

CURS 10

Expresii LAMBDA. Mecanisme definiționale evaluate. Funcții MAP

Cuprins

1. Definirea funcțiilor anonime. Expresii LAMBDA	1
1.1 Forma LABELS	2
1.2 Utilizarea expresiilor LAMBDA pentru evitarea apelurilor repetate	3
2. Mecanisme definiționale evaluate	5
Forme funcționale. Funcțiile APPLY și FUNCALL	8
3. Funcții MAP.....	11

1. Definirea funcțiilor anonime. Expresii LAMBDA

În situațiile în care

- funcția folosită o singură dată este mult prea simplă ca să merite a fi definită
- funcția de aplicat trebuie sintetizată dinamic (nu este, deci, posibil să fie definită static prin DEFUN)

se poate utiliza o formă funcțională particulară numită **expresie lambda**.

O expresie lambda este o listă de forma

(LAMBDA l f₁ f₂ ... f_m)

ce definește o funcție anonimă utilizabilă doar local, o funcție ce are definiția și apelul concentrate în același punct al programului ce le utilizează, l fiind lista parametrilor iar f₁...f_m reprezentând corpul funcției.

Argumentele unei expresii lambda sunt evaluate la apel. Dacă se dorește ca argumentele să nu fie evaluate la apel trebuie folosită forma **QLAMBDA**.

O astfel de formă LAMBDA se folosește în modul uzual:

((LAMBDA l f₁ f₂ ... f_m) par₁ par₂ ... par_n)

Exemple

Iată câteva exemple de utilizare a funcțiilor anonime

- **((lambda (l) (cons (car l) (cdr l))) '(1 2 3)) = (1 2 3)**
- **((lambda (l1 l2) (append l1 l2)) '(1 2) '(3 4)) = (1 2 3 4)**

- Să se definească o funcție care primește ca parametru o listă neliniară și returnează NIL dacă lista are cel puțin un atom numeric la nivel superficial și T, în caz contrar.

```
(defun f(l)
  (cond
    ((null l) t)
    (((lambda (v)
        (cond
          ((numberp v) t)
          (t nil)
        )
      )
     (car l)
     ) nil)
    (t (f (cdr l)))
  )
)
```

1.1 Forma LABELS

O formă specială pentru legarea locală a funcțiilor este forma LABELS.

Example

Ex1. evaluarea

```
(labels ((fct(l)
          (cdr l)
        )
        )
  (fct '(1 2))
)
```

va produce (2).

Ex2.

```
(labels ((temp (n)
          (cond
            ((= n 0) 0)
            (t (+ 2 (temp (- n 1)))))
        )
        )
  (temp 3)
)
```

va produce 6

Ex3. Să se scrie o funcție care primește ca parametru o listă de liste formate din atomi și întoarce T dacă toate listele conțin atomi numerici și NIL în caz contrar.

```
(test '((1 2) (3 4))) = T
(test '((1 2) (a 4))) = NIL
(test '((1 (2)) (a 4))) = NIL
```

Soluție

```
(DEFUN TEST (L)
  (COND
    ((NULL L) T)
    ((LABELS ((TEST1 (L)
                  (COND
                    ((NULL L) T)
                    ((NUMBERP (CAR L)) (TEST1 (CDR L)))
                    (T NIL)
                  )
                )
              )
      (TEST1 (CAR L))
    )
    (TEST (CDR L)))
  (T NIL)
)
```

1.2 Utilizarea expresiilor LAMBDA pentru evitarea apelurilor repetate

Ex1. Fie următoarea definiție de funcție

```
(defun g(l)
  (cond
    ((null l) nil)
    (t (cons (car (f l)) (cadr (f l))))
  )
)
```

Soluția pentru a evita apelul (**f l**) este folosirea unei funcții anonime utilizată local, care să poată fi apelată cu parametrul actual (**f l**).

Varianta 1

```
(defun g(l)
  (cond
    ((null l) nil)
    (t ((lambda (v)
```

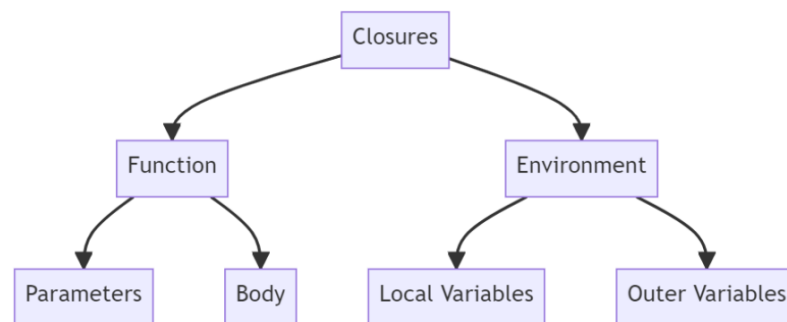

2. Mecanisme definiționale evaluate

Closure (închidere)

- *Lexical closure*
- <https://dept-info.labri.fr/~strandh/Teaching/MTP/Common/David-Lamkins/chapter15.html>
- [closure](#)

A closure is a function together with the environment in which it was created. The principal idea behind closures is that in Common Lisp, functions are “first-class objects,” that is, you can write code that takes a function as an argument, and you can write code that returns a function as one of its values.

- combinație între o funcție și mediul lexical în care aceasta e definită
 - o închidere dă acces la domeniul de vizibilitate al unei funcții exterioare dintr-o funcție interioară



- o **închidere** este o funcție care acces la variabilele care sunt în afara domeniului lor de vizibilitate lexical, chiar după ce aceasta și-a încheiat execuția
 - astfel funcția își “memorează” mediul în care a fost creată
- expresiile Lambda în Lisp reprezintă **închideri**
- conceptul provine din programarea funcțională, dar apare și în programarea imperativă
 - [C++ 11](#) - funcțiile lambda construiesc o închidere
 - Javascript, Python

Exemple Python

1)

```
def greet():
    # variable defined outside the inner function
    name = "John"

    # return a nested anonymous function
    return lambda: "Hi " + name

# call the outer function
message = greet()

# call the inner function
print(message())

# Output: Hi John
```

2)

```
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply

# Usage
double = multiplier(2)
triple = multiplier(3)

print(double(5)) # Output: 10
print(triple(5)) # Output: 15
```

Clojure

- <https://clojure.org/>
- dialect al limbajului Lisp pe platforme Java
- multi-paradigmă, orientat agent, concurent, funcțional, logic

Clozure Common Lisp (CCL)

- implementare Common Lisp
 - fire de execuție, compilare rapidă, mecanism pentru apelul funcțiilor externe (mecanism de *callback* – funcții Lisp apelate din cod extern)

Fie următoarele definiții și evaluări

```
> (defun f() 10)
F
> (setq f '11)
11
> (f)
10
> f
11
> (function f)
#<CLOSURE F NIL (DECLARE (SYSTEM::IN-DEFUN F)) (BLOCK F 10)>
> (setq g 7)
7
> (function g)
undefined function G
> (quote g) //echivalent cu 'g
G
> (function car)
#<SYSTEM-FUNCTION CAR>
> (function (lambda (l) (cdr l)))
#<CLOSURE :LAMBDA (L) (CDR L)>
```

De remarcat faptul că AND și OR nu sunt considerate funcții, ci operatori speciali.

> (function not)
 #<SYSTEM-FUNCTION NOT>
 > (function and)
 undefined function AND
 > (function or)
 undefined function OR

!!! Din punct de vedere semantic, standardul CommonLisp impune să se indice dacă e vorba de o funcție sau un simbol.

Argument		
Funcție f	# f	(function f)
Simbol x	' x	(quote x)

	Funcție f
Standard	#f
CLisp	'f
GCLisp, Emacs Lisp, alte dialecte	'f

	Expresie Lambda
Standard	#'(lambda)
CLisp	(lambda ...)
GCLisp, Emacs Lisp, alte dialecte	'(lambda)

Forma EVAL

Aplicarea formei EVAL este echivalentă cu apelul evaluatorului Lisp. Sintaxa funcției este

(EVAL **f**) : **e**

Efectul constă în evaluarea formei și returnarea rezultatului evaluării. Forma este evaluată de fapt în două etape: mai întâi este evaluată ca argument al lui EVAL iar apoi rezultatul acestei evaluări este din nou evaluat ca efect al aplicării funcției EVAL. De exemplu:

- (SETQ X '((CAR Y) (CDR Y) (CADR Y)))
- (SETQ Y '(A B C))
- (CAR X) se evaluează la (CAR Y)
- (EVAL (CAR X)) va produce A

Mecanismul este asemănător cu ceea ce înseamnă indirectarea prin intermediul pointerilor din cadrul limbajelor imperative.

- (SETQ L '(1 2 3))
- (SETQ P '(CAR L))
- P se evaluează la (CAR L)
- (EVAL P) va produce 1

- (SETQ B 'X)
- (SETQ A 'B)
- (EVAL A) se evaluează la X
- (SETQ L '(+ 1 2 3))
- L se evaluează la (+ 1 2 3)
- (EVAL L) se evaluează la 6

Observație. Lisp nu evaluează primul element dintr-o formă, ci numai îl aplică.

Ex: Asocierea (SETQ Q 'CAR) nu permite apelul sub forma (Q '(A B C)), un astfel de apel semnalând eroare în sensul că evaluatorul LISP nu găsește nici o funcție cu numele Q. Pe de altă parte, nici numai cu ajutorul funcției EVAL nu putem rezolva problema:

- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (EVAL P) va produce CAR
- ((EVAL P) '(A B C)) va produce mesaj de eroare: “Bad function when ...”

Mesajul de eroare de mai sus apare deoarece Lisp nu-și evaluează primul argument dintr-o formă.

!!! O listă este totdeauna evaluată dacă acest lucru nu este oprit explicit (prin QUOTE), în schimb primul argument al oricărei liste nu este niciodată evaluat!

Exemplu Fie lista L care pe prima poziție are un operator binar (+, -, *, /), iar pe următoarele 2 poziții are 2 operanzi (numerici). De exemplu, L = (* 2 3 - 4 1 ...). Se cere să returneze rezultatul aplicării operatorului (1-ul element al listei) asupra celor doi operanzi care urmează în listă. În exemplu nostru, ar trebui să se returneze valoarea 6.

Soluția este simplă: (EVAL (LIST (CAR L) (CADR L) (CADDR L))) = 6

Forme funcționale. Funcțiile APPLY și FUNCALL

Există situații în care forma funcției nu se cunoaște, expresia ei trebuind să fie determinată dinamic. Ar trebui să avem ceva de genul

(EXPR_FUNC p₁ ... p_n)

Deoarece EXPR_FUNC trebuie să genereze în cele din urmă o funcție ea este o așa-numită **formă funcțională**. Forma EXPR_FUNC este evaluată până ce se obține o funcție sau, în general, o expresie ce poate fi aplicată parametrilor.

Într-o astfel de situație, evaluarea parametrilor este amânată până în momentul reducerii formei funcționale EXPR_FUNC la funcția propriu-zisă F. Parametrii vor fi evaluați doar dacă F își evaluează, în prealabil, parametrii. Deci evaluarea formei de mai sus parcurge etapele:

- (i) reducerea formei `EXPR_FUNC` la `F` (eventual o expresie `LAMBDA` sau macrodefiniție) și substituția lui `EXPR_FUNC` prin `F` în forma de evaluat;
- (ii) evaluarea formei `(F p1 ... pn)`.

Există însă situații în care și numărul parametrilor trebuie stabilit dinamic, deci funcția determinată dinamic trebuie să accepte un număr variabil de parametri. Este nevoie deci de o modalitate de a permite aplicarea unei funcții asupra unei mulțimi de parametri sintetizată eventual dinamic. Acest lucru este oferit de funcțiile `APPLY` și `FUNCALL`.

(APPLY ff lp):e

- se evaluează argumentul și apoi se trece la evaluarea funcției
- Funcția `APPLY` permite aplicarea unei funcții asupra unor parametri furnizați sub formă de listă. În descrierea de mai sus, **ff** este o formă funcțională și **lp** este o formă reducibilă prin evaluare la o listă de parametri efectivi `(p1 p2 ... pn)`.

EXAMPLE

- `(APPLY #'CONS '(A B))` va produce `(A . B)`
- `(APPLY (FUNCTION CONS) '(A B))` va produce `(A . B)`
- `(APPLY #'MAX '(1 2 3))` va produce `3`
- `(APPLY #'+ '(1 2 3))` va produce `6`
- `(DEFUN F(L) (CDR L))`
- `(APPLY #'F '((1 2 3)))` va produce `(2 3)`
- `(APPLY #'(LAMBDA (L) (CAR L)) '((A B C)))` va produce `A`
- `(SETQ P 'CAR)`
- `(APPLY P '((A B C)))` va produce `A`
- `(APPLY #'P '((A B C)))` va produce eroare `undefined function P`
- `(SETQ Q 'CAR)`
- `(SETQ P 'Q)`
- `(APPLY (EVAL P) '((1 2 3)))` va produce `1`

(FUNCALL ff l1 l2 ...ln):e

- se evaluează argumentul și apoi se trece la evaluarea funcției
- `FUNCALL` este o variantă a funcției `APPLY` care permite aplicarea unei funcții (sau expresii) rezultate prin evaluarea unei forme funcționale **ff** asupra unui număr fix de parametri.

EXAMPLE

- `(FUNCALL #'CONS 'A 'B)` va produce `(A . B)`

- (FUNCALL (FUNCTION CONS) 'A 'B) va produce (A . B)
- (FUNCALL #'MAX '1 '2 '3)) va produce 3
- (FUNCALL #' + '1 '2 '3)) va produce 6
- (DEFUN F(L) (CDR L))
- (FUNCALL #'F '(1 2 3)) va produce (2 3)
- (FUNCALL #'(LAMBDA (L) (CAR L)) '(A B C)) va produce A
- (SETQ P 'CAR)
- (FUNCALL P '(A B C)) va produce A
- (FUNCALL # 'P '(A B C)) va produce eroare **undefined function P**
- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (FUNCALL (EVAL P) '(1 2 3)) va produce 1

!!! Atenție la folosirea AND și OR, deoarece nu sunt funcții.

- (APPLY #'AND '((T NIL))) va produce eroare **undefined function AND**
- (APPLY #'OR '((T NIL))) va produce eroare **undefined function OR**
- (FUNCALL #'AND '(T NIL)) va produce eroare **undefined function AND**
- (FUNCALL #'OR '(T NIL)) va produce eroare **undefined function OR**

Soluția este definirea unei funcții al cărei efect să fie aplicarea AND/OR pe elementele unei liste cu valori logice T, NIL.

$$SI(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } l_1 \text{ e fals} \\ SI(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun SI(l)
  (cond
    ((null l) t)
    ; (t (and (car l) (SI (cdr l))))
    ((not (car l)) nil)
    (t (SI (cdr l)))
  )
)
```

- (FUNCALL #'SI '(T NIL T)) va produce **NIL**
- (SI '(T T T)) va produce **T**

Exemple

Ex 1

- (DEFUN F()
 #'(LAMBDA (x) (CAR x))
)
- (FUNCALL (F) '(1 2 3)) → ???
- (APPLY (F) '((1 2 3))) → ???

Ex 2

- (FUNCALL (FUNCTION (LAMBDA (x) (cadr x))) '(1 2 3)) → ???

Exemplificare Closure Common Lisp

Ex1

```
(defun increment (x)
  #'(lambda (y)
      (+ x y)
    )
)

(setq inc5 (increment 5))
; returnează o nouă funcție (închidere) care adaugă 5 la argumentul său

(print (funcall inc5 3)) ; va afișa 8
```

Ex2

```
(defun two-funs (x)
  (list
    (function (lambda () x))
    (function (lambda (y) (setq x y)))
  )
)

(setq F (two-funs 6))

(funcall (car F)) => 6
(funcall (cadr F) 43) => 43
(funcall (car F)) => 43

(setq G (two-funs 5))
(funcall (car G)) => 5
(funcall (cadr G) 13) => 13
(funcall (car G)) => 13

(funcall (car F)) => 43
```

3. Funcții MAP

Rolul funcțiilor MAP este de a aplica o funcție în mod repetat asupra elementelor (sau sublistelor succesive) listelor date ca argumente.

(MAPCAR f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
 - CAR-ului listelor => e₁
 - CADDR-ului listelor => e₂
 -până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

Iată câteva exemple de aplicare ale funcției MAPCAR:

- (MAPCAR #'CAR '((A B C) (X Y Z))) se evaluează la (A X)
- (MAPCAR #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluează la (NIL T T)
- (MAPCAR #'LIST '(A B C)) se evaluează la ((A) (B) (C))
- (MAPCAR #'LIST '(A B C) '(1 2)) se evaluează la ((A 1) (B 2))
- (MAPCAR #'+ '(1 2 3) '(4 5 6)) se evaluează la (5 7 9)

Aplicarea funcției LIST este posibilă indiferent de numărul listelor argument deoarece LIST este o funcție cu număr variabil de argumente.

Observație. Dacă F este o funcție unară, care se aplică unei liste $L=(l_1 l_2 \dots l_n)$, atunci (MAPCAR #'F L) va produce lista $(F(l_1), F(l_2), \dots, F(l_n))$

Exemple

1. Să se definească o funcție MODIF care să modifice o listă dată ca parametru astfel: atomii nenumerici rămân nemodificați iar cei numerici își dublează valoarea; modificarea trebuie făcută la toate nivelurile.

(MODIF '(1 (b (4) c) (d (3 (5 f))))) va produce (2 (b (8) c) (d (6 (10 f))))

Model recursiv

$$\text{MODIF}(l) = \begin{cases} 2l & \text{dacă } l \text{ numar} \\ l & \text{dacă } l \text{ atom} \\ \bigcup_{i=1}^n \text{MODIF}(l_i) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

(DEFUN MODIF (L)

(COND

((NUMBERP L) (* 2 L)) ; determină operația asupra atomilor numerici

((ATOM L) L) ; determină operația asupra atomilor nenumeriți
 (T (MAPCAR #'MODIF L)) ; reprezintă strategia de parcurgere
)
)

2. Să se construiască o funcție LGM ce determină lungimea celei mai lungi subliste dintr-o listă dată L (dacă lista este formată numai din atomi atunci lungimea cerută este chiar cea a listei L).

(LGM '(1 (2 (3 4) (5 (6)) (7)))) va produce 4

Descrierea algoritmului se poate exprima astfel:

- valoarea LGM este maximul dintre lungimea listei L și maximul valorilor de aceeași natură calculate prin aplicarea lui LGM pentru fiecare element al listei L în parte;
- LGM(atom) = 0.

Model recursiv

$$LGM(l) = \begin{cases} 0 & \text{dacă } l \text{ e atom} \\ \max(n, \max(LGM(l_1), LGM(l_2), \dots, LGM(l_n))) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```
(DEFUN LGM(L)
  (COND
    ((ATOM L) 0)
    (T (MAX (LENGTH L) (APPLY #'MAX (MAPCAR #'LGM L)) ))
  )
)
```

Aplicarea funcțiilor ATOM și LENGTH calculează de fapt lungimile, (MAPCAR #'LGM L) realizând de fapt parcurgerea integrală a listei. Apelul (MAPCAR #'LGM L) furnizează o listă de lungimi. Deoarece trebuie să obținem maximul acestora, va trebui să aplicăm funcția MAX pe elementele acestei liste. Pentru aceasta folosim APPLY.

(MAPCAN f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară f este aplicată pe rând asupra:
 - CAR-ului listelor => e₁
 - CADR-ului listelor => e₂
 - CADDR-ului listelor => e₃
 -până când una din liste ajunge vidă
- Rezultatele sunt grupate cu NCONC într-o listă ce e returnată rezultat

(NCONC l1 l2 ... ln) : l

Relativ la modificarea sau nu a structurii listelor implicate, concatenarea de liste se poate efectua în două maniere: cu modificarea listelor (folosind funcția NCONC) și fără (folosind funcția APPEND)

- se evaluează argumentele și apoi se trece la evaluarea funcției
 - efect: **NCONC** realizează concatenarea efectivă (fizică) prin modificarea ultimului pointer (cu valoarea NIL) al primelor n-1 argumente și întoarce primul argument, care le va îngloba la ieșire pe toate celelalte.
-
- (SETQ L1 '(A B C) L2 '(D E)) se evaluează la (D E)
 - (APPEND L1 L2) se evaluează la (A B C D E)
 - L1 se evaluează la (A B C)
 - L2 se evaluează la (D E)
 - (SETQ L3 (NCONC L1 L2)) se evaluează la (A B C D E)
 - L1 se evaluează la (A B C D E)
-
- (SETQ L1 '(A) L2 '(B) L3 '(C)) se evaluează la (C)
 - L1 se evaluează la (A)
 - L2 se evaluează la (B)
 - L3 se evaluează la (C)
 - (NCONC L1 L2 L3) se evaluează la (A B C)
 - L1 se evaluează la (A B C)
 - L2 se evaluează la (B C)
 - L3 se evaluează la (C)

Exemple MAPCAN

- (MAPCAN #'CAR '((A B C) (X Y Z))) se evaluează la NIL, deoarece NCONC cere liste, și ca atare (NCONC 'A 'X) este NIL
- (MAPCAN #'LIST '(A B C) '(1 2)) se evaluează la (A 1 B 2)
- (MAPCAN #'LIST '(A B C)) se evaluează la (A B C)
- (MAPCAN #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluează la NIL
- (MAPCAN #'+ '(1 2 3) '(4 5 6)) se evaluează la NIL

Observație (MAPCAN vs. MAPCAR)

Fie următoarele definiții

```
(defun F(L)
  (cdr L)
)
```

(setq L '((1 2 3) (4 5 6) (7 8))) → ((1 2 3) (4 5 6) (7 8))

(mapcar #'F L) → ((2 3) (5 6) (8 9))

(mapcan #'F L) → (2 3 5 6 8 9)

⇒ (apply #'append (mapcar #'F L)) ≡ (mapcan #'F L)

(MAPLIST f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
 - listelor => e₁
 - CDR-ului listelor => e₂
 - CDDR-ului listelor => e₃
 -până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

Exemple MAPLIST

- (MAPLIST #'APPEND '(A B C) '(1 2 3)) furnizează ((A B C 1 2 3) (B C 2 3) (C 3))
- (MAPLIST #'(LAMBDA (X) X) '(A B C)) furnizează ((A B C) (B C) (C))
- (SETF TEMP '(1 2 7 4 6 5)) urmat de
(MAPLIST #'(LAMBDA (XL YL) (< (CAR XL)(CAR YL)))
TEMP (CDR TEMP)
) va furniza lista (T T NIL T NIL)

Comparativ cu exemplele date la MAPCAR, aici vom obține:

- (MAPLIST #'CAR '((A B C) (X Y Z))) se evaluează la ((A B C) (X Y Z))
- (MAPLIST #'LIST '(A B C) '(1 2)) se evaluează la (((A B C) (1 2)) ((B C) (2)))
- (MAPLIST #'LIST '(A B C)) se evaluează la (((A B C)) ((B C)) ((C)))
- (MAPLIST #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluează la (NIL NIL NIL)
- (MAPLIST #'+'(1 2 3) '(4 5 6)) va produce mesajul de eroare: “argument to + should be a number: (1 2 3)”.

(MAPCON f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
 - listelor => e₁
 - CDR-ului listelor => e₂
 - CDDR-ului listelor => e₃
 -până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **NCONC** într-o listă ce e returnată rezultat

Exemple MAPCON

- (MAPCON #'CAR '((A B C) (X Y Z))) furnizează (A B C X Y Z)

- (MAPCON #'LIST '(A B C) '(1 2)) furnizează ((A B C) (1 2) (B C) (2))
- (MAPCON #'LIST '(A B C)) furnizează ((A B C) (B C) (C))
- (MAPCON #'EQUAL '(A (B C) D) '(Q (B C) D X)) furnizează NIL
- (MAPCON #'+ '(1 2 3) '(4 5 6)) furnizează mesaj de eroare: : “[argument to + should be a number: \(1 2 3\)](#)”
- (DEFUN G(L)
 (MAPCON #'LIST L)
)

 (G '(1 2 3)) = ((1 2 3) (2 3) (3))
- (MAPCON #'(LAMBDA (L) (MAPCON #'LIST L)) '(1 2 3)) furnizează ((1 2 3) (2 3) (3) (2 3) (3) (3))