CURS 7

Programare funcțională. Introducere în limbajul LISP

Cuprins

Bibliografie

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

Programare logică

1. "*Learning Prolog provides you with a different perspective on programming that, once understood, can be applied to other languages as well.*"
2. "*You should learn Prolog as a part of your personal quest to become a better programmer.*"
3. Contraint programming
4. Inductive logic programming

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 funcţională renunţă la instrucţiunea 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 comparaţiei între **goto** şi structurile de control structurate **while**, **repeat** şi **for**).

- Principiile de lucru ale programării funcţionale se potrivesc cu cerinţele programării paralele: absenţa atribuirii, independenţa rezultatelor finale de ordinea de evaluare şi abilitatea de a opera la nivelul unor întregi structuri.

- **Inteligenţa artificială** a stat la baza promovării programării funcţionale. Obiectul acestui domeniu constă din studiul modului în care se pot realiza cu ajutorul calculatorului comportări care în mod obişnuit sunt calificate ca inteligente. Inteligenţa artificială, prin problematica aplicaţiilor specifice (simularea unor procese cognitive umane, traducerea automată, regăsirea informaţiilor) necesită, în primul rând, prelucrări simbolice şi mai puţin 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 execuţiei programului). Informațiile prelucrate sunt de obicei şiruri de caractere, liste sau arbori binari.

- **Pur funcțional** = absența totală a facilităților procedurale - controlul memorării, atribuiri, structuri nerecursive de ciclare de tip FOR

- **CE** , NU **CUM**.

- Limbaje funcționale – LISP (1958), Hope, ML, Scheme, Miranda, Haskell, Erlang (1995)
    - **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 cu *lazy evaluation* este *eager evaluation* (*strict evaluation*)
    - se evaluează o expresie cans 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 evaularea strictă ca și mecanism implicit de evaluare

- Lazy Python

2. Limbajul Lisp

- 1958 John McCarthy elaborează o primă formă a unui limbaj (**LISP** - **LIS**t **P**rocessing) 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 şi a altor informaţii prin structura de listă;
    - compunerea funcţiilor ca instrument de formare a unor funcţii mai complexe;
    - utilizarea recursivităţii în definiţia funcţiilor;
    - reprezentarea programelor Lisp ca date Lisp;
    - funcţia Lisp **eval** care serveşte deopotrivă ca definiţie formală a limbajului şi ca interpretor;
    - colectarea spaţiului disponibil (*garbage collection*) ca mijloc de tratare a problemei dealocării.

- Consideră *funcţia* 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 şiruri de simboluri grupate în expresii. Una dintre cauzele forţei 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 conversaţional (interactiv), interpretorul alternând prelucrarile de date cu intervenţia utilizatorului.

- În Lisp există standardul **CommonLisp** și standardul **CLOS** (Commom Lisp Object System) pentru programare orientată obiect.

- Mecanismul implicit de evaluare în **CommonLisp** – evaluarea strictă (*eager/strict evaluation*).
    - Biblioteca CLAZY – evaluarea întârziată (*lazy evaluation*)

- Verificarea tipurilor se face dinamic (la execuție) – *dynamic type checking*.

- Obiectele de bază în Lisp sunt *atomii* şi *listele*.

    - Datele primare (*atomii*) sunt numerele şi simbolurile (simbolul este echivalentul Lisp al conceptului de variabilă din celelalte limbaje). Sintactic, simbolul apare ca un şir de caractere (primul fiind o literă); semantic, el desemnează o S-expresie. Atomii sunt utilizaţi la construirea listelor (majoritatea S-expresiilor sunt liste).
    - O *listă* este o secvenţă de atomi şi/sau liste.
        - Liste *liniare*
        - Liste *neliniare*

- In Lisp s-a adoptat notaţia *prefixată* (notaţie ce sugerează şi interpretarea operaţiilor drept funcţii), simbolul de pe prima poziţie a unei liste fiind numele funcţiei ce se aplică.

- *Evaluarea* unei S-expresii înseamnă extragerea (determinarea) valorii acesteia. Evaluarea valorii funcţiei are loc după evaluarea argumentelor sale.

- În LISP nu există nedeterminism.

4. Structuri dinamice de date

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

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

valoare	legătură
C1	C2

- Variaţiuni ale conţinutului acestor două câmpuri generează alte genuri de structuri: dacă C1 conţine un pointer atunci se generează arbori binari, iar dacă C2 poate fi şi altceva decât pointer atunci apar aşa-numitele *perechi cu punct* (elemente cu două câmpuri-dată).
    - se pot astfel forma elemente ale căror câmpuri pot fi în egală măsură ocupate de informaţii atomice (numere sau simboluri) şi de informaţii structurale (referinţe spre alte elemente). Reprezentarea grafică a unei astfel de structuri este cea a unui arbore binar (structura arborescentă.
    - **Observaţie**: Orice LLSI se poate reprezenta ca arbore (fiind un caz particular al acestuia), însă nu orice arbore se poate reprezenta ca o listă!

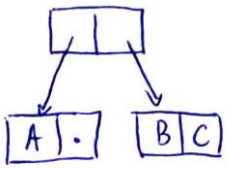
- **Orice listă are echivalent în notaţia perechilor cu punct**, însă NU orice pereche cu punct are echivalent în notaţia de listă (în general numai notaţiile cu punct în care la stânga parantezelor închise se află NIL pot fi reprezentate în notaţia de listă). În Lisp, ca şi în Pascal, NIL are semnificaţia de pointer nul.

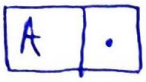
- Definiţia recursivă a echivalenţei între liste şi perechi cu punct în Lisp:

    a). dacă A este *atom*, atunci lista (A) este echivalentă cu perechea cu punct (A **.** NIL)
    b). dacă lista (l_1 l_2…l_n) este echivalentă cu perechea cu punct <**p**> atunci lista (l $l_1$$l_2$…$l_n$) este echivalentă cu perechea cu punct (l **.** <**p**>)

4.1 Exemple

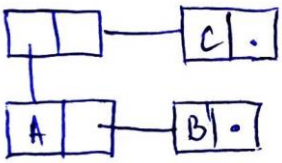
- (A . B) nu are echivalent listă; 

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

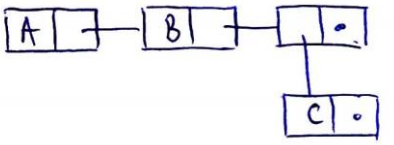
- (A) = (A . NIL) 

- (A B) = (A . (B)) 

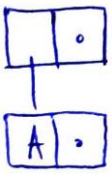
- (A B C) = (A . (B . (C . NIL))) 

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

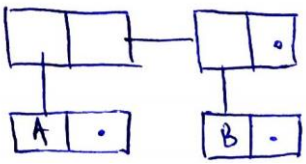
- (A B (C)) = (A . (B . ((C . NIL) . NIL)))



- ((A)) = ((A . NIL) . NIL)



- ((A) (B)) =((A . NIL) . ((B . NIL) . NIL)



TEMĂ 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 şir de cifre urmat sau nu de caracterul '.' şi precedat sau nu de semn ('+' sau '-')

2. **atom şir de caractere** (c) - şir de caractere cuprins între ghilimele

3. **simbol** (s) - un şir de caractere, altele decât delimitatorii: spaţiu ( ) ' " . virgula = [ ]
   - delimitatorii pot să apară într-un simbol numai dacă sunt evitaţi (folosind convenţia cu "\")
4. **atom** (a) – poate fi
   - n - atom numeric
   - c – atom şir de caractere
   - s – simbol

5. **listă** (l) - poate fi
   - () lista vidă - NIL
   - (e)

- $(e_1\ e_2\ ...\ e_n)$, n>1

unde e, e_1,...,e_n sunt **S-expresii**

6. **pereche cu punct** (pp) – este o construcție de forma $(e_1\ .\ e_2)$ unde e_1 şi 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 **şir 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 funcţii, caz în care mai întâi se evaluează toate argumentele, după care se aplică funcţia acestor valori.

Observaţie Funcţia 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).

Esenţa Lisp-ului constă din prelucrarea S-expresiilor. Datele au aceeaşi formă cu programele, ceea ce permite lansarea în execuţie ca programe a unor structuri de date precum şi modificarea unor programe ca şi cum ele ar fi date obişnuite.

6.1 Exemple

> 'A A	> (quote A) A

> (A) undefined function A	> (NIL) undefined function NIL
> ' (A) (A)	> ' (NIL) (NIL)
> () NIL	> (()) undefined function NIL

> NIL NIL	> ' (()) (())
> (A.B) undefined function A\.B	> ' (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)) → 2
- la toate nivelurile (suma ' (1 (2 a (3 4) b 5) c 1)) → 16

(CONS e_1 e_2): l sau pp construieste o lista/pereche cu punct

- funcția **constructor**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- formează o perechi cu punct având reprezentările celor două SE în cele două câmpuri. Elementul CONS construit în acest fel se numeşte *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 da capul listei/partea stanga a unei perechi cu punct

- 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 partea dreapta a unei perechi cu punct/tail la lista

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

CONS reface o listă pe care CAR si CDR au secţionat-o. De observat, însă, că obiectul obţinut ca urmare a aplicării funcţiilor CAR, CDR şi CONS, nu este acelaşi cu obiectul de la care s-a plecat:

- (CONS (CAR '(A B C)) (CDR '(A B C))) = (A B C)
- (CAR (CONS 'A '(B C))) = A
- (CDR (CONS 'A '(B C))) = (B C)

La folosirea repetată a funcţiilor de selectare se poate utiliza prescurtarea $Cx_1x_2...x_nR$, echivalentă cu Cx_1R o Cx_2R o ... o Cx_nR, unde caracterele x_i sunt fie 'A', fie 'D'. In funcţie de implementări, se vor putea utiliza în această compunere cel mult trei sau patru Cx_iR:

- (CAADDR '((A B) C (D E))) = D
- (CDAAAR '((((A) B) C) (D E))) = NIL
- (CAR '(CAR (A B C))) = 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 calculeze suma atomilor numerici de la nivelul superficial dintr-o listă neliniară .

Model recursiv

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

```
; 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)        null - testeaza daca lista este vida     ; daca lista este vida, returnezi 0
    ; NUMBERP – returnează T dacă argumentul e număr      !!!!!!
    ((numberp (car l)) (+ (car l) (suma (cdr l))))
    (t (suma (cdr l)))              t - true; daca niciuna din clauze nu a fost true, intra pe asta
  )
)
```

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

Model recursiv

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

```
; 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))))
  )
)
```