# Programare funcţională – Laboratorul 13

Aplicaţii recapitulative

Isabela Drămnesc

May 22, 2012

## 1 Aplicaţii:

### 1.1 Matrici

```
> (setq m (make-array '(4 2 3)))
#3A(((NIL NIL NIL) (NIL NIL NIL))
    ((NIL NIL NIL) (NIL NIL NIL))
    ((NIL NIL NIL) (NIL NIL NIL))
    ((NIL NIL NIL) (NIL NIL NIL)))

> (setq m (make-array '(4 2 3) :initial-element 0))

> (type-of m)

> (setq m (make-array '(4 2 3) :initial-contents '(((1 2 -8) (2 -2 2))
    ((4 3 1) (1 0 -5)) ((3 3 3) (8 8 8)) ((5 6 7) (10 20 30)))))

; #3A - avem o entitate Lisp de tip matrice cu trei dimensiuni
; deci crearea unei matrici se poate face si direct folosind #na,
; cu n- dimensiunea matricei

> (setq m #3A(((1 2 -8) (2 -2 2)) ((4 3 1) (1 0 -5))((3 3 3)
(8 8 8))))

> (setq m #3A(((1 2 -8) (2 -2 2)) ((4 3 1) (1 0 -5))((3 3 3)
(8 8 8)) ((5 6 7) (10 20 30))))

> (setq m #2a(((2 2 2) (1 1 1)) ((3 3 3) (4 4 4))))

> (type-of m)

> (setq m #2a((2 2 2) (1 1 1) (3 3 3) (4 4 4)))

> (type-of m)
```

### 1.2 Vectori

```
> (setq v (vector 1 2 3 9 8 7 -4 0 19 28))

;sau direct

> (setq v1 #(1 2 3 9 8 7 -4 0 19 28))

> (type-of v)

> (type-of v1)
```

## 1.3 Funcţii pe matrici

```
; -numerotarea incepe de la 0

; aref -referirea unei valori intr-o celula a matricei
; (aref (matrice indici-doriti))

> (setq m (make-array '(4 2 3) :initial-contents '(((1 2 -8) (2 -2 2))
    ((4 3 1) (1 0 -5)) ((3 3 3) (8 8 8)) ((5 6 7) (10 20 30)))))

> (aref m 0 0 0)

> (aref m 0 0 2)

> (aref m 0 1 2)

> (aref m 1 0 2)

> (aref m 1 0 0)

> (aref m 1 1 1)

> (aref m 3 1 1)

> (setf (aref m 3 1 1) -188888888)

> (aref m 3 1 1)

> m
```

## 1.4 Operaţii cu vectori şi proprietăţi

### 1.4.1 Sortare

```
> (defun bubble_sort (v &aux schimbat (n (length v)))
    (loop
       (setq schimbat nil)
       (dotimes (i (1- n))
           (when (> (aref v i) (aref v (1+ i)))
              (psetf (aref v i) (aref v (1+ i))
                     (aref v (1+ i)) (aref v i)
```

```
                    schimbat t)
            )
        )
    (unless schimbat (return v))
    )
  )
```

```
> (setq v (vector 9 0 -7 4 3 -2 3 1 8))

> (bubble_sort v)

> (length v)

> (concatenate 'vector v #(3 3 3 3))

> (setq v2 #(34 67 2 0 1 19 11 -67 -9 18))

> (sort v2 #'<)
```

### 1.4.2 Proprietăţi

```
; daca mai adaugam un argument optional fill-pointer la make-array
; o sa obtinem un vector care poate fi expandat. Primul argument din
; make-array specifica spatiul care trebuie alocat pentru vector, iar
; fill-pointer daca exista ca si parametru specifica lungimea initiala.

> (setf vec (make-array 10 :fill-pointer 3 :initial-element nil))

> (length vec)

> (vector-push 'a vec)

> vec

> (vector-pop vec)

> vec

> (vector-push 'a vec)

> vec

> (vector-push 'a vec)

> vec

; putem adauga elemente in vector cu vector-push atata timp cat
; fill-pointer e < decat dimensiunea
; data ca prim argument in make-array.
```

```
; Exemplu:

> (setf vec (make-array 5 :fill-pointer 3 :initial-element nil))

> (vector-push 'i vec)

> vec

> (vector-push 'i vec)

> vec

> (vector-push 'i vec)

> vec

;; alt exemplu

> (setf vect2 (make-array 5 :fill-pointer 2
 :initial-contents '((1 1 1) (2 2 2) () () ())))

> (vector-push '(3 3 3) vect2)

> vect2

> (vector-push 'isabela vect2)

> vect2

> (vector-push '18 vect2)

> vect2

> (vector-pop vect2)

> vect2

> (vector-pop vect2)

> vect2

> (vector-pop vect2)

> vect2

;; De exemplu in construirea unui dictionar sunt utile
; proprietatile de mai sus.
```

### 1.4.3 Adunarea a doi vectori (iterativ)

```
(defun adun-vect (a b)
  (loop for i from 0 to (1- (min (length a) (length b)))
        collect (+ (nth i a) (nth i b))))

(defun printare-suma-vectori ()
     (format t "~%~%Exemple adun-vect:~%~%")

     (setf a '(1 2 3)
           b '(1 2))
     (format t "(adun-vect ~a ~a) este ~a~%" a b (adun-vect a b))
     (setf a '(1 2 3)
           b '(-1 -2 -3))
     (format t "(adun-vect ~a ~a) este ~a~%" a b (adun-vect a b))
)
```

### 1.4.4 Produsul scalar a doi vectori dați ca liste

```
; Varianta 1

(labels ((dot-product (a b)
            (if (or (null a) (null b))
                 0
                 (+ (* (car a) (car b))
                    (dot-product (cdr a) (cdr b))
                 )
            )
          )
         )
      (dot-product '(1 2 3) '(10 20 30))
)

; Varianta 2

(apply #'+ (mapcar #'* '(1 2 3) '(10 20 30)))
```

### 1.4.5 Vectori rari

Un vector rar va fi reprezentat printr-o listă de subliste, fiecare sublistă (componentă) fiind formată din două elemente: un index şi valoarea corespunzatoare:

((< index1 >,< val1 >), . . . , (< indexn >,< valn >)).

De exemplu, vectorul

#(1.0, 0, 0, 0, 0, 0, −2.0)

va fi reprezentat prin

((1 1.0) (7 −2.0))

Pentru accesul la indexul si valoare am folosit doua functii index si val.

```
(defun comp (vector)
    (car vector)) ; extrage componenta

(defun rest-comp (vector)
      (cdr vector)) ; rest componente

(defun index (comp)
        (car comp)) ; extrage indexul din perechea
        ; (<index>, <valoare>)

(defun val (comp)
        (cadr comp)) ; extrage valoarea din perechea
        ; (<index>, <valoare>)

;suma a doi vectori rari U si V

(defun suma-vect (U V)
   (cond ((endp U) V)
         ((endp V) U)
         ((< (index (comp U)) (index (comp V)))
         (cons (comp U) (suma-vect (rest-comp U) V)))
         ((> (index (comp U)) (index (comp V)))
         (cons (comp V) (suma-vect U (rest-comp V))))
         (t (cons (list (index (comp U))
       (+ (val (comp U)) (val (comp V))))
     (suma-vect (rest-comp U) (rest-comp V))))))

> (suma-vect '((1 1.0) (7 -2.0)) '((2 2.3) (7 2.1)))


; prod-vect-const ce calculeaza
; produsul dintre un vector rar si un scalar.

(defun prod-vect-const (V s)
        (cond ((zerop s) nil) ; caz special
        ((endp V) nil) ; cond. terminare
        (t (cons (list (index (comp V))
        (* s (val (comp V))))
    (prod-vect-const (rest-comp V) s)))))

> (prod-vect-const '((1 23) (4 7) (5 20)) 8)
```

#### 1.4.6 Să se scrie o funcţie prod-vect-scalar ce calculează produsul scalar a doi vectori rari U si V.

## 1.5 Operaţii cu matrici

### 1.5.1 Transpusa unei matrici (iterativ)

```
(defun transpusa (matrice)
```

```
  (let ((n (array-dimension matrice 0))    ;; n este numarul de linii
        (m (array-dimension matrice 1)))   ;; m este numarul de coloane
    (let ((transpusa (make-array (list m n))))
                        ;; transpusa este o matrice noua
                        ;in care se va calcula rezultatul
      (loop for i from 0 to (1- n) do
            (loop for j from 0 to (1- m) do
                  (setf (aref transpusa j i)
                        (aref matrice i j)))
            finally (return transpusa)))))

(defun printare-transpusa ()
    (format t "~%~%Exemple transpusa:~%~%")

   (setf matrice #2a((1 2)
                     (3 4)))
   (format t "Transpusa matricii ~a este matricea ~a~%"
                           matrice (transpusa matrice))
   (setf matrice #2a((1 2 3)
                     (4 5 6)
                     (7 8 9)))
   (format t "Transpusa matricii ~a este matricea ~a~%"
                           matrice (transpusa matrice))
   (setf matrice #2a((1 0 0)
                     (0 1 0)
                     (0 0 1)))
   (format t "Transpusa matricii ~a este matricea ~a~%"
                           matrice (transpusa matrice))
)
```

### 1.5.2 Scrieţi functii (recursive) pentru adunarea a două matrici şi pentru înmulţirea unei matrici cu o constantă.

### 1.5.3 Matrici rare

O matrice rară poate fi reprezentată ca o lista de subliste în care fiecare sublistă are forma:

(< nr − linie > ((< index1 > ,< val1 >), . . . , (< indexn >,< valn >)))
(de exemplu: ((1 ((1 1.0) (5 −2.0))) (3 ((3 2.0) (11 −1.0))))).

Pentru a avea acces la elementele matricei rare vom folosi urmatoarele functii:

```
(defun linie (matrice)
       (car matrice)) ; extrage linie

(defun rest-linii (matrice)
       (cdr matrice)) ; rest linii

(defun nr-lin (linie)
       (car linie)) ; extrage index linie
```

```
(defun comp-lin (linie)
        (cadr linie)) ; extrage lista componente linie


; suma a doua matrici rare A si B

(defun suma-m (A B)
    (cond ((endp A) B)
          ((endp B) A)
          ((< (nr-lin (linie A)) (nr-lin (linie B)))
             (cons (linie A) (suma-m (rest-linii A) B)))
          ((> (nr-lin (linie A)) (nr-lin (linie B)))
             (cons (linie B) (suma-m A (rest-linii B))))
          (t (cons (list (nr-lin (linie A))
       (suma-vect (comp-lin (linie A)) (comp-lin (linie B))))
       (suma-m (rest-linii A) (rest-linii B))))))
```

#### 1.5.4 Probleme:

1. Modificaţi programele de mai sus astfel încât să fie eliminate componentele cu valori zero din rezultat. Adăugaţi funcţii ce permit citirea, respectiv afişarea unor tablouri rare.

2. Extindeţi mulţimea de funcţii de mai sus pentru a opera cu tablouri cu mai multe dimensiuni.

### 1.6 Inferente

```
; inferente bazate pe reguli if-then
; Lisp e un limbaj pentru "symbolic computation".

; reprezentare ca Donald e parintele lui Nancy

(parent donald nancy)

; regula:
(<- head body)

(<- (child ?x ?y) (parent ?y ?x))     se citeste
       Daca y e parintele lui x, atunci x e copilul lui y
       Putem demonstra orice fapt de forma (child x y) daca
    demonstram (parent y x)

(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
     ; x e tatal lui y daca x e parintele lui y si x e barbat

(<- (daughter ?x ?y) (and (child ?x ?y) (female ?x)))
     ; x e fiica lui y daca x e copilul lui y si x e femeie
     ; aceasta regula foloseste regula de mai sus (child ?x ?y)
```

```
; backward chaining = procesul de demonstrare continua aplicand reguli
;                     pana cand se ajunge la un fapt deja cunoscut.
; backward = acest mod de inferenta prima data considera partea lui then
;            pentru a vedea daca regula e utila inainte de a continua
;            demonstratia pentru partea if.
; chaining = regulile pot depinde de alte reguli, formand un "chain"
;            (defapt e mai mult un arbore de la ceea ce vrem sa demonstram
;            la ceea ce deja stim).


;; Matching

; avem nevoie de o functie care face matching
; compara 2 liste si decide ce posibilitati de matching avem

;Exemplu:

(p ?x ?y c ?x)
(p  a  b c  a)

; se poate face matching atunci cand ?x=a ?y=b

(p ?x b ?y a)
(p ?y b  c a)

; fac matching atunci cand ?x=?y=a

; scriem o functie care primeste ca parametri 2 arbori,
; daca se poate face matching, atunci se va returna o lista de asociere
; reprezentand variabilele si valorile lor in urma matching-ului.

(defun match (x y &optional binds)
   (cond
      ((eql x y) (values binds t))
      ((assoc x binds) (match (binding x binds) y binds))
      ((assoc y binds) (match x (binding y binds) binds))
      ((var? x) (values (cons (cons x y) binds) t))
      ((var? y) (values (cons (cons y x) binds) t))
      (t
        (when (and (consp x) (consp y))
           (multiple-value-bind (b2 yes)
                                (match (car x) (car y) binds)
             (and yes (match (cdr x) (cdr y) b2)))))))

(defun var? (x)
   (and (symbolp x)
        (eql (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
```

```
    (let ((b (assoc x binds)))
         (if b
             (or (binding (cdr b) binds)
                 (cdr b)))))


> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A)) ;
T

> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y)) ;
T

> (match '(a b c) '(a a a))
NIL


; match compara element cu element, construieste asignari de
; valori pentru variabile, numite "bindings" in
; parametrul "binds"

; daca s-a putut face matching, atunci se returneaza
; "bindings" generate, altfel nil.

;Explicati:

> (match '(p ?x) '(p ?x))
NIL ;
T

> (match '(p ?v b ?x d (?z ?z))
                '(p a ?w c ?y (e e))
                '((?v . a) (?w . b)))
((?Z . E) (?Y . D) (?X . C) (?V . A) (?W . B)) ;
T

;pentru ca facand matching x cu y si apoi y cu a,
; stabilim indirect ca x trebuie sa fie a.
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y)) ;
T
```

### 1.6.1 Definiţi cele 3 funcţii folosind macro-uri. Testaţi şi analizaţi.

### 1.6.2 Definiţi un macro care sa permită definirea de reguli.

De exemplu dacă am un fapt:

```
(parent donald nancy)
```

şi întrebăm programul

```
(parent ?x ?y)
```

să returneze

```
(((?x . donald) (?y . nancy)))
```