Medii de programare în Inteligența Artificială

JESS

Lect. univ. dr. Mihaela Colhon

http://inf.ucv.ro/~ghindeanu

Reprezentarea faptelor in JESS

Toate datele JESS necesare efectuarii unui rationament sunt incarcate in *memoria de lucru* (in engleza, *working memory*).

Memoria de lucru mai este denumita si *baza de fapte* (*fact base*) sau *baza de cunostinte* (*knowledge base*) si contine toate acele informatii necesare in obtinerea deductiilor. Structura acesteia aminteste de cea a bazelor de date relationale, deoarece toate faptele incarcate in memoria de lucru au o reprezentare comuna.

Informatia in JESS

În CLIPS există trei forme principale de reprezentare a informaţiei:

- fapte (**facts**),
- obiecte
- variabile globale.

Faptele sunt forma principală de reprezentare a informaţiei în sistemul CLIPS. Si in Jess **faptul** este forma fundamentală de date folosită de **reguli** şi constă într-un **nume de relaţie** urmat de zero sau mai multe **sloturi** (câmpuri simbolice cărora li se pot asocia valori).

Tipuri de fapte

Un fapt este cea mai mica structura de informatie care poate fi incarcata sau extrasa din memoria de lucru a unui sistem bazat pe reguli.

In Jess exista trei tipuri de fapte:

- *fapte ordonate* (*ordered facts*),
- *fapte neordonate* (*unordered facts*)
- *fapte de tip definstance* (*definstance facts*).

La fel ca toate celelalte constructii in Jess, faptele sunt reprezentate sub forma de liste.

Tipuri de fapte Jess. Exemple

ordered facts

- (person 30 Female)
- (person 30)
- (give-Feedback)
- (shutdown-now)

unordered facts (deftemplate)

- (person (age 30) (gender Female))
- (**assert** (monkey (location t5-7)
- (on top-of green-couch)))

definstance (shadow) facts

- connecting with the Java application

Nu au structura predefinita. Numele faptului este dat de prima constanta si se incarca in memorie cu **assert**

Structura se defineste cu **deftemplate** si se incarca in memorie cu **deffacts** sau **assert**

Fapte neordonate; structura se defineste cu **defclass** si se incarca in memorie cu **definstance**

Fapte ordonate

Un fapt ordonat (**ordered fact**) constă într-un nume de relaţie urmat de zero sau mai multe câmpuri separate de spaţii. Acesta ar putea fi comparat cu un slot de tip multifeld (multi-câmp) în care valorile se după numele de relaţie. Faptele ordonate sunt create automat de un deftemplate **implicit.**

De exemplu, urmatoarele scrieri sunt asimilate drept fapte ordonate:

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

Fapte ordonate

Observam, ca in esenta, faptele ordonate sunt liste simple in Jess in care capul listei are un rol special, el specificand categoria de informatie a datelor stocate in respectivul fapt. De exemplu:

- (lista-fructe mere pere mango)
- (angajat ”Ion Popescu” 35 sef-departament)

Câmpurile dintr-un fapt ordonat pot fi orice tip de dată primitivă (cu excepţia primului câmp care reprezintă numele de relaţie şi care trebuie să fie de tip simbol).

Functii JESS pentru gestionarea memoriei de lucru (1)

Jess are predefinite functii prin intermediul carora se poate manipula continutul memoriei de lucru.

- **assert:** incarca fapte in memoria de lucru
- **(clear):** sterge in totalitate continutul memorie de lucru
- **deffacts:** defineste continutul initial al memoriei de lucru
- **(facts):** afiseaza lista faptelor incarcate in momentul respectiv in memoria de lucru

Functii JESS pentru gestionarea memoriei de lucru (2)

- **(reset):** reinitializeaza memoria de lucru prin revenirea la continutul initial al memoriei de lucru
- **retract:** scoate faptele precizate din memorie de lucru
- **watch:** are rolul de a supraveghea continutului memoriei de lucru si de a anunta modificarile survenite; atunci cand se doreste suprimarea monitorizarii memoriei de lucru se apeleaza functia **unwatch**.

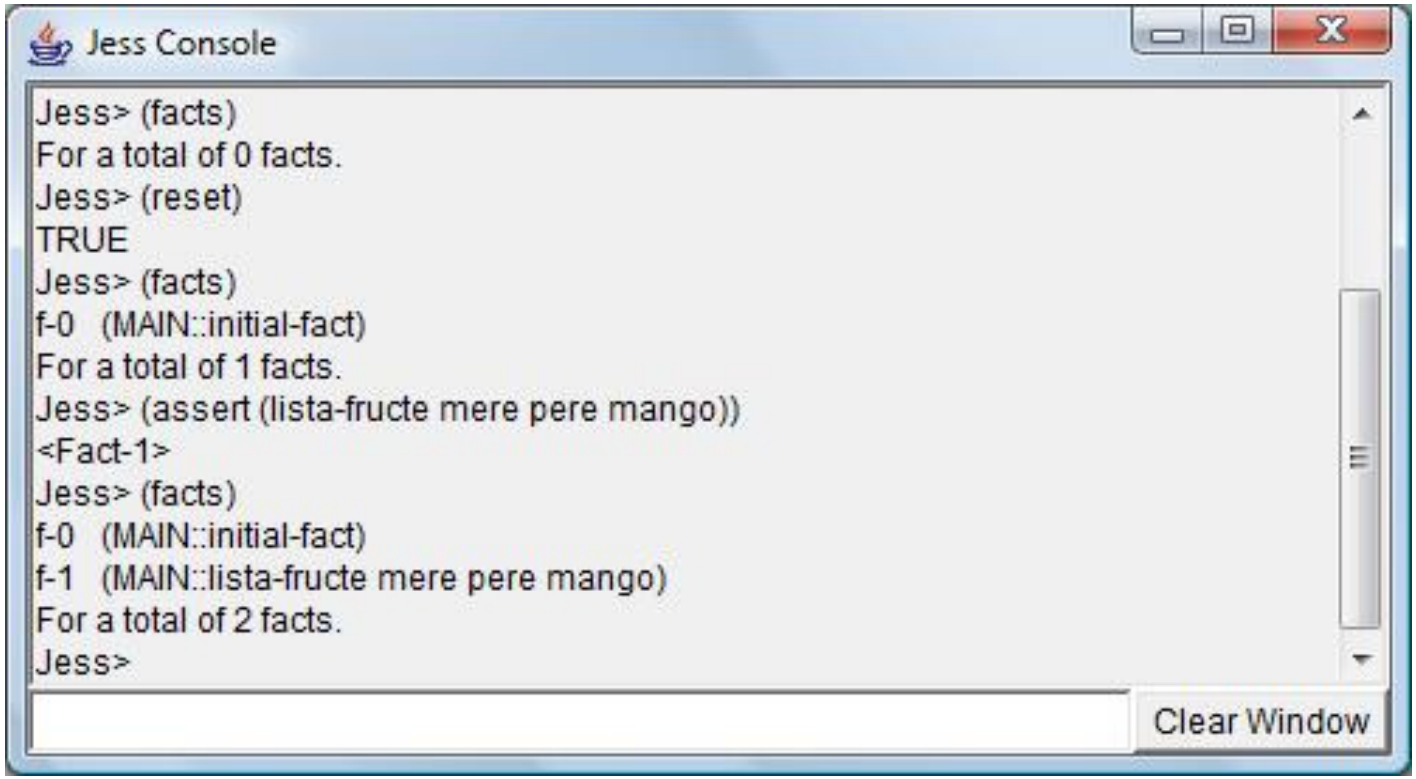
Incarcarea faptului MAIN

In functie de parametrii folositi in apelul functie **watch**, Jess ofera informatii despre memoria de lcuru, ca de exemplu:

```
Jess> (watch facts)
TRUE
Jess> (reset)
==> f-0 (MAIN::initial-fact)
TRUE
Jess> (unwatch facts)
TRUE
```

Semnul ==> indica incarcarea in memorie a unui nou fapt, in vreme ce <== indica extragerea din memorie a unuia sau mai multor fapte.

watch. Exemplu



Gruparea faptelor. Constructorul deffacts (1)

Într-un program este foarte avantajos ca să se poată adauga toate faptele ce reprezintă baza iniţială de cunoaştere într-o singură aserţiune. Aceste fapte sunt cunoscute a fi adevărate încă înainte de rularea programului. Se poate defini un grup de fapte care reprezintă baza iniţială de cunoaştere cu ajutorul construcţiei *deffacts*.

Pentru a realiza asertarea faptelor dintr-o declaraţie *deffacts* trebuie obligatoriu utilizată comanda **reset**. Efectul acestei comenzi este acela că şterge toate faptele din baza de fapte (ca şi comanda *clear*) şi introduce faptele din declaraţiile **deffacts**.

Jess> (reset)

Observatie. De asemenea, comanda (reset) incarca in memorie faptul initial MAIN.

Gruparea faptelor. Constructorul deffacts (2)

Avantajele oferite de **deffacts** sunt considerabile, ceea ce il fac un bun candidat pentru construirea bazei de cunostinte a oricarui program Jess bazat pe cunostinte:

- inglobeaza mai multe definiri de fapte; in loc sa apelam **assert** pentru fiecare fapt pe care vrem sa-l incarcam in memorie, **deffacts** poate fi folosit pentru un set de fapte; mai precis, cu ajutorul lui **deffacts** putem defini o lista de fapte
- faptele definite cu ajutorul lui **deffacts** compun continutul initial al memoriei de lucru sau *baza initiala de cunostinte* astfel incat la orice folosire ulterioara a comenzii **(reset)** se poate reveni la starea problemei descrisa de faptele astfel definite.

Gruparea faptelor. Exemplu

```
Jess> (clear)
TRUE
Jess> (deffacts tip_mere ”Sortimente de mere”
      (mar ”Cortland” dimensiune-mare culoare-rosie)
      (mar ”Golden Delicious” dimensiune-mare culoare-galbena)
      (mar ”Red Delicious” dimensiune-medie culoare-rosie)
      (mar ”Granny Smith” dimensiune-mare))
Jess> (facts)
For a total of 0 facts.
Jess> (reset)
TRUE
Jess> (facts)
```

- f-0 (MAIN::initial-fact)
- f-1 (MAIN::mar "Cortland" dimensiune-mare culoare-rosie)
- f-2 (MAIN::mar "Golden Delicious" dimensiune-mare culoare-galbena)
- f-3 (MAIN::mar "Red Delicious" dimensiune-medie culoare-rosie)
- f-4 (MAIN::mar "Granny Smith" dimensiune-mare)
- For a total of 5 facts.

Fapte neordonate. Constructorul deftemplate

Pana acum, am folosit **deffacts** pentru a defini fapte ordonate. Marele dezavantaj al acestora este ca nu pot structura datele in campuri sau slot-uri ca in programarea orientata-obiect. Pentru aceasta este indicat sa se foloseasca reprezentarea oferita de *faptele neordonate*.

Faptele neordonate sunt asemanatoare inregistrarilor dintr-un tabel de date relational, sloturile specificand numele campurilor in care se stocheaza datele inregistrarii.

Fapte neordonate. Constructorul deftemplate

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

Pentru acest lucru, constructorul **deftemplate** are rolul de a crea un şablon ce va fi folosit pentru a accesa câmpurile faptului după nume. Acesta este identic cu o *structură* definită într-un limbaj de programare procedural cum ar fi C.

Constructorul deftemplate:

In Jess functia **deftemplate** are urmatoarea sintaxa:

```
(deftemplate <nume_template>
        [extends          <nume_clasa>]
        [<comentariu>]
        [(slot <nume_slot>
        [(default | default-dynamic <valoare>)]
        [(type <typespec>))]*)
```

Iata cum ar arata template-ul pentru faptele **tip_mere** construite mai sus:

```
(deftemplate mar
                (slot soi (type STRING))
                (slot dimensiune (type STRING))
                (slot culoare (type STRING) (default rosie)))
```

Fapte ordonate vs. Fapte neordonate

Jess-ul face deosebirea între un *ordered-fact* şi un *deftemplate-fact* extrăgând primul câmp din fapt (numele de relaţie) şi comparându-l apoi cu numele fiecărui deftemplate. Dacă la un fapt **deftemplate**, schimbând ordinea sloturilor se obţinea acelaşi fapt, în schimb la un fact ordonat schimbând ordinea câmpurilor se obţine un alt fapt.

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

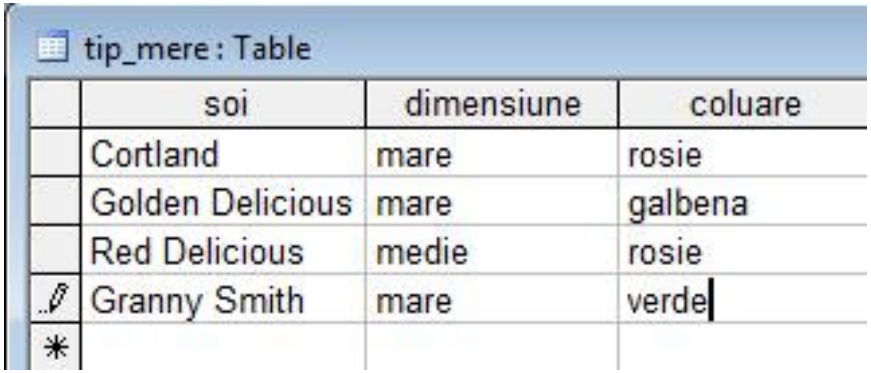
Uneori este mai avantajoasă folosirea unui fapt de tip deftempate în locul unui fapt ordonat, deoarece se specifică prin numele sloturilor semnificaţia fiecărui câmp. In cazul următor, dintre cele două fapte este de preferat un fapt deftemplate:

(locuinta “Bistitei” 32 X1 A 12)

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

Fapte neordonate vs Tabele relationale

Sa consideram urmatorul tabel relational:
Datele din tabelul **tip_mere** poate fi cu usurinta reprezentat in Jess folosind faptele neordonate, grupate intr-o constructie **deffacts**.



tip_mere : Table

	soi	dimensiune	coluare
	Cortland	mare	rosie
	Golden Delicious	mare	galbena
	Red Delicious	medie	rosie
	Granny Smith	mare	verde
*			

```
(deffacts tip_mere ”Sortimente de mere”
    (mar (soi ”Cortland”) (dimensiune mare) (culoare rosie))
    (mar (soi ”Golden Delicious”) (dimensiune mare) (culoare galbena))
    (mar (soi ”Red Delicious”) (dimensiune medie) (culoare rosie))
    (mar (soi ”Granny Smith”) (dimensiune mare) (culoare verde)))
```

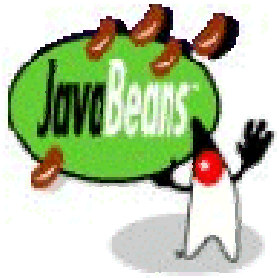
Instante de clase Java in Jess

Putem in Jess sa cream si sa folosim obiecte Java (instante de clase Java):

Jess> (bind ?ht (new java.util.HashMap))
<Java-Object:java.util.HashMap>
Jess> (call ?ht put "key1" "element1")
Jess> (call ?ht put "key2" "element2")
Jess> (call ?ht get "key1")
"*element1*"
Jess> (bind ?pt (new java.awt.Point))
<Java-Object:java.awt.Point>
Jess> (set-member ?pt x 37)
37

Fapte de tip *definstance* – sau obiecte Java incarcate in memoria de lucru

Un fapt de tip definstance (*definstance fact* sau *shadow fact*) este un fapt neordonat ale carui campuri corespund proprietatilor unui obiect *JavaBean*. Slot-urile faptelor neordonate au ca si corespondent in constructiile *JavaBean* datele membru sau proprietatile a caror valoare se poate modifica in timpul ciclului de viata al unui obiect.



Clasa JavaBeans

O clasa Java oarecare poate fi considerata clasa *JavaBean* daca respecta anumite conventii de notatie, de definire a constructorilor si a proprietatilor:

1. clasa trebuie sa aiba un constructor *public default* (fara parametrii)
2. Proprietatile clasei trebuie sa fie accesibile folosind metode de tip *getter* (de exemplu **getValue()**) sau *setter* (**setValue(newValue)**).
3. Clasa trebuie sa fie serializabila (adica sa implementeze interfata **java.io.Serializable**)

Exemplu de clasa JavaBeans

```
public class SimpleBean implements java.io.Serializable{
    /* proprietati */
    private String label = "";
    /* constructor default */
    public SimpleBean() {}

    /* metode getter si setter */
    public String getLabel() {
            return label;
    }
    public void setLabel(String label) {
            this.label = label;
    }
```

Constructorul defclass

Functia Jess care construieste template-uri pe baza definitiei unei clase Java este **defclass** in timp ce functia **definstance** poate fi folosita pentru a incarca in memorie fapte neordonate ce au structura obtinuta prin intermediul unui apel **defclass**. Astfel, faptele construite cu **definstance** pot fi asimilate unor instante de clase Java. Pentru a putea apela functia **defclass** cu numele unei clase Java, fisierul .class trebuie sa fie accesibil lui Jess, adica sa se gaseasca in una din caile specificate in CLASSPATH.

```
Jess> (defclass template_pct java.awt.Point)
java.awt.Point
```

Constructorul definstance (1)

Rezumand, pentru a crea o instanta a unei clase Java ce respecta formalismul *JavaBean*, avem nevoie de un *template* pe baza caruia sa putem construi faptul Jess corespunzator instantei dorite. Un astfel de *template* se defineste folosind functia **defclass** iar pe baza lui se incarca fapte in memorie cu ajutorul comenzii **definstance**, ca in exemplul urmator:

```
Jess> (defclass template_pct java.awt.Point)
java.awt.Point
```

In variabila **?pct** se memoreaza o instanta a clasei Java:

```
Jess> (bind ?pct (new java.awt.Point))
<External-Address:java.awt.Point>
```

iar aceasta instanta este corelata unui fapt Jess de tip **template_pct** folosind functia **definstance** cu modificatorul de acces **static**.

Constructorul definstance (2)

```
Jess> (definstance template_pct ?pct static)
<Fact-0>
Jess> (facts)
f-0    (MAIN::template_pct
                (class <External-Address:java.lang.Class>)
                (location <External-Address:java.awt.Point>)
                (x 0.0)
                (y 0.0)
                (OBJECT <External-Address:java.awt.Point>))
For a total of 1 facts.
```

Observam ca un fapt cu numele **MAIN::template_pct** este incarcat in memoria de lucru, acesta corespunzand clasei **java.awt.Point.**

Rationament bazat pe reguli in Jess

In acest moment stim cum sa populam memoria de lucru cu faptele necesare procesului de rationament. Pe langa acestea, este nevoie de o *baza de cunostinte* care in cazul sistemelor bazate pe reguli este compusa din *regulile de rationament*. O regulă este o colecţie de condiţii şi de acţiuni ce urmează a fi executate dacă condiţiile sunt îndeplinite.

Una din metodele primare de reprezentare a cunoaşterii în Jess este regula. O regulă este o colecţie de condiţii şi de acţiuni ce urmează a fi executate dacă condiţiile sunt îndeplinite. Un sistem expert trebuie să aibă atât reguli, cât şi fapte. Regulile executate sau ‘aprinse’ se bazează pe existenţa sau non-existenţa unor fapte sau instanţe ale unei clase definite de utilizator. Mecanismul (motorul de inferenţă) al Jess-ului aşteaptă potrivirea regulilor cu starea curentă a sistemului (reprezentată de lista de fapte şi lista de instanţe) şi apoi aplică acţiunile.

Definirea regulilor in Jess

Mutimea de reguli a unui program formeaza baza de cunostinte. Aceste constructii sunt mult mai rapide decat instructiunile IF-THEN. Membrul stang (LHS) este format doar din faptele incarcate in memorie, in timp ce membrul drept (RHS) este construit din apeluri de functii.

Observatie. Membrul stang al unei reguli nu poate contine apeluri de functii. Daca dorim sa definim conditia de activare a unei reguli prin una sau mai multe evaluari de functii putem folosi *functia predicativa* ***test***.

Jess aşteaptă îndeplinirea membrului stang al unei reguli verificând faptele cu cele incarcate in memoria de lucru. Atunci când toate faptele sunt validate, regula este pusă în **agenda**. Spunem in acest caz ca regula este *activată*. Agenda reprezintă în Jess ca si in LISP colecţia reguli activate la un moment dat de continutul memoriei

Sintaxa comenzii defrule

(defrule <nume-regula> {;comentariu optional}

<conditie$_1$>

...

<conditie$_n$>

conditiile regulii (LHS)

=>

<actiune$_1$>

...

<actiune$_m$>

concluziile regulii (RHS)

Reguli in Jess. LHS

- Conditiile care apar in membrul stang al unei reguli Jess sunt in mod implicit combinate folosind operatorul **and**.
- Putem insa sa definim premiza unei reguli folosind alte functii booleene:

  Exemplu:

```
(defrule or-not-and example
        (or ((not (a)) (and (b) (c))))
```

Exemple de reguli Jess

```
;; Exemplu de regula de avertizare asupra aparitiei unei erori
(defrule report-error
    (error-is-present)
    =>
    (printout t “There is an error” crlf)
;; Regula ce foloseste pattern bindings pentru a afisa
;; un mesaj mai util utilizatorului
(defrule report-err
    ?err <- (is-error (msg ?msg))
    =>
    (printout t "Error was: " ?msg crlf)
    (retract ?err))
```

Exemple de reguli

Revenind la exemplul din sectiunea anterioara unde au fost definite faptele **locuinta** putem presupune ca exista un dispozitiv GPS care furnizeaza in permanenta numele strazii pe care ne aflam:

IF strada furnizata de GPS este *Independentei*

THEN ofera informatii cu privire la Locuinta1

IF strada furnizata de GPS este *Bistritei*

THEN ofera informatii cu privire la Locuinta2

```
(defrule locuinta1
(locuinta (strada Independentei))
=>
(informatii “Bloc X1, Sc. A, Ap 12”)
```

```
(defrule locuinta2
(locuinta (strada Bistritei))
=>
(informatii “Bloc X2, Sc. B, Ap.12”)
```

Reguli Jess vs. Clauze Prolog

- Sa consideram urmatorul program Prolog:

```
a(25,30).
b(30,35).
c(X,Y):- a(X,Z), b(Z,Y), X<20.
```

- Transpunerea clauzei intr-o regula Jess se face astfel:

```
(defrule find-c
        (a ?x ?z)
        (b ?z ?y)
        (test (< ?x 20))
=>
        (assert (c ?x ?y)))
```

- Definirea memoriei de lucru:

```
(deffacts init-facts (a 25 30) (b 30 35))
```

Alte exemple de reguli Jess ☺

Sa consideram urmatoarul program Jess:

```
(deffacts init-facts (a 1) (b 1))
(defrule print_ab
        (a ?x)
        (b ?y)
=>
        (printout t ″Exista un a(″ ?x ″) si un b(″ ?y ″). ″ crlf))
```

Aceasta regula se va executa o singura data si va afisa:

Jess> Exista un a(1) si un b(1).

Alte exemple de reguli Jess ☺

Dar regula urmatoare, de cate ori va fi activata?

```
(deffacts init-facts (a 1) (b 1))
(defrule print_ab
        ?fapt <- (a ?x)
        (b ?y)
=>
        (printout t ″Exista un a(″ ?x ″) si un b(″ ?y ″). ″ crlf)
        (retract ?fapt)
        (assert (a ?x)))
```

Summary:

- Memoria de lucru Jess.
- Tipuri de fapte Jess.
- Fapte ordonate. Constructorul deffacts.
- Fapte neordonate. Constructorul deftemplate
- Constructorul deffacts.
- Fapte ordonate vs Fapte neordonate
- Fapte definstance. Constructorul definstance si defclass
- Comanda defrule. Sintaxa
- Reguli Jess. Pattern matching