v10) (+ ?v8 ?v9) ) )
(test (eq (+ ?v9 ?v2) (+ ?v10 ?v1) ) )
=>
(retract ?p1 ?p2 ?p3 ?p4 ?p5)
```

```
(assert (ordine2 ?v1 ?v4 ?v5 ?v8 ?v9 ?v2 ?v3
?v6 ?v7 ?v10))
(printout t "Ordine: " ?v1 " " ?v4 " " ?v5 "
" ?v8 " " ?v9 " " ?v2 " " ?v3 " " ?v6 " " ?v7
" " ?v10 crlf))
```



## Aplicația animal.clp

Prototip de sistem expert de identificare a mamiferelor pe baza întrebărilor cu răspunsuri de tip yes/no.

Se realizează trei fișiere:

1. fișierul sist.bat care va fi încărcat în mediul CLIPS cu comanda Load Batch din meniul File. Conținutul fișierului este:

```
(load "animal.clp")
```

```
(load "main.clp")
```

Pentru a evita erorile care pot apare la încărcarea celor două fișiere clp, se completează calea de directoare corespunzătoare.

2. fișierul animal.clp definește reguli care respectă combinațiile de caracteristici din tabela de mai jos:

```
;;; Tipul de sistem expert: identificare si  
selectie
```

```
;;;
```

;;; numar | zboara | pene | colti

|| mamifer

;;; ----- | ----- | ----- | -----

- || -----

;;; 0 | N | N | N ||

Y

;;; ----- | ----- | ----- | -----

- || -----

;;; 1 | N | N | Y ||

Y

;;; ----- | ----- | ----- | -----

- || -----

;;; 2 | N | Y | N ||

N

;;; ----- | ----- | ----- | -----

- || -----

; ; ; ;      3    |    N        |    Y        |    Y        | |

N

; ; ; ;    ----- |    ----- |    ----- |    -----

- | | -----

; ; ; ;      4    |    Y        |    N        |    N        | |

N

; ; ; ;    ----- |    ----- |    ----- |    -----

- | | -----

; ; ; ;      5    |    Y        |    N        |    Y        | |

Y

; ; ; ;    ----- |    ----- |    ----- |    -----

- | | -----

; ; ; ;      6    |    Y        |    Y        |    N        | |

N

; ; ; ;    ----- |    ----- |    ----- |    -----

- | | -----

```

;;;      7   |   Y   |   Y       |       Y           ||
N
;;; -----|-----|-----|-----
-||-----
;;;

```

```

(deftemplate caracteristici
(slot zboara)
(slot pene)
(slot colti))

```

```

(deftemplate clasificare (slot mamifer))

```

```

;;; Reguli yes/no folosite la identificarea
animalelor
;;; dupa caracteristici

```

```
(defrule mamifer-0 "numar 0"
  (caracteristici (zboara no) (pene no)
(colti no))
=>
  (assert (clasificare (mamifer yes))))

(defrule mamifer-1 "numar 1"
  (caracteristici (zboara no) (pene no)
(colti yes))
=>
  (assert (clasificare (mamifer yes))))

(defrule mamifer-2 "numar 2"
  (caracteristici (zboara no) (pene yes)
(colti no))
```

=>

```
(assert (clasificare (mamifer no))))
```

```
(defrule mamifer-3 "numar 3"  
  (caracteristici (zboara no) (pene yes)  
(colti yes))
```

=>

```
(assert (clasificare (mamifer no))))
```

```
(defrule mamifer-4 "numar 4"  
  (caracteristici (zboara yes) (pene no)  
(colti no))
```

=>

```
(assert (clasificare (mamifer no))))
```

```
(defrule mamifer-5 "numar 5"
```

```
(characteristici (zboara yes) (pene no)
(colti yes))
=>
(assert (clasificare (mamifer yes))))

(defrule mamifer-6 "numar 6"
  (characteristici (zboara yes) (pene yes)
(colti no))
=>
  (assert (clasificare (mamifer no))))

(defrule mamifer-7 "numar 7"
  (characteristici (zboara yes) (pene yes)
(colti yes))
=>
  (assert (clasificare (mamifer no))))
```

3. fișierul main.clp în care se definește funcția întrebare pentru interogarea utilizatorului (prezentată la capitolul funcții definite de utilizator)

```
(deffunction   intrebare   (?intreb   ?valori-  
permise)  
                "raspunsul la o intrebare cu  
valori permise ale raspunsului"  
  (printout t ?intreb " " ?valori-permise "  
? ")  
  (bind ?raspuns (read))  
  (if (lexemep ?raspuns) then (bind ?raspuns  
(lowercase ?raspuns)))
```



```

        (while (not (member ?raspuns ?valori-
permise)) do
            (printout t ?intreb " " ?valori-permise
" ? ")
            (bind ?raspuns (read))
            (if (lexemep ?raspuns) then (bind
?raspuns (lowercase ?raspuns))))
        ?raspuns)

```

```

;;; Deftemplate urmator permite culegerea
intrarilor de la
;;; utilizator intr-o forma asemanatoare cu
cea
;;; de la aplicatiile de baze de date
;;; Toate caracteristicile trebuiesc citite
inainte

```

```
;;; de a fi identificate
```

```
(deftemplate caracteristici2 (slot zboara))  
(deftemplate caracteristici1 (slot pene))  
(deftemplate caracteristici0 (slot colti))
```

```
(defrule adauga-in-memorie "adauga in baza de  
cunostinte"
```

```
  (caracteristici2 (zboara ?zb))  
  (caracteristici1 (pene ?p))  
  (caracteristici0 (colti ?c))  
=>
```

```
  (assert (caracteristici (zboara ?zb) (pene  
?p) (colti ?c))))
```

```
;;; *****
```

```

;;; * DEFROLE *
;;; *****

(defrule start "regula de start"
  (declare (salience 10000))
  (not (caracteristici2 (zboara ?)))
  =>
  (bind ?ans (intrebare "Animalul zboara"
                        (create$ yes
no)))
  (assert (caracteristici2 (zboara ?ans))))

(defrule r2
  (declare (salience 10000))
  (not (caracteristici1 (pene ?)))
  =>

```

```

        (bind ?ans (intrebare "Animalul are pene"
                               (create$      yes
no) ) )
        (assert (caracteristici1 (pene ?ans) ) )

(defrule r3
  (declare (salience 10000))
  (not (caracteristici0 (colti ?)))
  =>
  (bind ?ans (intrebare "Animalul are colti"
                        (create$      yes
no) ) )
  (assert (caracteristici0 (colti ?ans) ) )

(defrule mamifer-da "determina daca animalul
este mamifer"

```

```
(clasificare (mamifer yes))  
=>  
(printout t "Animalul este mamifer" crlf))  
  
(defrule mamifer-nu "determina daca animalul  
nu este mamifer"  
  (clasificare (mamifer no))  
  =>  
  (printout t "Animalul nu este mamifer"  
crlf))
```

## **Aplicația persoane.clp**

Exemplu de folosire a faptelor neordonate definite prin constructorul `deftemplate` pentru realizarea unei structuri ierarhice "persoana" cu

următoarele câmpuri: nr-id, nume, sex, data-nastere, tata, mama, culoare-ochi, inaltime. Se realizează trei fișiere:

1. fișierul persoane.bat

```
(load "persoane.clp")
```

```
(load "date.clp")
```

2. fișierul persoane.clp care definește sloturile faptului persoana și o regulă care afișează tatăl unei persoane:

```
(deftemplate persoana "Structura persoana"  
  (slot nr-id (type INTEGER))  
  (multislot nume)  
  (slot sex (type SYMBOL) (allowed-symbols  
M F) )  
  (multislot data-nastere )
```

```
(multislot tata)
(multislot mama)
(slot culoare-ochi (type STRING))
(slot inaltime (type FLOAT))
```

```
(defrule tata
  (persoana (nume ?nume1 ?nume2) (tata ?t-nume1
?t-nume2))
  (persoana (nume ?t-nume1 ?t-nume2) (sex M) )
=>
  (printout t ?t-nume1" "?t-nume2" este tatal
lui "?nume1" "?nume2 crlf))
```

3. fișierul date.clp care introduce baza de cunoștințe  
(deffacts familie-1

(persoana (nr-id 1) (nume Popescu  
Sorin) (sex M)  
                  (data-nastere 18 noiembrie 1952)  
                  (tata Popescu Nicolae) (mama  
Popescu Ioana)  
                  (culoare-ochi "caprui")  
(inaltime 1.85))

(persoana (nr-id 2) (nume Popescu  
Nicolae) (sex M)  
                  (data-nastere 23 ianuarie 1927)  
                  (tata Popescu Ion) (mama Popescu  
Maria)  
                  (culoare-ochi "albastru")  
(inaltime 1.82))



(persoana (nr-id 3) (nume Popescu  
Ioana) (sex F)  
                  (data-nastere 30 mai 1934)  
                  (tata Vasilescu Gheorghe) (mama  
Vasilescu Elena)  
                  (culoare-ochi "caprui")  
(inaltime 1.64))

(persoana (nr-id 4) (nume Popescu  
Diana) (sex F)  
                  (data-nastere 8 iunie 1954)  
                  (tata Popescu Nicolae) (mama  
Popescu Ioana)  
                  (culoare-ochi "albastru")  
(inaltime 1.69))

```
(persoana (nr-id 5) (nume Vasilescu
Gheorghe) (sex M)
          (data-nastere 10 august 1908)
          (tata Vasilescu Dumitru) (mama
Vasilescu Ileana)
          (culoare-ochi "verzi") (inaltime
1.80))
```

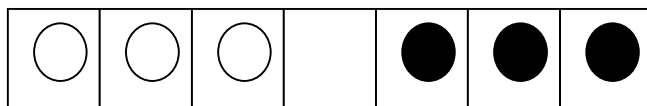
```
(persoana (nr-id 6) (nume Vasilescu
Elena) (sex F)
          (data-nastere 18 decembrie 1910)
          (tata Vasilescu Marian) (mama
Ionescu Ana)
          (culoare-ochi "verzi") (inaltime
1.59))
)
```

**Exercițiu:** să se determine toate legăturile de rudenie care există între datele introduse în fișierul date.clp. Să se afișeze toate persoanele de sex F care au ochi verzi. Să se determine toate persoanele cu vârstă cuprinsă între 20 și 40 ani. Să se afișeze toate persoanele de sex M cu înălțimea cel puțin 1.70.

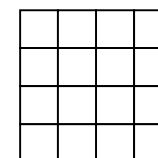


## Probleme propuse

1. Să se scrie un sistem în mediul CLIPS care să afișeze calendarul unei luni dintr-un an, ambele introduse de la tastatură.
2. Să se mute piesele albe în locul celor negre, iar cele negre în locul pieselor albe, după regula: piesele pot fi mutate în spațiul liber vecin, sau pot sări peste piesa învecinată dacă dincolo de ea este un spațiu liber, piesele se mișcă numai înainte. Problema se rezolvă din 15 mutări.



3. Să se distribuie numerele de la 1 la 16 în caroiul din figură astfel încât suma numerelor de pe fiecare rând, coloană și diagonală să fie egală cu 34.



4. Să se definească toate regulile pentru calculul unui triunghi dreptunghic.