Functional Programming – Laboratory 13

Revision

Isabela Drămnesc

May 23, 2012

1 Exercises

1.1 Matrices

```
> (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 - denotes a 3-dimensional matrix
; so we can create a matrix directly by using #na,
; where n- is the matrix dimension

> (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 Vectors

```
> (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 Functions on matrices

```
; -numbering starts from 0

; aref -reference value in a cell of the matrix
; (aref (matrix desired-indices))

> (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 Operations with vectors and properties

1.4.1 Sorting

```
> (defun bubble_sort (v &aux changed (n (length v)))
     (loop
        (setq changed 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))
```

```
                        changed t)
              )
          )
      (unless changed (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 Properties

```
; if we add an optional argument fill-pointer to make-array
; then we will get a vector which can be expanded. The first argument of
; make-array specifies the space to be allocated for the vector
; and fill-pointer as parameter, if exists, specifies the initial length.

> (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

; we can add elements to the vector by using vector-push until
; fill-pointer is < than the dimension given as first argument
; in make-array.
```

```
; Example:

> (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

;; other example

> (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

;; For example, in order to build a dictionary
; the properties introduced above are useful.
```

1.4.3 Add two vectors (iterative)

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

(defun print-sum-vectors ()
     (format t "~%~%Example add-vect:~%~%")

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

1.4.4 The scalar product of two vectors given as lists

```
; Version 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))
)

; Version 2

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

1.4.5 Rare vectors

A rare vector will be represented as a list of sublists, each sublist has 2 elements: an index and the corresponding value:

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

For example, the vector

```
#(1.0, 0, 0, 0, 0, 0, -2.0)
```

is represented as

```
((1 1.0) (7 -2.0))
```

In order to access the index and the value we use two functions index and val.

```
(defun comp (vector)
    (car vector)) ; extracts the component

(defun rest-comp (vector)
      (cdr vector)) ; the rest of components

(defun index (comp)
        (car comp)) ; extracts the index from the pair
                    ; (<index>, <value>)

(defun val (comp)
        (cadr comp)) ; extracts the value from the pair
                     ; (<index>, <value>)

;the sum of two rare vectors U and V

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

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


; prod-vect-const for calculating
; the product of a rare vector and a scalar.

(defun prod-vect-const (V s)
        (cond ((zerop s) nil) ; a special case
        ((endp V) nil) ; end condition
        (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 Write a function prod-vect-scalar for calculating the scalar product of two rare vectors U and V.

1.5 Operations on matrices

1.5.1 The transpose of a matrix (iterative)

```
(defun transpose-m (matrix)
```

```
  (let ((n (array-dimension matrix 0))    ;; n is the number of lines
        (m (array-dimension matrix 1)))   ;; m is the number of columns
    (let ((transpose-m (make-array (list m n))))
                         ;; transpose-m is a new matrix
                         ;which will contain the result
      (loop for i from 0 to (1- n) do
            (loop for j from 0 to (1- m) do
                  (setf (aref transpose-m j i)
                        (aref matrix i j)))
            finally (return transpose-m)))))

(defun print-transpose-m ()
    (format t "~%~%Example transpose:~%~%")

   (setf matrix #2a((1 2)
                    (3 4)))
   (format t "The transpose of the matrix ~a is the matrix ~a~%"
                          matrix (transpose-m matrix))
   (setf matrix #2a((1 2 3)
                    (4 5 6)
                    (7 8 9)))
   (format t "The transpose of the matrix ~a is the matrix ~a~%"
                          matrix (transpose-m matrix))
   (setf matrix #2a((1 0 0)
                    (0 1 0)
                    (0 0 1)))
   (format t "The transpose of the matrix ~a is the matrix ~a~%"
                          matrix (transpose-m matrix))
)
```

1.5.2 Write (recursive) functions for the sum of two matrices and for multiply a matrix with a constant.

1.5.3 Rare matrices

A rare matrix can be represented as a list of sublists, where each sublist is of the form:

```
(< nr - line > ((< index1 >,< val1 >), . . . , (< indexn >,< valn >)))
(example: ((1 ((1 1.0) (5 -2.0))) (3 ((3 2.0) (11 -1.0))))).
```

In order to access the elements of the rare matrix we use the following functions:

```
(defun lines (matrix)
        (car matrix)) ; extracts a line

(defun rest-lines (matrix)
        (cdr matrix)) ; rest of the lines

(defun nr-lines (line)
        (car line)) ; extracts the index line
```

```
(defun comp-lines (line)
        (cadr line))


; the sum of two rare matrices A and B

(defun sum-m (A B)
    (cond ((endp A) B)
          ((endp B) A)
          ((< (nr-lines (line A)) (nr-lines (line B)))
             (cons (line A) (sum-m (rest-line A) B)))
          ((> (nr-lines (line A)) (nr-lines (line B)))
             (cons (line B) (sum-m A (rest-lines B))))
          (t (cons (list (nr-lines (line A))
       (sum-vect (comp-lines (line A)) (comp-lines (line B))))
       (sum-m (rest-lines A) (rest-lines B))))))
```

1.5.4 Problems:

1. Modify the programs above such that the components with the values zero are eliminated. Add functions which allow reading, respectively printing a rare array.

2. Extend the set of the functions above in order to operate on n-dimensional arrays.

1.6 Inferences

```
; inferences based on if-then rules
; Lisp is a language for "symbolic computation".

; the representation that Donald is the father of Nancy

(parent donald nancy)

; rule:
(<- head body)

(<- (child ?x ?y) (parent ?y ?x))      is read as:
       If y is the parent of x, then x is the child of y
       We can prove any fact of the form (child x y) if we
    prove (parent y x)

(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
     ; x is the father of y if x is the parent of y and x is a man

(<- (daughter ?x ?y) (and (child ?x ?y) (female ?x)))
     ; x is daughter of y if x is the child of y and x is female
     ; this rule uses the rule above (child ?x ?y)
```

```lisp
; backward chaining = is the process of continue proving by applying some rules
;                     a fact already known is reached.
; backward = this way of inference first considers the then part
;            in order to see if the rule is useful before going on
;            the proof for the if part.
; chaining = the rules can depend on other rules, by forming a "chain"
;            (actually is more than a tree from what we want to prove
;            to what we already know).


;; Matching

; we need a function for matching
; compare 2 lists and then decide which possibility of
; matching do we have

;Example:

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

; matching can be done when ?x=a ?y=b

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

; matching is done when ?x=?y=a

; we write a function with two tree parameters,
; if matching can be done, then returns an association list
; representing the variables and their corresponding values
; after matching is done.

(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 compares element by element, constructs assignment
; values for variables, called "bindings" in the
; parameter "binds"

; if matching is done, then returns
; "bindings" generated, otherwise nil.

;Explain:

> (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

;because matching x with y and then y with a,
; we establish indirectly that x has to be a.
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y)) ;
T
```

1.6.1 Define the 3 functions above by using macros. Test and analyze.

1.6.2 Define a macro which allows us to define new rules.

For example if we have a fact:

```
(parent donald nancy)
```

and we ask the program

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

the result will be

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