Seminar 6 – Funcții MAP

- 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, pentru fiecare element din lista dată ca parametru, urmată de colectarea rezultatelor într-o listă. 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 nenumerici? 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 *triple,* a.î. 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 fiind 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 a doua clauză 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 pentru a tripla și elementele din subliste? Putem observa că funcția MAPCAR exact aceasta face, adică 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))
)
)
```

- Acum, nici nu mai este necesar 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, astfel, 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 se apelează *triple,* din nou, pentru toate elementele și așa mai departe.
- Modelul matematic recursiv pentru funcția *triple*:

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

- 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.

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

```
(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 pentru ca să știm care nod trebuie numărat și care nu, vom avea nevoie de un parametru care să ne indice nivelul curent. Dacă nivelul curent este par și am găsit un nod (adică parametrul e atom) vom număra nodul respectiv. Dacă am găsit un nod, dar nivelul nu e par, nu îl numărăm, iar dacă am găsit subarbore (adică sublistă), continuăm în subarbore, folosind MAPCAR, incrementând nivelul curent.

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

   - De data aceasta, 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 lista 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 o listă, și dacă încercăm apelul (MAPCAR #'noduriPare x (+ 1 nivel)) vom obține 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 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 necesită 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) si (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.

$$numara\ (l) = \begin{cases} 0, dacă\ l\ e\ atom \\ 1 + \sum_{i=1}^{n} numara(l_i), dacă\ l\ e\ listă\ și\ verif(l)\ e\ adevărat \\ \sum_{i=1}^{n} numara(l_1), altfel \end{cases}$$

```
(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 este mai complicat. Dacă sublista conține măcar un atom numeric, atunci rezultatul va fi dat de valoarea acelui 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 cel puțin un atom numeric și una care să verifice dacă primul atom numeric e 5 sau nu.

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

```
(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))))
)
)
```

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

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

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

```
(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 ramăne de implementat funcția verif (a 2-a variantă)

$$verif(l_1 \ldots l_n) = \begin{cases} False, dacă transform(l_1 \ldots l_n) = \emptyset \\ True, x1 = 5, unde\, x1 .. xm = transform\,(l_1 \ldots l_n) \\ False, altfel \end{cases}$$

```
(DEFUN verif (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.

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

```
(defun verif(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 *numara* am presupus că verif returnează t sau nil. Acum, că returnează altceva, trebuie să modificăm implemetarea (sau să mai facem o funcție care apelează verif și returnează t sau nil în funcție de rezultatul lui *verif*).

$$numara(L) = \begin{cases} 0, dacă\, L\ este\ atom \\ 1 + \sum_{i=1}^{n} numara(L_1), dacă\, verif(L) = 5 \\ \sum_{i=1}^{n} numara(L_i), altfel \end{cases}$$

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