

Seminar 6 – Funcții MAP

- Funcțiile MAP sunt funcții care au ca efect aplicarea în mod repetat a unei funcții asupra parametrului/parametrilor. În Lisp există mai multe funcții MAP, în seminarul curent focusându-ne pe funcția MAPCAR, care este cel mai frecvent folosită.
- Să presupunem că am implementat o funcție care primește ca și parametru un număr și returnează numărul înmulțit cu 3.

```
(DEFUN triple (x) (* x 3))
```

- Folosind funcția MAPCAR putem aplica această funcție asupra fiecărui element dintr-o listă. De exemplu: (MAPCAR #'triple '(1 2 3 4 5)). Acest apel este echivalent cu apelul funcției *triple*, pe rând, independent, pentru fiecare element din lista dată ca parametru. Fiecare apel pentru *triple* va returna o valoare, iar aceste valori sunt colectate într-o listă de MAPCAR. Astfel, apelul (MAPCAR #'triple '(1 2 3 4 5)) este echivalent cu (list (triple 1) (triple 2) (triple 3) (triple 4) (triple 5)). Rezultatul va fi lista (3 6 9 12 15).
- Ce se întâmplă dacă lista conține și atomi nenumeriți? De exemplu, dacă apelăm (MAPCAR #'triple '(1 a 2 b 3 c))? Rezultatul va fi o eroare, pentru că funcția *triple* va da o eroare pentru un parametru care nu este număr: *argument to * should be a number: A*. Ca o soluție putem modifica funcția noastră *triple*, dacă parametrul e atom nenumeric, să returneze parametrul nemodificat.

```
(DEFUN triple (x)
  (COND
    ((numberp x) (* x 3))
    (t x)
  )
)
```

- Acum pentru apelul (MAPCAR #'triple '(1 a 2 b 3 c)) nu mai primim eroare, rezultatul va fi lista (3 A 6 B 9 C).
- Ce se întâmplă dacă lista este neliniară și conține subliste? De exemplu, dacă apelăm (MAPCAR #'triple '(1 a (2 b) 3 c))? Nu obținem eroare, pentru că funcția *triple* nu dă eroare dacă parametrul *x* este o listă, ci intră pe ramura *t* și returnează direct lista. În schimb, dacă lista conține numere, acele numere nu vor fi triplate, vor rămâne nemodificate. Deci rezultatul pentru apelul (MAPCAR #'triple '(1 a (2 b) 3 c)) va fi (3 A (2 B) 9 C).
- Ce să facem să putem tripla elementele și din subliste? Putem observa că funcția MAPCAR exact acest lucru face, aplică funcția *triple* pentru elementele listei, deci o putem modifica în așa fel încât dacă parametrul *x* e listă, să se apeleze, folosind MAPCAR, funcția *triple* pentru elementele listei.

```
(DEFUN triple (x)
  (COND
    ((numberp x) (* x 3))
    ((atom x) x)
    (t (mapcar #'triple x))
  )
)
```

- Iar acum nici nu mai avem nevoie să apelăm funcția cu (MAPCAR #'triple '(1 a (2 b) 3 c)), ci putem apela direct cu (triple '(1 a (2 b) 3 c)), parametrul fiind listă, execuția va intra direct pe ultima ramură, unde se va apela MAPCAR pentru elementele listei.
- Putem observa că funcția *triple* va tripla numerele dintr-o listă de pe orice nivel. Dacă lista conține subliste, prin MAPCAR funcția *triple* va fi apelată pentru elementele sublistei. Dacă careva dintre aceste elemente este listă, prin MAPCAR iar se apelează *triple* pentru toate elementele, și așa mai departe...
- Iar modelul matematic recursiv pentru funcția *triple* va arăta așa:

$$triple(x) = \begin{cases} 3 * x, \text{dacă } x \text{ e număr} \\ x, \text{dacă } x \text{ e atom} \\ \bigcup_{i=1}^n triple(x_i), \text{dacă } x \text{ e listă} \end{cases}$$

- Funcția MAPCAR returnează o listă. Dacă vrem ca rezultatul să fie un număr, putem folosi funcția *apply* pentru a aplica asupra listei rezultat o funcție care să calculeze rezultatul final. De exemplu dacă vrem să returnăm produsul elementelor dintr-o listă.

$$product(x) = \begin{cases} x, \text{dacă } x \text{ e număr} \\ 1, \text{dacă } x \text{ e atom} \\ \prod_{i=1}^n product(x_i), \text{dacă } x \text{ e listă} \end{cases}$$

```
(DEFUN product (x)
(COND
  ((numberp x) x)
  ((atom x) 1)
  (t (apply '* (mapcar #'product x))))
)
```

1. Să se returneze numărul nodurilor de pe niveluri pare într-un arbore n-ar, reprezentat în modul următor: (radacina (subarb_1) (subarb_2) ... (subarb_n)). Nivelul rădăcinii se consideră 1. De exemplu pentru arborele (A (B (D (G) (H)) (E (I)))) (C (F (J (L)) (K)))) rezultatul este 7.
 - Funcția MAPCAR ne poate ajuta să parcurgem toate nivelurile din arbore, dar ca să știm exact care nod trebuie numărat și care nu, vom avea nevoie de un parametru care să ne arate nivelul curent. Dacă nivelul curent este par și am găsit un nod (adică parametrul e atom, și nu listă) vom număra nodul respectiv. Dacă am găsit nod, dar nivelul nu e par, nu îl număram, iar dacă am găsit subarbore (adică sublistă), continuăm în subarbore, folosind MAPCAR, incrementând nivelul curent.

$$noduriPare(arb, nivel) = \begin{cases} 1, x \text{ e atom și nivel e par} \\ 0, x \text{ e atom} \\ \sum_{i=1}^n noduriPare(arb_i, nivel + 1) \end{cases}$$

- De data asta vom folosi cu MAPCAR o funcție cu 2 parametri. Când sunt mai mulți parametri, MAPCAR va aplica funcția respectivă pe perechi de elemente din parametri (primul element din prima listă cu primul element din a 2-a listă, al 2-lea element din prima listă cu al 2-lea element din a 2-a listă, etc.) până când se termină lista mai scurtă. Problema este că la noi parametrul 2, *nivel*, e un număr, nu e listă, și dacă încercăm ceva de genul (MAPCAR #'noduriPare x (+ 1 nivel)) vom avea o eroare, deoarece parametrul 2 va fi tratat ca listă, și MAPCAR va încerca să ia CAR din parametru. Soluția în asemenea situații este să folosim o expresie lambda. Expresiile lambda sunt niște funcții, în general simple, care nu au nume, și sunt definite direct acolo unde e nevoie de ele. O expresie lambda ne poate ajuta să "transformăm" funcția noastră care primește 2 parametri, într-o funcție care vrea un singur parametru.

```
(DEFUN noduriPare(arb nivel)
(COND
  ((and (atom arb) (= (mod nivel 2) 0)) 1)
  ((atom arb) 0)
  (t (apply '+ (mapcar #'(lambda (a) (noduriPare a (+ 1 nivel))) arb)))
)
)
```

- Ne mai trebuie o funcție care să apeleze noduriPare inițializând valoarea parametrului pentru nivel.

$$noduri(arb) = noduriPare(arb, 0)$$

```
(DEFUN noduri (arb) (noduriPare arb 0))
```

2. Se dă o listă neliniară. Calculați numărul sublistelor în care primul atom numeric este numărul 5. De exemplu, pentru lista (A 5 (B C D) 2 1 (G (5 H) 7 D) 11 14) sunt 3 asemenea liste: lista originală, (G (5 H) 7 D) și (5 H).
 - Pentru a rezolva problema vom avea nevoie de 2 funcții. Avem nevoie de o funcție care, primind o listă, verifică dacă lista respectă condiția, adică primul atom numeric din listă este sau nu numărul 5. Cealaltă funcție de care vom avea nevoie va număra câte subliste sunt care respectă condiția, folosind prima funcție pentru verificare.
 - Vom începe cu a 2-a funcție. Presupunem că funcția de verificare se numește *verif*, și returnează T, dacă în lista parametru primul număr este 5, sau nil dacă nu.

$$numara(l) = \begin{cases} 0, & \text{dacă } l \text{ e atom} \\ 1 + \sum_{i=1}^n numara(l_i), & \text{dacă } l \text{ e listă și } verific(l) \text{ e adevărat} \\ \sum_{i=1}^n numara(l_i), & \text{altfel} \end{cases}$$

```
(DEFUN numara (l)
(COND
  ((atom l) 0)
  ((verif l) (+ 1 (apply '+ (mapcar #'numara l))))
  (t (apply '+ (mapcar #'numara l))))
```

-)
-)
- Cum verificăm dacă primul atom numeric dintr-o listă este numărul 5 sau nu? Elementele listei pot fi de 3 tipuri (atom numeric, atom nenumeric, listă). Și trebuie să considerăm cazul când am ajuns la sfârșitul listei
 - Lista vidă – returnăm nil
 - Atom numeric – verificăm dacă este 5 sau nu
 - Atom nenumeric – continuăm verificarea în restul listei
 - Listă – aici e mai complicat. Dacă sublista conține măcar un atom numeric, atunci rezultatul va fi dat de acel atom numeric și nu contează ce este în restul listei. Dacă sublista nu conține atomi numerici, atunci rezultatul va fi dat de verificarea pentru restul listei.
- Deci pentru verificare va fi nevoie de 2 funcții: una care să verifice dacă o listă conține măcar un atom numeric și funcția care verifică dacă primul atom numeric e 5 sau nu.

$$contineNumar(l_1 \dots l_n) = \begin{cases} False, \text{dacă } n = 0 \\ True, \text{dacă } l_1 \text{ e număr} \\ contineNumar(l_2 \dots l_n), \text{dacă } l_1 \text{ e atom} \\ contineNumar(l_1) \text{ sau } contineNumar(l_2 \dots l_n), \text{ altfel} \end{cases}$$

```
(DEFUN contineNumar(l)
  (COND
    ((null l) nil)
    ((numberp (car l)) t)
    ((atom (car l)) (contineNumar (cdr l)))
    (t (or (contineNumar (car l)) (contineNumar (cdr l)))))
  )
)
```

$$verif(l_1 \dots l_n) = \begin{cases} False, \text{dacă } n = 0 \\ True, \text{dacă } l_1 = 5 \\ False, \text{dacă } l_1 \text{ e număr dar nu e 5} \\ verf(l_2 \dots l_n), \text{dacă } l_1 \text{ e atom} \\ verf(l_1), \text{dacă } contineNumar(l_1) \text{ e adevărat} \\ verf(l_2 \dots l_n), \text{ altfel} \end{cases}$$

```
(DEFUN verf(l)
  (COND
    ((null l) nil)
    ((numberp (car l)) (equal (car l) 5))
    ((atom (car l)) (verf (cdr l)))
    ((contineNumar (car l)) (verf (car l)))
    (t (verf (cdr l))))
  )
)
```

- Verificarea este complicată de faptul că lista nu este liniară. O altă variantă ar fi să transformăm lista într-una liniară, pentru că atunci verificarea devine mult mai simplă. Opțional, putem face această transformare păstrând doar atomi numerici din lista originală.

$$transform(l_1...l_n) = \begin{cases} \emptyset, & \text{dacă } n = 0 \\ l_1 \cup transform(l_2...l_n), & \text{dacă } l_1 \text{ e atom numeric} \\ transform(l_2...l_n), & \text{dacă } l_1 \text{ e atom} \\ transform(l_1) \cup transform(l_2...l_n), & \text{altfel} \end{cases}$$

```
(DEFUN transform(l)
(COND
  ((null l) nil)
  ((numberp (car l)) (cons (car l) (transform (cdr l))))
  ((atom (car l)) (transform (cdr l)))
  (t (append (transform (car l)) (transform (cdr l)))))
)
```

- Și ne mai rămâne de implementat funcția `verif` (a 2-a variantă)

$$verif(l_1...l_n) = \begin{cases} False, & \text{dacă } transform(l_1...l_n) = \emptyset \\ True, & \text{dacă primul element din } transform(l_1...l_n) \text{ este } 5 \\ False, & \text{altfel} \end{cases}$$

```
(DEFUN verific (l)
(COND
  ((null (transform l)) nil)
  ((equal (car (transform l)) 5) t)
  (t nil)
)
```

- O altă variantă este să facem funcția `verif` o funcție care să returneze 3 valori nu doar `t` și `nil`. Putem returna, de exemplu, 0 pentru lista vidă (adică pentru o listă care nu conține niciun atom numeric), 1 dacă primul atom numeric este numărul 5 și 2 dacă primul atom numeric nu este 5 (dar oricare 3 valori în loc de 0, 1 și 2 pot fi folosite).
- Și o altă variantă este să facem funcția `verif` să returneze primul atom numeric sau `nil` dacă lista nu conține niciun atom numeric.

$$verif(l_1l_2...l_n) = \begin{cases} \emptyset, & \text{dacă } n = 0 \\ l_1, & \text{dacă } l_1 \text{ este atom numeric} \\ verific(l_2...l_n), & \text{dacă } l_1 \text{ este atom nenumeric} \\ verific(l_2...l_n), & \text{dacă } verific(l_1) = \emptyset \\ verific(l_1), & \text{altfel} \end{cases}$$

```
(defun verific(l)
(cond
  ((null l) nil)
  ((numberp (car l)) (car l))
  ((atom (car l)) (verif (cdr l)))
  ((null (verif (car l))) (verif (cdr l)))
  (t (verif (car l))))
```

)
)

- Dacă definim `verif` așa, trebuie să modificăm definiția funcției `numara`. La implementarea funcției numără am presupus că `verif` returnează `t` sau `nil`. Acum că returnează altceva, trebuie să modificăm un pic implemetarea (sau să mai facem o funcție care apelează `verif` și returnează `t` sau `nil` în funcție dacă rezultatul lui `verif` este 5 sau nu).

$$numara(L) = \begin{cases} 0, & \text{dacă } L \text{ este atom} \\ 1 + \sum_{i=1}^n numara(L_i), & \text{dacă } verific(L) = 5 \\ \sum_{i=1}^n numara(L_i), & \text{altfel} \end{cases}$$

```
(defun numara (l)
  (cond
    ((atom l) 0)
    ((equal 5 (verif l)) (+ 1 (apply #' + (mapcar #'numara l))))
    (t (apply #' + (mapcar #'numara l))))
  )
)
```