CURS 9

Funcțiile SET, SETQ, SETF. Arbori binari. Exemple

Cuprins

1. SET, SETQ, SETF

Pentru a da valori simbolurilor în Lisp se folosesc funcţiile cu **efect secundar** SET şi SETQ.

Acţiunea prin care o funcţie, pe lângă calculul valorii sale, realizează şi modificări ale structurilor de date din memorie se numeşte ***efect secundar***.

(SET s_1 f_1 ... s_n f_n): e

- se evaluză argumentele și apoi se trece la evaluarea funcției
- efect:
    - valorile argumentelor de rang par (f_i) devin valorile argumentelor de rang impar corespunzătoare evaluate în prealabil la simboluri (s_i)
- rezultatul întors valoarea ultimei forme evaluate (f_n)

Iată câteva exemple

- (SET 'X 'A) = A — X se evaluează la A
- (SET X 'B) = B — A se evaluează la B
- (SET 'X (CONS (SET 'Y X) '(B))) = (A B) — X := (A B) şi Y := A
- (SET 'X 'A 'L (CDR L)) — X := A şi L := (CDR L)

(SETQ s_1 f_1 ... s_n f_n): e

- se evalueaza doar formele $f_1,...,f_n$
- efect:
    - valorile argumentelor de rang par (f_i) devin valorile argumentelor de rang impar corespunzătoare neevaluate (s_i)
- rezultatul întors valoarea ultimei forme evaluate (f_n)

Iată câteva exemple

- (SETQ X 'A) = A
- (SETQ A '(B C)) = (B C)
- (CDR A) = (C)

- (SETQ X (CONS X A)) = (A B C) X se evaluează la (A B C)

Lista vidă este singurul caz de listă care are un nume simbolic, NIL, iar NIL este singurul atom care se trateaza ca şi lista, () şi NIL reprezentând acelaşi obiect (elementul NIL are în campul CDR un pointer spre el insuşi, iar in CAR are NIL).

- (CDR NIL) = NIL
- (CAR NIL) = NIL

Observaţie. Atentie la diferenta dintre () = NIL, lista vidă, şi (()) = (NIL), listă ce are ca singur element pe NIL.

- (CONS NIL NIL) = (())

(SETF f_1 f'_1 ... f_n f'_n): e

- este un macro, are efect destructiv
- $f_1,\ldots,f_n$ sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp
- $f'_1,\ldots,f'_n$ sunt forme ale căror valori vor fi legate de locaţiile desemnate de parametrii f_1, ..., f_n corespunzători.
- efect:
  - se evaluează formele de rang par (f'_i) şi valorile acestora se leagă de locațiile desemnate de formele f_i corespunzătoare
- rezultatul întors valoarea ultimei forme evaluate (f'_n)

De remarcat că Lisp oferă pe lângă funcţiile SET şi SETQ macro-ul SETF. Parametrii p_1, ..., p_n sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp, iar e_1, ..., e_n sunt forme ale căror valori vor fi legate de locaţiile desemnate de parametrii p_1, ..., p_n corespunzători. Rezultatul întors de SETF este valoarea ultimei expresii evaluate, e_n.

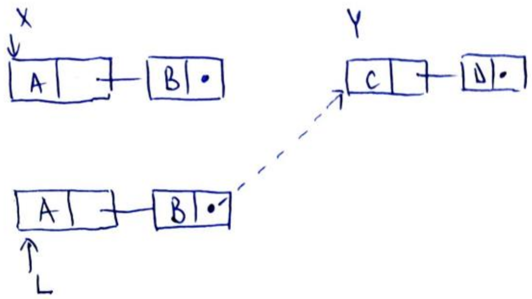
Pentru a ne lămuri în legătură cu modul de operare a lui SETF, să vedem următorul exemplu:

- (SETQ A ‘(B C D))
  - în acest moment A se evaluează la (B C D)
- (SETF (CADR A) ‘X)
  - efectul este înlocuirea lui (CADR A) cu X
- în acest moment A se evaluează la (B X D)
  - (SET (CADR A) ‘Y)
- efectul este evaluarea lui (CADR A) la X şi iniţializarea simbolului X la valoarea Y
- în acest moment A se evaluează la (B X D)
- în acest moment X se evaluează la Y

- (SETQ X '(A B)) X se evaluează la (A B)
- (SETQ Y '(A B)) Y se evaluează la (A B)
- (SETF (CAR X) 'C) X se evaluează la (C B), Y se va evalua tot la (A B)

!!! Atenție la funcția APPEND

- (SETQ X '(A B)) X se evaluează la (A B)
- (SETQ Y '(C D)) Y se evaluează la (C D)
- (SETQ L (APPEND X Y)) L se evaluează la (A B C D)



- (SETF (CAR X) 'E) X se evaluează la (E B)
- L ??
- (SETF (CADR Y) 'F) Y se evaluează la (C F)
- L ??

2. Arbori binari

Un arbore binar se poate memora sub forma unei liste, în următoarele două moduri:

V1 Un arbore având
- rădăcină,
- subarborii subarbore-stâng și subarbore-drept

se va reprezenta sub forma unei liste neliniare de forma

(rădăcină lista-subarbore-stâng lista-subarbore-drept)

unde
- lista-subarbore-stâng reprezintă lista asociată (în memorarea sub forma V1) a subarborelui stâng al nodului rădăcină
- lista-subarbore-drept reprezintă lista asociată (în memorarea sub forma V1) a subarborelui drept al nodului rădăcină

De exemplu arborele

```
      a
    /   \
   b     d
    \   /
     f e
```

se reprezintă, în varianta **V1** sub forma listei (a (b () (f)) (d (e)))

Reprezentarea V1 este cea mai potrivită pentru reprezentarea sub formă de listă a unui arbore cu rădăcină, fiind adecvată definiției recursive a unui arbore (binar).

V2 Un arbore având

- rădăcină,
- nr-arbori subarbori
- subarborii subarbore-stâng și subarbore-drept

se va reprezenta sub forma unei liste liniare de forma

(rădăcină nr-subarbori lista-subarbore-stâng lista-subarbore-drept)

unde

- lista-subarbore-stâng reprezintă lista asociată (în memorarea sub forma V2) subarborelui stâng al nodului rădăcină
- lista-subarbore-drept reprezintă lista asociată (în memorarea sub forma V2) subarborelui drept al nodului rădăcină

Dezavantajul reprezentării **V2** este dat faptul că nu este potrivită pentru un arbore ordonat (de ex., în cazul arborelui binar nu se face distincție între subarborele stâng și cel drept).

De exemplu arborele

```
        a
       / \
      b   d
     / \ / \
    c  f e  h
   /  /
  i  g
```

se reprezintă, în varianta **V2** sub forma listei (a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)

Observație. Reprezentările V1 și V2 pot fi generalizate pentru arbori *n*-ari, varianta V1 fiind cea mai potrivită.

3. Exemple

EXEMPLU 3.1 Se dă un arbore binar reprezentat în V1 (a se vedea Secțiunea 1). Să se determine lista liniară a nodurilor obținute prin parcurgerea arborelui în inordine.

(inordine '(a (b () (f)) (d (e)))) = (b f a e d)

În reprezentarea unui AB în varianta V1, sub forma unei liste *l* de forma (rădăcină lista-subarbore-stâng lista-subarbore-drept), observăm următoarele

- (car l) – primul element al listei este rădăcina arbrelui
- (cadr l) – al doilea element al listei, la nivel superficial, este subarborele stâng

- (caddr l) – al treilea element al listei, la nivel superficial, este subarborele drept

Model recursiv

$$inordine(\ l_1\ l_2 l_3) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ inordine(\ l_2) \oplus\ l_1 \oplus inordine(l_3) & altfel \end{cases}$$

```
(defun inordine(l)
        (cond
                ((null l) nil)
                (t (append (inordine (cadr l)) (list (car l)) (inordine (caddr l))))
               ; (t (append (inordine (cadr l)) (cons (car l) (inordine (caddr l)))))
        )
)
```

EXEMPLU 3.2 Se dă un arbore binar reprezentat în V2 (a se vedea Secțiunea 2). Să se determine lista nodurilor obținute prin parcurgerea arborelui în inordine.

În reprezentarea unui AB în V2, față de reprezentarea V1, nu este clar identificat subarborele stâng și subarborele drept, pentru ca AB să poată fi prelucrat recursiv.

<u>Idee:</u> Dacă am reuși să extragem lista corespunzătoare (reprezentării în V2) subarborilor stâng și drept, problema s-ar reduce la cea prezentată în EXEMPLU 3.3.

Vom folosi o funcție auxiliară pentru a determina subarborele stâng al unui AB reprezentat în V2 (presupunem reprezentarea corectă).

(stang ' (a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (b 2 c 1 i 0 f 1 g 0)

Vom folosi o funcție auxiliară, **parcurg_st**, care va parcurge lista începând cu al 3-lea element și care va returna subarborele stâng. **<u>Idee:</u>** se colectează elementele, începând cu al 3 -lea element al listei, ...până când?

(parcurg_st ' (b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (b 2 c 1 i 0 f 1 g 0)

$$stang(l_1\ l_2 \dots l_n) = parcurg\_st(l_3 \dots l_n, 0, 0)$$

nv – număr vârfuri
nm – număr muchii

$$parcurg\_st(l_1\ l_2 \dots l_k, nv, nm) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ \emptyset & daca\ \ nv = 1 + nm \\ l_1 \oplus\ l_2 \oplus parcurg\_st(l_3 \dots l_k, nv + 1, nm + l_2) & altfel \end{cases}$$

```
(defun parcurg_st(arb nv nm)
      (cond
           ((null arb) nil)
           ((= nv (+ 1 nm)) nil)
           (t (cons (car arb) (cons (cadr arb) (parcurg_st (cddr arb) (+ 1 nv) (+ (cadr arb) nm)))))
      )
)

(defun stang(arb)
  (parcurg_st (cddr arb) 0 0)
)
```

Temă. Scrieți funcția pentru determinarea subarborelui drept.
1) Se poate folosi aceeași idee ca la parcurgerea pentru subarborele stâng, doar că în momentul în care nv=1+nm, se va returnma lista rămasă
2) Pentru a reduce complexitatea timp, se poate defini o sigură funcție cae determină atât subarborele stâng, cât și cel drept.

(drept ' (a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (d 2 e 0 h 0)

Observație. Se poate folosi o funcție care să returneze atât subarborele stâng, cât și subarborele drept.

Având funcțiile anterioare care determină subarborele stâng și cel drept, lista nodurilor obținute prin parcurgerea arborelui în inordine se va face similar modelului recursiv descris în **EXEMPLU 3.1.**

$$inordine(l_1\, l_2 \dots l_n) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ inordine(stang(l_1\, l_2 \dots l_n)) \oplus\ l_1 \oplus inordine(drept(l_1\, l_2 \dots l_n)) & altfel \end{cases}$$

EXEMPLU 3.3 Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se determine lista (mulțimea) submulțimilor listei.

(subm '(1 2)) → (() (2) (1) (1 2))

? Cum obținem lista submulțimilor (1 2) dacă știm să generăm lista submulțimilor listei (2)? (() (2))

Idee: Dacă lista e vidă, submulţimea sa e lista vidă. Pentru determinarea submulţimilor unei liste [E|L], care are capul E şi coada L, vom proceda în felul următor:

i. determină o submulţime a listei L
ii. plasează elementul E pe prima poziţie într-o submulţime a listei L

Vom folosi o funcție auxiliară **insPrimaPoz**(*e*, *l*) care are ca parametri un element *e* si o listă *l* de liste liniare și returnează o copie a listei *l* în ale cărei liste se inserează *e* pe prima poziție.

(insPrimaPoz '3 '(() (2) (1) (1 2)) → ((3) (3 2) (3 1) (3 1 2))

; *l* este o listă de liste liniare
; se inserează *e* pe prima poziție, în fiecare listă din *l*, și se returnează rezultatul

$$insPrimaPoz(e, l_1\, l_2 \ldots l_n) = \begin{cases} \emptyset & daca\ lista\ e\ vida \\ (e, l_1) \oplus insPrimaPoz(e, l_2 \ldots l_n) & altfel \end{cases}$$

```
(defun insPrimaPoz(e l)
	(cond
		((null l) nil)
		(t (cons (cons e (car l)) (insPrimaPoz e (cdr l))))
	)
)
```

; *l* este o mutime reprezentată sub forma unei liste liniare

$$subm(l_1\, l_2 \ldots l_n) = \begin{cases} (\emptyset) & daca\ lista\ e\ vida \\ subm(l_2 \ldots l_n) \oplus insPrimaPoz(l_1, subm(l_2 \ldots l_n)) & altfel \end{cases}$$

```
(defun subm(l)
	(cond
		((null l) (list nil))
		(t (append (subm (cdr l)) (insPrimaPoz (car l) (subm (cdr l)))))
	)
)
```

În codul Lisp scris anterior observăm faptul că apelul recursiv (subm (cdr l) din cea de-a doua clauză COND se repetă, ceea ce evident, nu este eficient din perspectiva complexității timp. O soluție pentru evitarea acestui apel repetat va f folosire unei funcții anonime LAMBDA (se va discuta în Cursul 10).

De asemenea, în Cursul 11 vom discuta despre simplificarea implementării funcției subm, renunțând la utilizarea funcției auxiliare, folosind o funcție MAP.

EXEMPLU 3.4 Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se determine lista (mulțimea) permutărilor listei inițiale.

(PERMUTĂRI '(1 2 3)) → ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

? Cum obţinem lista permutărilor (1 2 3) dacă ştim să generăm lista permutărilor listei (2 3)? ((2 3) (3 2))

Idee: Dacă lista e vidă, lista permutărilor sale este lista vidă. Pentru determinarea permutărilor unei liste [E|L], care are capul E şi coada L, vom proceda în felul următor:

1. determină o permutare L1 a listei L;
2. plasează elementul E pe toate poziţiile listei L1 şi produce în acest fel lista X care va fi o permutare a listei iniţiale [E|L].

Vom folosi câteva funcţii auxiliare:

1) o funcţie care returnează lista obţinută prin inserarea unui element **E** pe o anumită poziţie **N** într-o listă **L** (1 ≤ N ≤ lungimea listei **L**+1).

(INS '1 2 '(2 3)) = (2 1 3)
(INS '1 3 '(2 3)) = (2 3 1)

Modelul recursiv

$$ins(e, n, l_1\ l_2 \dots l_k) = \begin{cases} (e\ \ l_1\ l_2 \dots l_k) & daca\ n = 1 \\ l_1 \oplus ins(e, n-1, l_2 \dots l_k) & altfel \end{cases}$$

```
(DEFUN INS (E N L)
        (COND
                ((= N 1) (CONS E L))
                (T (CONS (CAR L) (INS E (- N 1) (CDR L))))
        )
)
```

)

2) o funcţie care să returneze mulţimea formată din listele obţinute prin inserarea unui element **E** pe poziţiile **1, 2,…, lungimea listei L+1** într-o listă **L**.

(INSERARE '1 '(2 3)) = ((2 3 1) (2 1 3) (1 2 3))

Observaţie: Se va folosi o funcţie auxiliară (INSERT E N L) care returnează mulţimea formată cu listele obţinute prin inserarea unui element pe poziţiile **N, N-1, N-2,....,1** în lista **L**.

(INSERT '1 2 '(2 3)) = ((2 1 3) (1 2 3))

$$insert(e,n,l_1\ l_2 \dots l_k) = \begin{cases} \emptyset & daca\ n = 0 \\ ins(e,n,l_1\ l_2 \dots l_k) \oplus insert(e,n-1,l_1 l_2 \dots l_k) & altfel \end{cases}$$

```
(DEFUN INSERT (E N L)
        (COND
                ((= N 0) NIL)
                (T (CONS (INS E N L) (INSERT E (- N 1) L)))
        )
)
```

$$inserare(e,l_1 l_2 \dots l_n) = insert(e,n+1,l_1 l_2 \dots l_n)$$

```
(DEFUN INSERARE (E L)
        (INSERT E  (+ (LENGTH L) 1)  L)
)
```

Temă Continuați implementarea pentru funcția PERMUTĂRI.

EXEMPLU 3.5. Se o listă liniară numerică. Să se determine lista sortată, folosind sortarea arborescentă (un ABC intermediar).

(sortare '(5 1 4 6 3 2)) = (1 2 3 4 5 6)

Pentru sortarea arborescentă a unei liste, vom proceda astfel:

1. Construim un ABC intermediar cu elementele listei inițiale
    - pentru memorarea ABC vom face folosi o listă liniară – varianta V1 (Secțiunea 1)
    - pentru construirea ABC intermediar vom avea nevoie de inserarea unui element în arbore
2. Parcurgerea în inordine a arborelui construit anterior ne va furniza lista inițială ordonată.

Model recursiv – inserarea unui element în ABC

$$inserare(e,l_1\ l_2 l_3) = \begin{cases} (e) & daca\ l\ e\ vida \\ l_1 \oplus inserare(e,l_2) \oplus\ l_3 & daca\ e \le l_1 \\ l_1 \oplus l_2 \oplus inserare(e,l_3) & altfel \end{cases}$$

```
(defun inserare(e arb)
        (cond
                ((null arb) (list e))
```

```
                ((<= e (car arb)) (list (car arb) (inserare e (cadr arb)) (caddr arb)))
                (t (list (car arb) (cadr arb) (inserare e (caddr arb))))
        )
)
```

<u>Model recursiv</u> – construirea ABC din listă

$$construire(l_1\ l_2 \ldots l_n) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ inserare(l_1 construire(\ l_2 \ldots l_n)) & altfel \end{cases}$$

```
(defun construire(l)
        (cond
                ((null l) nil)
                (t (inserare (car l) (construire (cdr l))))
        )
)

; lista nodurilor unui ABC parcurs în inordine
(defun inordine (arb)
        (cond
                ((null arb) nil)
                (t (append (inordine (cadr arb)) (list (car arb)) (inordine (caddr arb))))
        )
)

; sortarea arborescentă a listei
(defun sortare(l)
        (inordine (construire l))
)
```

Care este complexitatea timp a sortării arborescente?

<u>Temă</u> Se dă un arbore binar, ale cărui noduri sunt distincte, reprezentat sub forma unei liste neliniare de forma (<u>rădăcină</u> <u>lista-subarbore-stâng</u> <u>lista-subarbore-drept</u>). Să se determine o listă liniară reprezentând calea de la rădăcină către un nod *e* dat.

De exemplu :

- (setq arb '(a (b () (f)) (d (e () (k)) (l))))

    arborele

```
        a
      /   \
     b     d
      \   / \
       f e   l
          \
           k
```

(cale 'm arb) → NIL
(cale 'f arb) → (a b f)
(cale 'k arb) → (a d e k)

$$
cale(e, l_1\ l_2 l_3) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ (e) & daca\ e = \ l_1 \\ l_1 \oplus cale(e, l_2) & daca\ e \in l_2 \\ l_1 \oplus cale(e, l_3) & daca\ e \in l_3 \\ \emptyset & altfel \end{cases}
$$