

## CURS 7

### Programare funcțională. Introducere în limbajul LISP

#### Cuprins

Bibliografie .....	1
1. Programare funcțională.....	1
1.1 Evaluare întârziată ( <i>lazy evaluation</i> ) .....	2
2. Limbajul Lisp.....	3
3. Elemente de bază ale limbajului Lisp .....	4
4. Structuri dinamice de date .....	4
4.1 Exemple .....	5
5. Reguli sintactice.....	6
6. Reguli de evaluare.....	7
6.1 Exemple .....	7
7. Funcții Lisp .....	8
(CONS $e_1$ $e_2$ ): l sau pp .....	8
(CAR l sau pp): e .....	8
(CDR l sau pp): e .....	8
7.1 Exemple .....	9

#### Bibliografie

Capitolul 2, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

#### 1. Programare funcțională

- Programare orientată spre valoare, programare aplicativă.
- Se focalizează pe valori ale datelor descrise prin expresii (construite prin definiții de funcții și aplicări de funcții), cu evaluare automată a expresiilor.
- Apare ca o nouă paradigmă de programare.
- Programarea funcțională renunță la instrucțiunea de atribuire prezentă ca element de bază în cadrul limbajelor imperative; mai corect spus această atribuire este prezentă doar la un nivel mai scăzut de abstractizare (analog comparației între **goto** și structurile de control structurate **while**, **repeat** și **for**).
- Principiile de lucru ale programării funcționale se potrivesc cu cerințele programării paralele: absența atribuirii, independența rezultatelor finale de ordinea de evaluare și abilitatea de a opera la nivelul unor întregi structuri.

- **Inteligența artificială** a stat la baza promovării programării funcționale. Obiectul acestui domeniu constă din studiul modului în care se pot realiza cu ajutorul calculatorului comportări care în mod obișnuit sunt calificate ca inteligente. Inteligența artificială, prin problematica aplicațiilor specifice (simularea unor procese cognitive umane, traducerea automată, regăsirea informațiilor) necesită, în primul rând, prelucrări simbolice și mai puțin calcule numerice.
- Caracteristica limbajelor de prelucrare simbolică a datelor constă în posibilitatea manipulării unor structuri de date oricât de complexe, structuri ce se construiesc dinamic (în cursul execuției programului). Informațiile prelucrate sunt de obicei șiruri de caractere, liste sau arbori binari.
- **Pur funcțional** = absența totală a facilităților procedurale - controlul memorării, atribuirii, structuri nerecursive de ciclare de tip FOR
- **CE , NU CUM.**
- Limbaje funcționale – LISP (1958), Hope, ML, Scheme, Miranda, Haskell, Erlang (1995)
  - **Haskell**
    - pur funcțional, concurent
    - aplicații industriale, aplicații web
  - **Erlang** – programare funcțională concurentă
    - aplicații industriale: **telecomunicații**
- LISP nu e PUR funcțional
- Programare funcțională în limbaje precum Python, Scala, F#

## 1.1 Evaluare întârziată (*lazy evaluation*)

**Evaluarea întârziată** (*lazy evaluation*) este una dintre caracteristicile limbajelor funcționale.

- se întârzie evaluarea unei expresii până când valoarea ei e necesară (*call by need*)
  - în limbajul Haskell se evaluează expresiile doar când se știe că e necesară valoarea acestora
- în contrast cu *lazy evaluation* este *eager evaluation* (*strict evaluation*)
  - se evaluează o expresie cand se știe că e posibil să fie folosită valoarea acesteia
- **exemplu:**  $f(x, y) = 2 * x; k=f(d, e)$ 
  - *lazy evaluation* – se evaluează doar  $d$
  - *eager evaluation* – evaluăm  $d$  și  $e$  când calculăm  $k$ , chiar dacă  $y$  nu e folosit în funcție
- majoritatea limbajelor de programare (C, Java, Python) folosesc evaluarea strictă ca și mecanism implicit de evaluare
- Lazy Python

## 2. Limbajul Lisp

- 1958 John McCarthy elaborează o primă formă a unui limbaj (**LISP** - **LISt Processing**) destinat prelucrărilor de liste, formă ce se baza pe ideea transcrierii în acel limbaj de programare a expresiilor algebrice.
  - Câteva caracteristici
    - calcule cu expresii simbolice în loc de numere;
    - reprezentarea expresiilor simbolice și a altor informații prin structura de listă;
    - compunerea funcțiilor ca instrument de formare a unor funcții mai complexe;
    - utilizarea recursivității în definiția funcțiilor;
    - reprezentarea programelor Lisp ca date Lisp;
    - funcția Lisp **eval** care servește deopotrivă ca definiție formală a limbajului și ca interpretor;
    - colectarea spațiului disponibil (*garbage collection*) ca mijloc de tratare a problemei dealocării.
  - Consideră *funcția* ca obiect fundamental – transmis ca parametru, returnat ca rezultat al unei prelucrări, parte a unei structuri de date.
  - Domeniul de utilizare al limbajului Lisp este cel al calculului simbolic, adică al prelucrărilor efectuate asupra unor șiruri de simboluri grupate în expresii. Una dintre cauzele forței limbajului Lisp este posibilitatea de manipulare a unor obiecte cu structură ierarhizată.
  - **Domenii de aplicare** – învățare automată (*machine learning*), bioinformatică, comerț electronic (Paul Graham - <http://www.paulgraham.com/avg.html>), sisteme expert, *data mining*, prelucrarea limbajului natural, agenți, demonstrarea teoremelor, învățare automată, înțelegerea vorbirii, prelucrarea imaginilor, planificarea roboților.
- “Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.” (Paul Graham, 2001)*
- Editorul GNU Emacs de sub Unix e scris în Lisp.
  - **GNU CLisp**, GNU Emacs Lisp.

### 3. Elemente de bază ale limbajului Lisp

- Un program Lisp prelucrează expresii simbolice (*S-expresii*). Chiar programul este o astfel de S-expresie.
- Modul uzual de lucru al unui sistem Lisp este cel conversațional (interactiv), interpretorul alternând prelucrările de date cu intervenția utilizatorului.
- În Lisp există standardul **CommonLisp** și standardul **CLOS** (Common Lisp Object System) pentru programare orientată obiect.
  - **Common Lisp** e un *limbaj funcțional* (dar și imperativ, orientat-obiect, putând fi folosit în manieră declarativă).
- Mecanismul implicit de evaluare în **CommonLisp** – evaluarea strictă (*eager/strict evaluation*).
  - Extensie **CLAZY** – evaluare întârziată (*lazy evaluation*) în Common Lisp
- Verificarea tipurilor se face dinamic (la execuție) – *dynamic type checking*.
- Obiectele de bază în Lisp sunt *atomii* și *listele*.
  - Datele primare (*atomii*) sunt numerele și simbolurile (simbolul este echivalentul Lisp al conceptului de variabilă din celelalte limbaje). Sintactic, simbolul apare ca un șir de caractere (primul fiind o literă); semantic, el desemnează o S-expresie. Atomii sunt utilizați la construirea listelor (majoritatea S-expresiilor sunt liste).
  - O *listă* este o secvență de atomi și/sau liste.
    - Liste *liniare*
    - Liste *neliniare*
- În Lisp s-a adoptat notația *prefixată* (notație ce sugerează și interpretarea operațiilor drept funcții), simbolul de pe prima poziție a unei liste fiind numele funcției ce se aplică.
- *Evaluarea* unei S-expresii înseamnă extragerea (determinarea) valorii acesteia. Evaluarea valorii funcției are loc după evaluarea argumentelor sale.
- În LISP nu există nedeterminism.

### 4. Structuri dinamice de date

- Una dintre cele mai cunoscute și mai simple SDD este *lista liniară simplu înlănțuită* (LLSI).

- Un element al listei este format din două câmpuri: *valoarea* și *legătura* spre elementul următor. Legăturile ne dau informații de natură structurală ce stabilesc relații de ordine între elemente):

valoare	legătură
C1	C2

- Variațiuni ale conținutului acestor două câmpuri generează alte genuri de structuri: dacă C1 conține un pointer atunci se generează arbori binari, iar dacă C2 poate fi și altceva decât pointer atunci apar așa-numitele *perechi cu punct* (elemente cu două câmpuri-dată).
  - se pot astfel forma elemente ale căror câmpuri pot fi în egală măsură ocupate de informații atomice (numere sau simboluri) și de informații structurale (referințe spre alte elemente). Reprezentarea grafică a unei astfel de structuri este cea a unui arbore binar (structura arborescentă).
- Orice listă are echivalent în notația perechilor cu punct**, însă NU orice pereche cu punct are echivalent în notația de listă (în general numai notațiile cu punct în care la stânga parantezelor închise se află NIL pot fi reprezentate în notația de listă). În Lisp, ca și în Pascal, NIL are semnificația de pointer nul.
- Definiția recursivă a echivalenței între **liste** și **perechi cu punct** în Lisp:
  - dacă A este *atom*, atunci lista (A) este echivalentă cu perechea cu punct (A . NIL)
  - dacă lista (l<sub>1</sub> l<sub>2</sub>...l<sub>n</sub>) este echivalentă cu perechea cu punct <p> atunci lista (l<sub>1</sub> l<sub>2</sub>...l<sub>n</sub>) este echivalentă cu perechea cu punct (l . <p>)

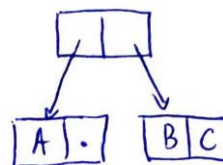
(A B) (A . (B))

#### 4.1 Exemple

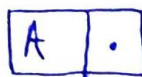
- (A . B) nu are echivalent listă;



- ((A . NIL) . (B . C)) nu are echivalent listă.



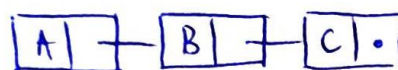
- (A) ↔ (A . NIL)



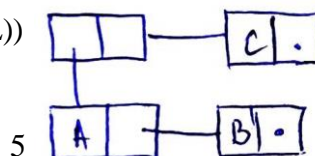
- (A B) ↔ (A . (B))



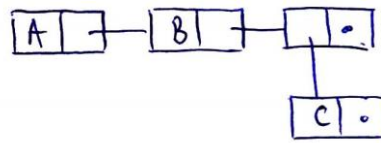
- (A B C) ↔ (A . (B . (C . NIL)))



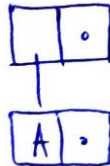
- ((A B) C) ↔ ((A . (B . NIL)) . (C . NIL))



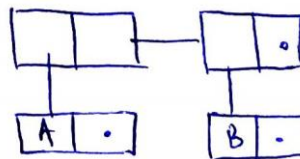
- $(A \ B \ (C)) \leftrightarrow (A \ . \ (B \ . \ ((C \ . \ NIL) \ . \ NIL)))$



- $((A)) \leftrightarrow ((A \ . \ NIL) \ . \ NIL)$



- $((A) \ (B)) \leftrightarrow ((A \ . \ NIL) \ . \ ((B \ . \ NIL) \ . \ NIL))$



**TEMA** Desenați structura fiecăreia din listele de mai jos:

1.  $((A \ . \ NIL) \ . \ ((B \ . \ NIL) \ . \ NIL)) = ((A) \ (B))$
2.  $((NIL)) = ((NIL \ . \ NIL)) \ . \ NIL$
3.  $(( )) = (NIL \ . \ NIL)$
4.  $(A \ (B \ . \ C)) = (A \ . \ ((B \ . \ C) \ . \ NIL))$

## 5. Reguli sintactice

1. **atom numeric** (n) - un șir de cifre urmat sau nu de caracterul '.' și precedat sau nu de semn ('+' sau '-')
2. **atom șir de caractere** (c) - șir de caractere cuprins între ghilimele
3. **simbol** (s) - un șir de caractere, altele decât delimitatorii: spațiu ( ) ' " . virgula = [ ]
  - delimitatorii pot să apară într-un simbol numai dacă sunt evitați (folosind convenția cu "\")
4. **atom** (a) – poate fi
  - n - atom numeric
  - c – atom șir de caractere
  - s – simbol
5. **listă** (l) - poate fi
  - ( ) lista vidă – NIL !!! Lista vida este singura lista care este considerată

atom

- $(e_1 e_2 \dots e_n), n \geq 1$   
unde  $e, e_1, \dots, e_n$  sunt **S-expresii**

6. **pereche cu punct** (pp) – este o construcție de forma  $(e_1 . e_2)$  unde  $e_1$  și  $e_2$  sunt **S-expresii**
7. **S-expresie** (e) – poate fi
  - a - atom
  - l – listă
  - pp – pereche cu punct
8. **formă** (f) - o S-expresie evaluabilă
9. **program Lisp** este o succesiune de **forme** (S-expresii evaluabile).

## 6. Reguli de evaluare

- (a) un **atom numeric** se evaluează prin numărul respectiv
- (b) un **șir de caractere** se evaluează chiar prin textul său (inclusiv ghilimelele);
- (c) o **listă** este evaluabilă (adică este **formă**) doar dacă primul ei element este numele unei funcții/operator, caz în care mai întâi se evaluează toate argumentele, după care se aplică funcția acestor valori.

**Observație** Funcția QUOTE întoarce chiar S-expresia argument, ceea ce este echivalent cu oprirea încercării de a evalua argumentul. În locul lui QUOTE se va putea utiliza caracterul ' (apostrof).

*Esența Lisp-ului constă din prelucrarea S-expresiilor. Datele au aceeași formă cu programele, ceea ce permite lansarea în execuție ca programe a unor structuri de date precum și modificarea unor programe ca și cum ele ar fi date obișnuite.*

### 6.1 Exemple

> 'A A	> (quote A) A
> (A) the function A is undefined	> (NIL) the function NIL is undefined
> '(A) (A)	> '(NIL) (NIL)
> () NIL	> () the function NIL is undefined
> NIL NIL	> '() ()
> (A.B) the function A\B is undefined	> '(A.B) (A\B)

## 7. Funcții Lisp

Prelucrările de liste se pot face la:

- nivel superficial
- la orice nivel

**Exemplu.** Să se calculeze suma atomilor numerici dintr-o listă neliniară

- la nivel superficial (suma '(1 (2 a (3 4) b 5) c 1))  $\rightarrow 2$
- la toate nivelurile (suma '(1 (2 a (3 4) b 5) c 1))  $\rightarrow 16$

**(CONS e<sub>1</sub> e<sub>2</sub>): l sau pp**

- funcția **constructor**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- formează o pereche cu punct având reprezentările celor două SE în cele două câmpuri. Elementul CONS construit în acest fel se numește *celulă CONS*. Valorile argumentelor nu sunt afectate.

Iată câteva exemple:

- (CONS 'A 'B) = (A . B)
- (CONS 'A '(B)) = (A B)
- (CONS '(A B) '(C)) = ((A B) C)
- (CONS '(A B) '(C D)) = ((A B) C D)
- (CONS 'A '(B C)) = (A B C)
- (CONS 'A (CONS 'B '(C))) = (A B C)

**(CAR l sau pp): e**

- se evaluează argumentul și apoi se trece la evaluarea funcției
- extrage primul element al unei liste sau partea stângă a unei perechi cu punct

**(CDR l sau pp): e**

- se evaluează argumentul și apoi se trece la evaluarea funcției
- extrage lista fără primul element sau respectiv partea dreaptă a unei perechi cu punct.

Iată câteva exemple:

- (CAR '(A B C)) = A
- (CAR '(A . B)) = A
- (CAR '((A B) C D)) = (A B)
- (CAR (CONS '(B C) '(D E))) = (B C)
- (CDR '(A B C)) = (B C)
- (CDR '(A . B)) = B



- $(\text{CDR '}((A\ B)\ C\ D)) = (C\ D)$
- $(\text{CDR (CONS '(B\ C) '(D\ E))} = (D\ E)$

CONS reface o listă pe care CAR și CDR au secționat-o. De observat, însă, că obiectul obținut ca urmare a aplicării funcțiilor CAR, CDR și CONS, nu este același cu obiectul de la care s-a plecat:

- $(\text{CONS (CAR '(A\ B\ C)) (CDR '(A\ B\ C)))} = (A\ B\ C)$
- $(\text{CAR (CONS 'A '(B\ C)))} = A$
- $(\text{CDR (CONS 'A '(B\ C)))} = (B\ C)$

La folosirea repetată a funcțiilor de selectare se poate utiliza prescurtarea  $C_{x_1x_2...x_n}R$ , echivalentă cu  $C_{x_1}R$  o  $C_{x_2}R$  o ... o  $C_{x_n}R$ , unde caracterele  $x_i$  sunt fie 'A', fie 'D'. În funcție de implementări, se vor putea utiliza în această compunere cel mult trei sau patru  $C_{x_i}R$ :

- $(\text{CAADDR '}((A\ B)\ C\ (D\ E))) = D$
- $(\text{CDAAAR '}(((A)\ B)\ C)\ (D\ E))) = \text{NIL}$
- $(\text{CAR '(CAR (A\ B\ C)))} = \text{CAR}$

## 7.1 Exemple

### Operare CLisp

- (ed) – editor
- se scrie funcția **fct** în fișierul **f.lsp**. De exemplu, fișierul **f.lsp** conține următoarea definiție  

```
(defun fct(l)
  (cdr l)
)
```
- se încarcă fișierul **f.lsp**  

```
> (load 'f)
```
- dacă nu sunt erori, se evaluează funcția **fct**  

```
> (fct '(1 2)) → (2)
```

**EXEMPLU 7.1.1** Să se genereze o listă cu numerele întregi din intervalul [a, b].

```
(defun interval (a b)
  ;; returnează o listă cu numerele întregi din intervalul [a, b]
  (if (> a b)
      nil
      (cons a (interval (+ a 1) b)))
)
```

**EXEMPLU 7.1.2** Să se calculeze suma atomilor numerici de la nivelul superficial dintr-o listă neliniară .

Model recursiv

$$suma(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + suma(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ suma(l_2 \dots l_n) & \text{altfel} \end{cases}$$

; suma atomilor de la nivelul superficial al unei liste neliniare

; (suma '(1 (2 (3 4) 5) 1)) → 2

(defun suma(l)

; **forma COND** – forma condițională: permite ramificarea prelucrărilor

(cond

((null l) 0)

; **NUMBERP** – returnează T dacă argumentul e număr

((numberp (car l)) (+ (car l) (suma (cdr l)))))

(t (suma (cdr l))))

)

)

**EXEMPLU 7.1.2** Să se calculeze suma atomilor numerici de la orice nivel dintr-o listă neliniară.

Model recursiv

$$suma(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + suma(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ suma(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom} \\ suma(l_1) + suma(l_2 \dots l_n) & \text{altfel} \end{cases}$$

; să se calculeze suma atomilor numerici dintr-o listă neliniară

; (la toate nivelurile) (suma '(1 (2 a (3 4) b 5) c 1)) → 16

(defun suma(l)

(cond

((null l) 0)

((numberp (car l)) (+ (car l) (suma (cdr l)))))

; **ATOM** – returnează T dacă argumentul e atom

((atom (car l)) (suma (cdr l)))

; **ultima clauză e pentru primul element listă**

(t (+ (suma (car l)) (suma (cdr l)))))

)

)