1. ΕΙΣΑΓΩΓΗ

Η LISP είναι μια από τις παλαιότερες γλώσσες προγραμματισμού. Δημιουργήθηκε στο τέλος της δεκαετίας του ’50 από τον John McCarthy, έναν από τους πρωτεργάτες του επιστημονικού πεδίου της Τεχνητής Νοημοσύνης. Ανήκει στην κατηγορία των λεγόμενων συναρτησιακών γλωσσών (functional languages), στηρίζεται δηλ. στη μαθηματική θεωρία των αναδρομικών συναρτήσεων.

Βασικη δομή δεδομένων στη LISP είναι η λίστα, απ’ όπου πήρε και το όνομά της (LISt Processing). Δηλ. οι επεξεργασίες σ’ ένα πρόγραμμα LISP είναι κατά βαση επεξεργασίες λιστών. Δεδομένου δε ότι μια λίστα μπορεί να περιέχει σύμβολα ή συμβολικές δομές, η LISP είναι κατάλληλη για επεξεργασία συμβόλων και συμβολικών δομών, δηλ. γι’ αυτό που ονομάζουμε *συμβολικό υπολογισμό* (symbolic computation). Ενώ ο αριθμητικός υπολογισμός (arithmetic computation) βασίζεται σε αριθμητικές πράξεις, ο συμβολικός υπολογισμός αναφέρεται στη δημιουργία και διαχείριση συμβολικών δομών. Αυτός είναι ο λόγος που η LISP έγινε (και είναι) η δημοφιλέστερη γλώσσα στην επιστημονική κοινότητα της Τεχνητής Νοημοσύνης (ΤΝ). Οι περισσότερες εφαρμογές ΤΝ είναι γραμμένες (τουλάχιστον το πρωτότυπό τους) σε LISP. Η LISP είναι από τα βασικά εφόδια γι’ αυτό που ονομάζεται *ευφυής προγραμματισμός* (AI programming).

Επειδή η LISP είναι γλώσσα σχετικά χαμηλότερου επιπέδου από άλλες γλώσσες, έχει χρησιμοποιηθεί ως γλώσσα ανάπτυξης εργαλείων ΤΝ υψηλότερου επιπέδου, όπως είναι τα κελύφη έμπειρων συστημάτων τα βασισμένα σε κανόνες, οι αποδείκτες θεωρημάτων κ.ά.

2. ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ-ΟΡΙΣΜΟΙ

2.1 Πρόγραμμα LISP-Τύποι Δεδομένων

Όπως είπαμε, η LISP είναι μια *συναρτησιακή γλώσσα* και το είδος του προγραμματισμού που υπηρετεί ονομάζεται *συναρτησιακός προγραμματισμός* (functional programming). Αυτό σημαίνει ότι βασικό στοιχείο του ενός προγράμματος LISP είναι η *συνάρτηση*. Έπίσης, ένα άλλο χαρακτηριστικό αυτού του είδους προγραμματισμού είναι η *έλλειψη πλευρικών επιπτώσεων* (side effects), με κάποιες εξαιρέσεις βέβαια. Αυτό σημαίνει ότι κάθε συνάρτηση έχει συνήθως ένα αποτέλεσμα, το εξαγόμενό της, και όχι και άλλα δευτερεύοντα. Τέλος, ένα άλλο χαρακτηριστικό είναι η *αναδρομική φύση* του. Η αναδρομή είναι φυσικό στοιχείο ενός προγράμματος LISP.

Ένα *πρόγραμμα LISP* δεν είναι τίποτε άλλο από ένα σύνολο συναρτήσεων, όπου η μια συνάρτηση (μπορεί να) καλεί μια ή περισσότερες συναρτήσεις και τον εαυτό της.

Η αναπαράσταση υπολογιστικών οντοτήτων στη LISP γίνεται με τις *συμβολικές εκφράσεις* ή *σ-εκφράσεις* (s-expressions). Αυτές αποτελούν τα δομικά στοιχεία της γλώσσας, δηλ. τα στοιχεία απο τα οποία συντίθενται οι προτάσεις της LISP. Οι συμβολικές εκφράσεις αναπαριστούν και δεδομένα, δηλ. μη εκτιμήσιμα στοιχεία, και τμήματα προγράμματος, δηλ. εκτιμήσιμα στοιχεία (ή με άλλα λόγια στοιχεία για τα οποία χρειάζεται να γίνει κάποιο είδος υπολογισμού). Πιο συγκεκριμένα, μια συμβολική έκφραση που χρειάζεται εκτίμηση (δηλ. υπολογισμό του αποτελέσματός της) ονομάζεται *συναρτησιακός τύπος* (functional form) ή απλώς *τύπος* (form).

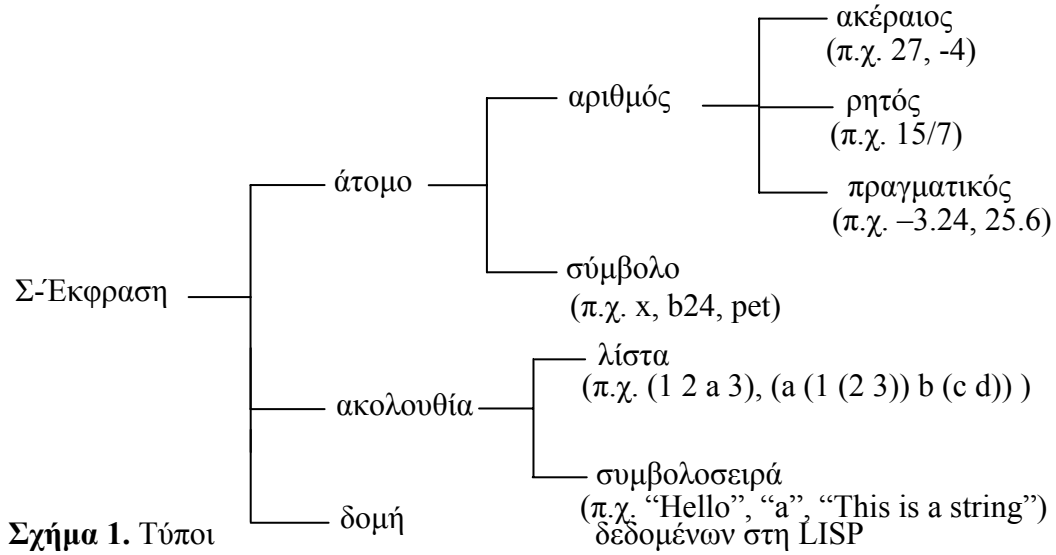
Στο σχήμα 2.1 απεικονίζονται οι τύποι δεδομένων της LISP. Υπάρχουν τέσσερις βασικοί τύποι, τα *άτομα*, οι *λίστες*, οι *συμβολοσειρές* και οι *δομές*. Τα άτομα διακρίνονται σε *αριθμούς* και *σύμβολα*. Οι αριθμοί διακρίνονται σε *ακεραίους*, *ρητούς*

και πραγματικούς. Η LISP δεν απαιτεί δηλώσεις τύπων για τις μεταβλητές. Αυτό ελευθερώνει τον προγραμματιστή από τέτοιες λεπτομέρειες, αλλά δεν βοηθά στον έλεγχο του προγράμματος.

Το μόνο που διαθέτει είναι τα *ειδικά σύμβολα* T (true) και NIL. Το NIL είναι ταυτόσημο με την κενή λιστα '()' και αντιπροσωπεύει το ψεύδος μιας πρότασης, ενώ το T την αλήθεια.

Μια λίστα είναι της μορφής (e_1 e_2 … e_n), όπου e_i άτομο ή λίστα. Μπορεί να έχει απεριόριστο μήκος, δηλ. να έχει (θεωρητικά) άπειρα στοιχεία, και (θεωρητικά) απεριόριστο βάθος, δηλ. κάποια στοιχεία της να είναι λίστες, που περιέχουν άλλες λίστες κ.ο.κ.

Μια συμβολοσειρά είναι μια ακολουθία χαρακτήρων, που συνήθως βρίσκεται σε διπλά εισαγωγικά (βλ. σχήμα 1). Οι λίστες και οι συμβολοσειρές αποτελούν τις ακολουθίες.



Σχήμα 1. Τύποι

Το μόνο που διαθέτει είναι τα *ειδικά σύμβολα* T (true) και NIL. Το NIL είναι ταυτόσημο με την κενή λιστα '()' και αντιπροσωπεύει το ψεύδος μιας πρότασης, ενώ το T την αλήθεια.

Μια λίστα είναι της μορφής (e_1 e_2 … e_n), όπου e_i άτομο ή λίστα. Μπορεί να έχει απεριόριστο μήκος, δηλ. να έχει (θεωρητικά) άπειρα στοιχεία, και (θεωρητικά) απεριόριστο βάθος, δηλ. κάποια στοιχεία της να είναι λίστες, που περιέχουν άλλες λίστες κ.ο.κ.

Μια συμβολοσειρά είναι μια ακολουθία χαρακτήρων, που συνήθως βρίσκεται σε διπλά εισαγωγικά (βλ. σχήμα 1). Οι λίστες και οι συμβολοσειρές αποτελούν τις ακολουθίες.

Ένας συναρτησιακός τύπος (ή τύπος) είναι μια έκφραση που έχει τη μορφή:

(<function-name> $<arg_1>$ … $<arg_n>$)

όπου <function-name> είναι το *όνομα* μιας συνάρτησης (είτε θεμελιώδους-ενσωματωμένης είτε ορισμένης από τον χρήστη) και κάθε $<arg_i>$ είναι μια συμβολική έκφραση ή άλλος συναρτησιακός τύπος. Τα $<arg_i>$ αποτελούν τα *ορίσματα* ή *παράμετροι* της συνάρτησης.

Η LISP είναι κυρίως γλώσσα διερμηνευόμενη (παρ' ότι συνήθως υπάρχει ταυτόχρονα και μεταγλωττιστής). Ο *κύκλος εκτέλεσης του διερμηνευτή* της LISP είναι ο εξής:

ΑΝΑΓΝΩΣΗ-ΕΚΤΙΜΗΣΗ-ΕΚΤΥΠΩΣΗ
(READ-EVAL-PRINT)

ΑΝΑΓΝΩΣΗ: Γίνεται ανάγνωση του προγράμματος και κρατούνται θέσεις μνήμης για τις μεταβλητές και παραμέτρους που συναντώνται. Αυτό ονομάζεται δέσμευση (binding). Οι μεταβλητές και οι παράμετροι είναι σύμβολα LISP.

ΕΚΤΙΜΗΣΗ: Γίνεται εκτίμηση, δηλ. βρίσκεται το αποτέλεσμα (ή η τιμή) των συμβολικών εκφράσεων και των συναρτησιακών τύπων του προγράμματος.

ΕΚΤΥΠΩΣΗ: Εκτυπώνεται στην οθόνη το αποτέλεσμα (ή η έξοδος) του προγράμματος.

2.2 Κανόνες Εκτίμησης

Για την εκτίμηση (evaluation) των σ-εκφράσεων υπάρχουν συγκεκριμένοι κανόνες, που συνοψίζονται παρακάτω, όπου το σύμβολο '*' παριστάνει την προτροπή (prompt) του περιβάλλοντος LISP (άλλη συνήθης προτροπή είναι το '>') και το '→' σημαίνει «εκτιμάται σε» και υπονοεί το πάτημα του RETURN:

- Κάθε αριθμός εκτιμάται στον εαυτό του. Π.χ.
  * 5 → 5
- Τα ειδικά σύμβολα εκτιμώνται στον εαυτό τους. Π.χ.
  * t → T
  * nil → NIL
- Κάθε σύμβολο εκτιμάται στην τιμή του, εκτός αν υπάρχει το **quote** (**'**) μπροστά απ' αυτό, οπότε εκτιμάται. στον εαυτό του. Ουσιστικά, κάθε σύμβολο χωρίς quote είναι μια *μεταβλητή* (ή *παράμετρος*). Με quote είναι μια *σταθερά*. Μια μεταβλητή που δεν της έχει δοθεί τιμή είτε φανερά είτε κατά την εκτέλεση του προγράμματος, έχει την τιμή NIL. Π.χ.
  * mark → 8 (η τιμή της)
  * (quote mark) → MARK ή
  * 'mark → MARK
- Κάθε λίστα εκτιμάται στην τιμή της, εκτός αν υπάρχει **quote** (**'**) μπροστά από τη λίστα, οπότε εκτιμάται στον εαυτό της. Μια λίστα χωρίς quote είναι ένας *συναρτησιακός τύπος*, αλλιώς είναι μια *τιμή*.Εκτίμηση ενός τύπου σημαίνει εκτίμηση των ορισμάτων του και εφαρμογή της αντίστοιχης συνάρτησης στα αποτελέσματα. Π.χ.
  * (+ 2 3) → 5

    (εκτιμάται το 2 στον εαυτό του, όπως και το 3 και εφαρμόζεται η συνάρτηση '+', του αθροίσματος, οπότε επιστρέφεται το αποτέλεσμα: 5).
  * (* 2 (+ 3 4)) → 14

    (εκτιμάται το 2 στον εαυτό του, το (+ 3 4) κατ' αντιστοιχία με το παραπάνω στο 7 και εφαρμόζεται η συνάρτηση '*', του πολλαπλασιασμού, οπότε επιστρέφεται το αποτέλεσμα: 14).
  * '(1 2 3) → (1 2 3)

    (εκτιμάται στον εαυτό του, λόγω quote).

Ο χρήστης μπορεί να παρακάμψει την παρουσία του quote, χρησιμοποιώντας τη συνάρτηση **eval**, η οποία εκτελεί αναγκαστική εκτίμηση μιας σ-έκφρασης. Π.χ.

* ‘(/ 10 2) → (/ 10 2)
* (eval ‘(/ 10 2)) → 5

3. ΘΕΜΕΛΙΩΔΕΙΣ ΣΥΝΑΡΤΗΣΕΙΣ

3.1 Διαχείρισης Λιστών

Συναρτήσεις προσπέλασης

• **car** (ή **first**)

Σύνταξη: (car <list>)

Επιστρέφει: το πρώτο στοιχείο της λίστας. Π.χ.

* (car ‘(1 2 3)) → 1

* (car ‘((a b) c d)) → (A B)

• **cdr** (ή **rest**)

Σύνταξη: (cdr <list>)

Επιστρέφει: τη λίστα χωρίς το πρώτο στοιχείο της. Π.χ.

* (cdr ‘(1 2 3)) → (2 3)

* (cdr ‘((a b) c d)) → (C D)

• **cxxxxr (x ≡ a, d)** 1-4

Με τη σύνταξη αυτή δίνεται η δυνατότητα πραγματοποίησης πολλών συνδυασμών-συναρτήσεων, εναλλάσσοντας ‘a’ και ‘d’. Π.χ.

(caadr lista) ≡ (car (car (cdr lista)))

(cdadr lista) ≡ (cdr (car (cdr lista)))

• **last**

Σύνταξη: (last <list>)

Επιστρέφει: Τη λίστα με μόνο το τελευταίο στοιχείο. Π.χ.

* (last ‘(1 2 3)) → (3)

• **butlast:** (butlast <list> <n>)

Σύνταξη: (butlast <list> <n>)

Επιστρέφει: Τη λίστα χωρίς τα τελευταία n στοιχεία. Π.χ.

* (butlast ‘(1 2 3 4) 2) → (1 2)

• **nthcdr**

Σύνταξη: (nthcdr <n> <list>)

Επιστρέφει: Τη λίστα χωρίς τα πρώτα n στοιχεία. Π.χ.

* (nthcdr 2 ‘(1 2 3 4)) → (3 4)

• **length**

Σύνταξη: (length <list>)

Επιστρέφει: Το μήκος της λίστας (ακέραιος). Π.χ.

* (length ‘(a 2 b)) → 3

• **reverse**

Σύνταξη: (reverse <list>)

Επιστρέφει: Την ανάστροφη λιστα. Π.χ.

* (reverse ‘(1 2 3)) → (3 2 1)

Συναρτήσεις σύνθεσης

• **cons**

Σύνταξη: (cons <atom> <atom>) ή (cons <list> <atom>)

Επιστρέφει: Το αντίστοιχο ζεύγος. Π.χ.

* (cons ‘x ‘a) → (X **.** A)

* (cons ‘(a b) ‘c) → ((A B) **.** C)

• **cons**

Σύνταξη: (cons <s-expression> <list>)

Επιστρέφει: Τη λίστα με επί πλέον πρώτο στοιχείο τη <s-expression>. Π.χ.

* (cons ‘x ‘(a b)) → (X A B)

• **list**

Σύνταξη: (list <s-expression>*) (Το * σημαίνει μία ή περισσότερες επαναλήψεις)

Επιστρέφει: Μια λίστα που περιέχει τις <s-expression>. Π.χ.

* (list ‘a ‘(2 3) ‘b) → (A (2 3) B)

• **append**

Σύνταξη: (append <list>*) (Το * σημαίνει μία ή περισσότερες επαναλήψεις)

Επιστρέφει: Μια λίστα που συγχωνεύει όλες τις <list>. Π.χ.

* (append ‘(a b) ‘(c) ‘(d)) → (A B C D)

Συναρτήσεις Τροποποίησης

• **push**

Σύνταξη: (push <s-expression> <symbol>), όπου το <symbol> έχει σαν τιμή μια λίστα

Επιστρέφει: Τη λίστα με επί πλέον πρώτο στοιχείο την <s-expression>

Πλευρικό αποτέλεσμα: Η τιμή του <symbol> γίνεται η νέα λίστα. Π.χ.

* x → (A B C)

* (push 1 x) → (1 A B C)* x → (1 A B C)

• **pop**

Σύνταξη: (pop <symbol>), όπου το <symbol> έχει σαν τιμή μια λίστα

Επιστρέφει: Τη λίστα με χωρίς πρώτο στοιχείο της

Πλευρικό αποτέλεσμα: Η τιμή του <symbol> γίνεται η νέα λίστα. Π.χ.

* x → (1 A B C)

* (pop x) → (A B C)* x → (A B C)

3.2 Διαχείρισης Λιστών-Ζευγών

• **assoc**

Σύνταξη: (assoc <s-expression> <alist>), όπου <alist> είναι λίστα ζευγών

Επιστρέφει: Το ζεύγος (το πρώτο που συναντά), που έχει σαν πρώτο στοιχείο την <s-expression>. Π.χ.

* (assoc ‘x ‘((y.b) (x.a) (z.c)) → (x.a)

• **rassoc**

Σύνταξη: (rassoc <s-expression> <alist>), όπου <alist> είναι λίστα ζευγών

Επιστρέφει: Το ζεύγος που έχει σαν δεύτερο στοιχείο την <s-expression>. Π.χ.

* (rassoc ‘c ‘((y.b) (x.a) (z.c)) → (z.c)

• **acons**

Σύνταξη: (acons <atom1> <atom2> <alist>), όπου <alist> είναι λίστα ζευγών

Επιστρέφει: Τη λίστα ζευγών με επί πλέον πρώτο στοιχείο το ζεύγος των δύο ατόμων. Π.χ.

* (acons ‘z ‘c ‘((x.a))) → ((z.c) (x.a))

3.3 Συναρτήσεις καταχώρησης

• **setq** (άμεση καταχώρηση)

Σύνταξη: (setq {<symbol> <s-expression>/<form>}*) (Το σύμβολο ‘/’ σημαίνει ‘ή’)

Λειτουργία: Εκτιμάται η κάθε <s-expression> και το αποτέλεσμα καταχωρείται στο αντίστοιχο <symbol>

Επιστρέφει: Το αποτέλεσμα της εκτίμησης της τελευταίας <s-expression>. Π.χ.

* (setq x ‘(1 2 3)) → (1 2 3)

* x → (1 2 3)

* (setq x ‘(1 2 3) y 5 z (+ 5 4)) → 9

* x → (1 2 3)

* y → 5

* z → 9

• **set** (έμμεση καταχώρηση)

Σύνταξη: (setf {<s-expression-1>/<form-1> <s-expression-2>/<form-2>}*)

Λειτουργία: Εκτιμώνται οι κάθε <s-expression-1>/<form-1> και <s-expression-2>/<form-2>. Το αποτέλεσμα της κάθε <s-expression-2>/<form-2> καταχωρείται στο αποτέλεσμα της αντίστοιχης<s-expression-1>/<form-1>, το οποίο πρέπει να είναι σύμβολο.

Επιστρέφει: Το αποτέλεσμα της εκτίμησης της πρώτης <s-expression-2>. Π.χ.

* weekday → monday
* (set weekday '(1)) → (1)
* weekday → monday
* monday → (1)

•**setf** (γενικευμένη καταχώρηση)

Σύνταξη: (setf {<s-expression-1>/<form-1> <s-expression-2>/<form-2>}*)

Λειτουργία: Εκτιμώνται οι κάθε <s-expression-1>/<form-1> και <s-expression-2>/<form-2>. Το αποτέλεσμα της κάθε <s-expression-2>/<form-2> καταχωρείται στο αποτέλεσμα της αντίστοιχης <s-expression-1>/<form-1>, το οποίο μπορεί να είναι οτιδήποτε.

Επιστρέφει: Το αποτέλεσμα της εκτίμησης της πρώτης <s-expression-2>. Π.χ.

* (setf x '(1 2 3)) → (1 2 3)
* x → (1 2 3)
* (setf (car x) 'a) → a
* x → (a 2 3)
* (setf (cdr x) '(b c)) → (b c)
* x → (a b c)

3.4 Αριθμητικές Συναρτήσεις

Βασικών πράξεων

• + , - , * , / , float

Π.χ.

* (/ 27 9) → 3
* (/ 4.16 1.3) → 3.2
* (/ 22 7) → 22/7

* (float (/ 22 7)) → 3.14...

Υπολογισμών

• round, max, min, expt, sqrt, abs

Π.χ.

* (max 2 4 3) → 4
* (min 2 4 3) → 2
* (expt 2 3) → 8 (δηλ. 2^3)
* (sqrt 6.25) → 2.5
* (sqrt -9) → #C(0.0 3.0) (ο αντίστοιχος μιγαδικός)
* (abs –5.5) → 5.5

* (round (/ 22 7)) → 3 (πλησιέστερος ακέραιος, πρώτη γραμμή)
  1/7 (υπόλοιπο, δεύτερη γραμμή)
* (setf x (round (/ 22 7))) → 3
* x → 3

Αναγνώρισης

zerop: T αν το όρισμά του έχει τιμή ‘0’
plusp: T αν το όρισμά του είναι θετικός αροθμός
minusp: T αν το όρισμά του είναι αρνητικός αριθμός
evenp: T αν το όρισμά του είναι άρτιος (ακέραιος) αριθμός
oddp: T αν το όρισμά του είναι περιττός (ακέραιος) αριθμός
> : T αν τα ορίσματά του (τουλάχιστον 2) είναι κατά φθίνουσα σειρά
< : T αν τα ορίσματά του (τουλάχιστον 2) είναι κατ’ αύξουσα σειρά

3.5 Συναρτήσεις Αναγνώρισης Τύπων

atom: T αν το όρισμά του είναι άτομο (δηλ. αριθμός ή σύμβολο)
numberp: T αν το όρισμά του είναι αριθμός
symbolp: T αν το όρισμά του είναι σύμβολο
listp: T αν το όρισμά του είναι λίστα
null: T αν το όρισμά του είναι μια κενή λίστα
endp: T αν το όρισμά του είναι μια κενή λίστα
(Η διαφορά των null και endp έγκειται στο γεγονός ότι η null μπορεί να έχει όρισμα που δεν είναι λίστα, ενώ η endp δεν μπορεί, οδηγεί σε σφάλμα εκτέλεσης).

3.6 Συναρτήσεις Σύγκρισης

Ισότητας

Γενική Σύνταξη: (<eq-fun> <argum1> <argum2>)
Λειτουργία: Εκτιμώνται τα <argum1> <argum2> και μετά συγκρίνονται τα αποτελέσματά τους, ανάλογα με το ποιά είναι η συνάρτηση σύγκρισης <eq-fun>.

• =
Επιστρέφει: T όταν πρόκειται για δύο αριθμούς ίδιας τιμής, αλλά όχι απαραίτητα και ίδιου τύπου. Π.χ.
* (= 2 2.0) → T

• **eq**
Επιστρέφει: T όταν πρόκειται για δύο ίδια σύμβολα. Π.χ.
* (eq ‘exit ‘exit) → T

• **eql**
Επιστρέφει: T όταν πρόκειται για δύο ίδια σύμβολα ή δύο αριθμούς ίδιας τιμής και τύπου. Π.χ.
* (eql 2 2.0) → NIL
* (eql 2 2) → T

•**equal**

Επιστρέφει: T όταν πρόκειται για δύο απόλυτα ίδιες εκφράσεις (ελέγχει αν ικανοποιούν το eq, αν όχι, τότε ελέγχει αν είναι λίστες και τα στοιχεία τους ικανοποιούν το equal κ.ο.κ.). Π.χ.

* (equal '(+ α b) '(+ a b)) → T

* (equal 2 2.0) → NIL

* (equal 2 2) → T

•equalp

Επιστρέφει: T όταν πρόκειται για δύο ίδιες εκφράσεις. Π.χ.

* (equal (+ 2 3) 5.0) → NIL, ενώ* (equalp (+ 2 3) 5.0) → T

Ανισότητας

< , <= , > , >= (Εφαρμόζονται μόνο σε αριθμούς)

Επιστρέφουν: T όταν ισχύει η αντίστοιχη ανισότητα μεταξύ των δύο αριθμών. Π.χ.

* (> 3 2.5) → T* (<= 3.0 3) → T

3.7 Συνάρτηση Συμμετοχής

Η σύνάρτηση συμμετοχής **member** είναι μια συνάρτηση που ελέγχει αν ένα στοιχείο ανήκει σε μια λίστα (σε πρώτο επίπεδο). Η σύνταξή της είναι η εξής:

* (member <element> <list form>)

Εκτιμάται ο <list form>, το αποτέλεσμα του οποίου πρέπει να είναι μια λίστα, και γίνεται έλεγχος αν το <element> είναι στοιχείο της λίστας που προέκυψε. Η σύγκριση των στοιχείων γίνετασι με βάση τη συνάρτηση σύγκρισης το eql. Τελικά, επιστρέφεται το τμήμα της λίστας από το στοιχείο (συμπεριλαμβανομένου) και δεξιά.

Μπορούμε ν' αλλάξουμε τη συνάρτηση σύγκρισης μέσω της λέξης κλειδί test (βλ. παραδείγματα).

Παραδείγματα:

* (member 'a '(b c d a e)) → (a e)

* (member 'a '(b c (d a) e)) → NIL (δεν είναι μέλος σε πρώτο επίπεδο)

* (member 3 ‘(2 3.0 5)) → NIL

* (member 3 ‘(2 3.0 5) :test #’equalp) → (3.0 5)

4. ΣΥΝΑΡΤΗΣΕΙΣ ΧΡΗΣΤΗ-ΣΥΝΑΡΤΗΣΗ LET

4.1 Ορισμός-Κλήση Συναρτήσεων

Η LISP διαθέτει μια θεμελιώδη συνάρτηση, την defun, μέσω της οποίας ο προγραμματιστής ορίζει τις συναρτήσεις ενός προγράμματος για τη λύση ενός προβλήματος. Η σύνταξη της defun έχει ως ακολούθως:

(**defun** <function-name> (<$param_1$> … <$param_n$>)
    <$form_1$>
      …
    <$form_m$>)

όπου <function-name> είναι το όνομα της συνάρτησης που θέλουμε να ορίσουμε, τα <$param_i$> είναι τα *ορίσματα* ή *τυπικές παράμετροι* της συνάρτησης και τα <$form_i$> είναι συναρτησιακοί τύποι.

Π.χ. ο παρακάτω κώδικας LISP ορίζει μια συνάρτηση mesos-oros, που έχει δύο ορίσματα (num1 και num2) και υπολογίζει το μέσο όρο των ορισμάτων της.

```
(defun mesos-oros (num1 num2)
  (/ (+ num1 num2) 2))
```

Κλήση μιας συνάρτησης σημαίνει την εφαρμογή της σε κάποιες τιμές των παραμέτρων (πραγματικές παράμετροι). Π.χ. μια κλήση της παραπάνω συνάρτησης είναι:

* (mesos-oros 5 3) → 4

Οι συναρτήσεις χρήστη μπορούν να χρησιμοποιηθούν όπως και οι ενσωματωμένες. Για παράδειγμα, η παρακάτω συνάρτηση ‘emb-trapez’ χρησιμοποιεί στον ορισμό της την προηγούμενη συνάρτηση ‘mesos-oros’.

```
(defun emb-trapez (h b1 b2)
    (* h (mesos-oros b1 b2)))
```

4.2 Συνάρτηση Let

Η συνάρτηση **let** χρησιμοποιείται για ανάθεση (αρχικών) τιμών σε μεταβλητές. Η σύνταξή της έχει ως εξής:

```
(let ((<param1>  <init-value1>)
      (<param2>  <init-value2>)
                 . . .
      (<paramn>  <init-valuen>))
  <form1>
     ...
  <formn>)
```

Στην παραπάνω δομή, εκτιμώνται τα <init-value1>, <init-value2>, ... <init-valuen> και τα αποτελέσματά τους καταχωρούνται στα <param1>, <param2>, ... <paramn> αντίστοιχα.

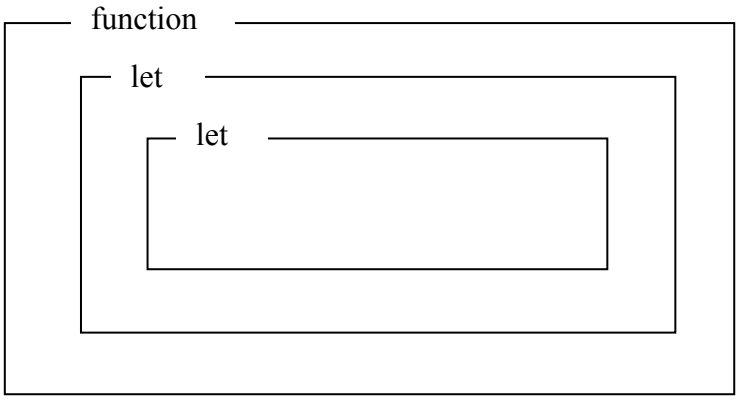
Η χρήση του let κάνει το κώδικα πιο αναγνώσιμο. Π.χ. η παραπάνω συνάρτηση θα μπορούσε να γραφεί:

```
(defun emb-trapez (h b1 b2)
  (let ((ypsos h)
        (imiathr-basewn (mesos-oros b1 b2))
    (* ypsos imiathr-basewn))
```

Αυτό θα γίνει εμφανέστερο αργότερα σε πιο σύνθετα προγράμματα.

Εκτός του let υπάρχει και η παραλλαγή του το let*. Η διαφορά τους είναι ότι το Το let δρα παράλληλα, δηλ. πρώτα εκτιμώνται οι αρχικές τιμές και μετά αποδίδονται στις μεταβλητές, ενώ το let* δρα ακολουθιακά, δηλ. κάθε αρχική τιμή εκτιμάται και αποδίδεται στην αντίστοιχη μεταβλητή με τη σειρά αναγραφής. Αυτό δίνει τη δυνατότητα σε αρχική τιμή κάποιας μεταβλητής να μπορεί να εξαρτάται από την αρχική τιμή κάποιας προηγούμενης μεταβλητής. Ας δούμε το παρακάτω παράδειγμα:

```
* (setf x ‘one)
* (let  ((x ‘two) (y x)) (list x y))
* (ONE TWO)
* (let*  ((x ‘two) (y x)) (list x y))
* (TWO TWO)
```



Σχήμα 2. Νοητοί φράκτες σε συνάρτηση

Ένα άλλο χαρακτηριστικό του let (και φυσικά και του let*) είναι ότι δημιουργεί ένα νοητό φράχτη (virtual fence) μέσα σε μια συνάρτηση. Επίσης, η ίδια η συνάρτηση δημιουργεί ένα νοητό φράχτη γύρω της (βλ. Σχ. 2). Μεταβλητές μέσα σ’ ένα φράχτη έχουν νόημα στους (ή είναι ορατές από τους) εσωτερικούς του φράχτες, όχι όμως αντίστροφα. Δηλ. μεταβλητές μέσα σ’ ένα φράχτη παίζουν το ρόλο τοπικών μεταβλητών του φράχτη. Π.χ. στην παρακάτω συνάρτηση

```
(defun funx (p1 p2)
   (let ((x p1))
```

```
 …
(let ((y p2))
      …        )
   ...           )
... )
```

οι παράμετροι p1, p2 είναι ορατές και από το σώμα της συνάρτησης και από τα δύο let, η μεταβλητή x είναι ορατή και από τα δύο let και η y μόνο από το δεύτερο (εσωτερικό) let. Επομένως π.χ. η χρήση της y στο σώμα της συνάρτησης ή στο σώμα του πρώτου (εξωτερικού) let δεν έχει νόημα.

4.3 Ειδικές (ή Καθολικές) Μεταβλητές

Σύμφωνα με τα παραπάνω, όλες οι μεταβλητές που χρησιμοποιούνται σε μια συνάρτηση είναι ουσιαστικά τοπικές μεταβλητές, διότι η εμβέλειά τους περιορίζεται από την ύπαρξη των νοητών φρακτών. Όμως, συχνά έιναι απαραίτητη η ύπαρξη μεταβλητών καθολικής εμβέλειας, δηλ. μεταβλητών που να είναι ορατές από (δηλ. να έχουν νόημα σε) όλες τις συναρτήσεις ενός προγράμματος.

Γι' αυτό η LISP έχει προνοήσει για την κάλυψη τέτοιων περιπτώσεων. Έτσι, μπορούμε να δηλώσουμε *ειδικές μεταβλητές* (special variables), που ξεφεύγουν από τις κανονικές, που ονομάζονται *λεξικογραφικές μεταβλητές* (lexical variables), και έχουν καθολική εμβέλεια.

Η δήλωση μιας τέτοιας μεταβλητής γίνεται μέσω του defvar:

(defvar <όνομα-μεταβλητής>)

Συνήθως, το όνομα μιας ειδικής μεταβλητής περικλείεται ανάμεσα σε δύο '*', ώστε να αναγνωρίζεται εύκολα στο πρόγραμμα. Έτσι, για να δηλώσουμε μια ειδική (καθολική) μεταβλητή με όνομα 'x' γράφουμε:

```
(defvar *x*)
```

Οι δηλώσεις τέτοιων μεταβλητών πρέπει να γίνονται πριν από οποιαδήποτε συνάρτηση τις χρησιμοποιεί. Γι' αυτό συνήθως οι δηλώσεις αυτές γίνονται στην αρχή ενός προγράμματος.

Μπορούμε να δώσουμε και αρχική τιμή σε μια ειδική μεταβλητή:

```
(defvar *x* 2)
```

Αν δεν δώσουμε, θεωρείται ως έχουσα την τιμή NIL.

5. ΕΛΕΓΧΟΣ ΡΟΗΣ ΠΡΟΓΡΑΜΜΑΤΟΣ

Η LISP διαθέτει αρκετές διατάξεις ελέγχου της ροής ενός προγράμματος, και επιλογής (ή διακλάδωσης ή απόφασης) και επανάληψης.

5.1 Διατάξεις επιλογής

Διάταξη if

Η διάταξη if έχει την παρακάτω σύνταξη:

(**if** <condition> <then form> [<else form>])

όπου ότι υπάρχει μεταξύ '[' και ']' είναι προεραιτικό.

Η λειτουργία της έχει ως εξής: Εκτιμάται η συνθήκη <condition>. Αν το αποτέλεσμα είναι διάφορο του NIL, εκτιμάται η έκφραση <then form> και επιστρέφεται η τιμή της, αλλιώς εκτιμάται η <else form>, αν υπάρχει, και επιστρέφεται η τιμή της, αλλιώς (δηλ. αν δεν υπάρχει) επιστρέφεται NIL.

Μπορεί να γίνει χρήση των λογικών τελεστών **not**, **or** και **and** στη συνθήκη, αλλά και εν γένει σε μια έκφραση . Π.χ.

```
(defun mesos-oros-10 (num1 num2)
   (if (and (< num1 10)
            (< num2 10))
      (mesos-oros num1 num2)
       nil))
```

Η συνάρτηση αυτή παίρνει δύο ορίσματα, εξετάζει αν είναι και τα δύο αριθμοί μικρότεροι του 10 και αν είναι καλεί τη συνάρτηση ‘mesos-oros’, η οποία επιστρέφει τον μέσο όρο των αριθμών. Αν δεν είναι επιστρέφει NIL. Στην περίπτωση αυτή το NIL θα μπορούσε να παραληφθεί, διότι έτσι κι αλλιώς αυτό επιστρέφεται. Η παρουσία του απλώς συμβάλλει στην αναγνωσιμότητα του κώδικα.

Διάταξη cond

Η διάταξη cond έχει την παρακάτω σύνταξη:

(**cond** (<condition1> [<action1-1>, <action1-2>, ... <action1-n>])
(<condition2> [<action2-1>, <action2-2>, ... <action2-n>])
. . .
(<conditionm> [<actionm-1>, <actionm-2>, ... <actionm-n>]))

όπου τα <actioni-j> είναι προεραιτικά.

Η λειτουργία της έχει ως εξής: Εκτιμώνται οι συνθήκες <condition1>, …, <conditionm> με τη σειρά. Με το πρώτο <conditioni> που θα δώσει αποτέλεσμα διάφορο του NIL, εκτιμώνται οι αντίστοιχες έκφρασεις ενέργειας <actioni-j> και επιστρέφεται η τιμή της τελευταίας. Αν δεν υπάρχουν εκφράσεις ενέργειας, αυτό που επιστρέφεται είναι το αποτέλεσμα του <conditioni>. Αν καμμία συνθήκη δεν δώσει αποτέλεσμα διάφορο του NIL, τότε επιστρέφει NIL. Π.χ.

```
(defun mesos-oros-10 (num1 num2)
    (cond ((and (< num1 10) (< num2 10))
           (mesos-oros num1 num2))
          (t  nil)))
```

Η συνάρτηση αυτή είναι η ίδια με την προηγούμενη γραμμένη με cond αντί για if. Προσέξτε ότι χρησιμοποιούμε για τελευταία συνθήκη το ‘t’. Αυτό είναι μια κοινή τεχνική, όταν θέλουμε η τελευταία έκφραση ενέργειας να εκτελεστεί οπωσδήποτε, αν καμμία από τις προηγούμενες δεν έχει εκτελεστεί.

Διάταξη case

Η διάταξη case έχει την παρακάτω σύνταξη:

(**case** <key term>
(<key1> <action1-1>[, <action1-2>, ... <action1-n>])
(<key2> <action2-1>[, <action2-2>, ... <action2-n>])
. . .
(<keym> <actionm-1>[, <actionm-2>, ... <actionm-n>]))

Η λειτουργία της έχει ως εξής: Εκτιμάται ο <key form> και συγκρίνεται με καθένα από τα <key1>, …, <keym> (χωρίς εκτίμηση) με τη σειρά με βάση το κατηγόρημα (συνάρτηση) σύγκρισης eql. Με το πρώτο <keyi> για το οποίο η σύγκριση θα δώσει αποτέλεσμα T, εκτιμώνται οι αντίστοιχες έκφρασεις ενέργειας <actioni-j> και επιστρέφεται η τιμή της τελευταίας. Αν καμμία σύγκριση δεν δώσει αποτέλεσμα T, τότε επιστρέφει NIL, εκτός εάν η τελευταία πρόταση της case έχει σαν <keym> το "otherwise" ή το "t", οπότε εκτελούνται οι ενέργειές της. Π.χ.

```
(defun compute-area (shape b h)
  (case shape
    (triangle (* 0.5 b h))
    (rectangle (* b h))
    (otherwise 0)))
```

και

* (compute area 'rectangle 4.5 6.0) → 27.0

Αν κάποιο (α) από τα <keyi> είναι λίστα, τότε η σύγκριση γίνεται με βάση το κατηγόρημα (συνάρτηση) member. Π.χ.

```
(defun compute-area (shape b h)
```

```
(case shape
  (triangle (* 0.5 b h))
  ((rectangle door window)(* b h))
  (otherwise 0)))
```

Οπότε

* (compute area ‘door 2.5 0.9) → 2.25

Χειρισμός ακολουθίας συναρτησιακών τύπων

Μερικές φορές θέλουμε να συνδυάσουμε εμφανώς διάφορους τύπους σε μια ακολουθία, ώστε να εκτελεστούν όλοι με τη σειρά και να επιστραφεί το αποτέλεσμα του ενός (ενώ τα αποτελέσματα των άλλων να λειτουργήσουν σαν πλευρικά). Για την περίπτωση αυτή η LISP διαθέτει δύο συναρτήσεις, την prog1 και την progn, με την εξής σύνταξη:

(prog1 <form1> <form2> … <formn>)

(progn <form1> <form2> … <formn>)

Εκτελούνται τα <formi> με τη σειρά και επιστρέφεται σαν αποτέλεσμα το αποτέλεσμα του <form1>, στην περίπτωση του prog1 και του <formn> στην περίπτωση του progn.

5.2 Διατάξεις επανάληψης

Διάταξη dotimes

Η διάταξη dotimes έχει την παρακάτω σύνταξη:

(dolist (<parameter> <up-bound form> [<result form>])

<form1>
… <formn>)

Η λειτουργία της έχει ως εξής: Εκτιμάται το <up-bound form>, που πρέπει να έχει σαν αποτέλεσμα ένα αριθμό, έστω n. Στη συνέχεια, αποδίδονται οι ακέραιες τιμές 0 … n-1 στην <parameter> διαδοχικά. Για κάθε τιμή εκτελούνται τα <form1> … <formn>. Τέλος, εκτιμάται το <result form> και επιστρέφεται η τιμή του. Αν δεν υπάρχει <result form>, επιστρέφεται NIL. Π.χ.

```
(defun plus-one (num-list)
  (let  ((new-list nil))
     (dotimes (counter (length num-list) (reverse new-list))
          (push (+ 1 elem) new-list))))
```

Η παραπάνω συνάρτηση αυξάνει κατά ένα τις τιμές των στοιχείων μιας λίστας. Δηλ.

* (plus-one '(1 2 3)) → (2 3 4)

Στον παρακάτω πίνακα φαίνονται οι τιμές των διαφόρων παραμέτρων/μεταβλητών κατά την εκτέλεση της διάταξης επανάληψης. Όπως φαίνεται, γίνονται τρεις επαναλήψεις, όσα και το μήκος (length) της num-list. Μετά το τέλος των τριών επαναλήψεων η τιμή της new-list είναι η λίστα (4 3 2), που είναι η αντίστροφη (λόγω της λειτουργίας της push) από αυτήν που θέλουμε σαν αποτλεσμα. Γι' αυτό στη θέση του <result form> θέτουμε την εφαρμογή της συνάρτησης αντιστροφής (reverse) στην new-list.

counter	Εκτέλεση push	new-list
0	$1^{η}$	(2)
1	$2^{η}$	(3 2)
2	$3^{η}$	(4 3 2)

<u>Διάταξη dolist</u>

Η διάταξη dolist έχει την παρακάτω σύνταξη:

```
(dolist (<parameter> <list form> [<result form>])
   <form1>
      …
   <formn>)
```

Η λειτουργία της έχει ως εξής: Εκτιμάται το <list form>, που πρέπει να έχει σαν αποτέλεσμα λίστα. Στη συνέχεια αποδίδονται τα στοιχεία της λίστας ένα-ένα σαν τιμές στο <parameter>. Σε κάθε απόδοση τιμής εκτελούνται τα <form1> … <formn>. Τέλος εκτιμάται το <result form> και το αποτέλεσμά του επιστρέφεται σαν αποτέλεσμα της διάταξης. Αν δεν υπάρχει <result form>, επιστρέφεται NIL. Π.χ. η παραπάνω συνάρτηση γράφεται τώρα ως εξής:

```
(defun plus-one (num-list)
   (let  ((new-list nil))
      (dolist  (elem num-list (reverse new-list))
            (push (+ 1 elem) new-list))))
```

Στον παρακάτω πίνακα φαίνονται οι τιμές των διαφόρων παραμέτρων/μεταβλητών κατά την εκτέλεση της διάταξης επανάληψης. Όπως φαίνεται, και δω γίνονται τρεις επαναλήψεις, όσα και τα στοιχεία της num-list.

elem	Εκτέλεση push	new-list
1	$1^{η}$	(2)
2	$2^{η}$	(3 2)
3	$3^{η}$	(4 3 2)

Μια επανάληψη dolist είναι δυνατόν να διακοπεί με τη χρήση μιας πρότασης return, που έχει την ακόλουθη σύνταξη:

(return <form>).

Όταν η ροή εκτέλεσης του προγράμματος βρει και εκτελέσει μια τέτοια πρόταση, τότε σταματούν οι επαναλήψεις και επιστρέφεται σαν αποτέλεσμα το αποτέλεσμα της <σ-έκφραση>. Π.χ. αν θέλαμε να δημιουργήσουμε μια συνάρτηση που να προσθέτει ένα σε κάθε στοιχείο μιας λίστας μέχρι όμως να συναντήση τον αριθμό 5, θα γράφαμε:

```
(defun plus-one (num-list)
   (let  ((new-list nil))
      (dolist  (elem num-list (reverse new-list))
         (if (= elem 5)
            (return (append (reverse new-list) (member 5 num-list))
            (push (+ 1 elem) new-list)))))
```

Τότε,

* (plus-one '(1 4 3 8 5 6)) → (2 5 4 9 5 6)

Διάταξη do

Η διάταξη do είναι συνθετότερη της dolist, αλλά και με περισσότερες δυνατότητες. Η σύνταξή της είναι η εξής:

```
(do ((<param1> [<init-val1> [<update-form1>]])
                  . . .      (<paramn> [<init-valn> [<update-formn>]]))
   (<termin-test> [[<intermed-form>*] <result-form>])
   <form1>
      ...
   <formn>)
```

όπου ένα ή περισσότερα <intermed-form> δεν μπορούν να υπάρχουν αν δεν υπάρχει <result-form>.

Η λειτουργία της έχει ως εξής: Εκτιμώνται τα <init-vali> και καταχωρούνται στις αντίστοιχες παραμέτρους (παράλληλα). Εξετάζεται το <termin-test>. Αν είναι NIL (δηλ. δεν αληθεύει), εκτελείται το σώμα (<form1> … <formn>). Στη συνέχεια εκτιμώνται τα <update-formi> και τα αποτελέσματά τους αποδίδονται στις αντίστοιχες παραμέτρους (παράλληλα). Κατόπιν, εξετάζεται το <termin-test> κ.ο.κ. Αν το <termin-test> βρεθεί να αληθεύει (είναι δηλ. T), τότε εκτιμώνται τα <intermed-form>, αν υπάρχουν, και το <result-form>, αν υπάρχει. Επιστρέφεται δε το αποτέλεσμα του <result-form>. Αν δεν υπάρχει <result-form>, επιστρέφεται NIL. Π.χ. η προηγούμενη συνάρτηση μπορεί να γραφεί ως εξής, με βάση τη διάταξη do:

```
(defun plus-one (num-list)
  (do  ((new-list num-list (cdr new-list))
        (result nil))
      ((null new-list) (reverse result))
      (push  (+ 1 (car new-list)) result)))
```

Όπως και η dolist έτσι και η do μπορεί να διακοπεί με τη χρήση μιας πρότασης return.

Επίσης, όπως και για το let, έτσι και για το do υπάρχει το do*, στο οποίο οι αποδόσεις αρχικών τιμών και ενημερώσεων στις παραμέτρους γίνεται σειριακά (και όχι παράλληλα).

Γενικά αποφεύγεται η χρήση του do, λόγω μεγαλύτερης πολυπλοκότητας, εφ' όσον η χρήση του dotimes ή του dolist είναι επαρκής.

Διάταξη loop

Μια άλλη διάταξη επανάληψης, που δεν χρησιμοποιείται όμως συχνά, είναι η loop, που έχει την εξής σύνταξη:

(loop <form1>, <form2>, ..., <formn>)

Στη διάταξη αυτή τα <form1>, <form2>, ..., <formn> εκτελούνται συνεχώς με τη σειρά αναγραφής, μέχρις ότου εκτελεστεί κάποια πρόταση return, οπότε και

επιστρέφεται το αποτέλεσμά της. Π.χ. η παραπάνω συνάρτηση θα μπορούσε να γραφεί:

```
(defun plus-one (num-list)
   (let  ((new-list nil)
          (rest-list num-list))
        (loop
           (if (endp rest-list)
              (return (reverse new-list)))
           (setf elem (car rest-list))
           (setf rest-list (cdr rest-list))
           (push (+ 1 elem) new-list))))
```

6. ΑΝΑΔΡΟΜΗ ΚΑΙ ΠΑΡΑΜΕΤΡΟΙ ΕΙΔΙΚΟΥ ΣΚΟΠΟΥ

6.1 Αναδρομικές Συναρτήσεις

Μια συνάρτηση ονομάζεται *αναδρομική* (recursive) όταν ορίζεται μέσω του εαυτού της (αναδρομικός ορισμός), όταν δηλ. στον ορισμό της καλεί τον εαυτό της (αναδρομική κλήση). Ένα βασικό στοιχείο σε μια αναδρομική συνάρτηση είναι να υπάρχει μια συνθήκη τερνατισμού, η οποία δίνει τέλος στις κλήσεις της συνάρτησης από τον εαυτό της. Π.χ. η παρακάτω συνάρτηση είναι μια αναδρομική συνάρτηση που αφαιρεί τους αρνητικούς αριθμούς από μια λίστα και την επιστρέφει χωρίς αυτούς:

```
(defun  filter-negs (num-list)
   (cond ((null num-list) nil)
         ((plusp (car num-list))
          (cons (car num-list)
                (filter-negs (cdr num-list))))
         (t (filter-negs (cdr num-list)))))
```

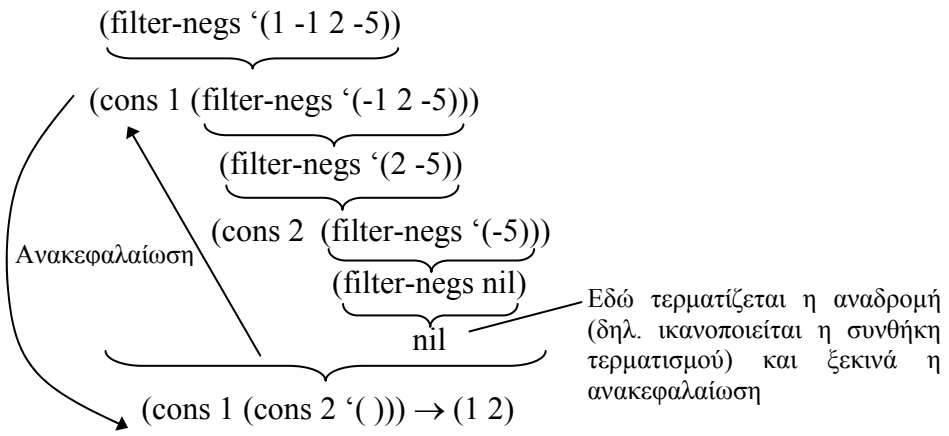
Συνθήκη τερματισμού

Αναδρομικές κλήσεις

Αν καλέσουμε τη συνάρτηση με όρισμα τη λίστα (1 –1 2 –5), τότε

* (filter-negs ‘(1 -1 2 -5)) → (1 5)

Στο παρακάτω σχήμα φαίνεται η διαδικασία της αναδρομικής εκτέλεσης:



Η *αναδρομή* (recursion) είναι χαρακτηριστικό της LISP, διότι εν γένει ο συναρτησιακός προγραμματισμός την ευνοεί. Είναι φυσικό και εύκολο σε μια τέτοια γλώσσα να χρησιμοποιούμε αναδρομή και αναδρομικές συναρτήσεις. Υπάρχουν προβλήματα που λύνονται μόνο με αναδρομή. Επίσης, η χρήση αναδρομής δημιουργεί μικρότερο και κομψότερο κώδικα. Ένα βασικό πρόβλημα της αναδρομής είναι ότι απαιτεί περισσότερο χώρο στη μνήμη απ' ότι μια επανάληψη, διότι όλα τα ενδιάμεσα αποτελέσματα κρατούνται στη μνήμη έως ότου γίνει η ανακεφαλαίωση, δηλ. τερματίσει η αναδρομή και εκτελεστούν όλοι οι μετέωροι υπολογισμοί. Επομένως, γενικά προτιμούμε τις επαναληπτικές διαδικασίες/συναρτήσεις, όπου η λύση μπορεί να δοθεί και με επανάληψη.

6.2 Παράμετροι Ειδικού Σκοπού

Η LISP προσφέρει ένα αριθμό τύπων για δήλωση τυπικών παραμέτρων/ορισμάτων ειδικού σκοπού σε μια συνάρτηση. Οι παράμετροι αυτοί διευκολύνουν τη συγγραφή προγραμμάτων. Τέτοιες παράμετροι είναι οι *προεραιτικές* (optional), οι *υπόλοιπες* (rest), οι *κλειδιά* (key) και οι *βοηθητικές* (auxiliary). Κάθε κατηγορία παραμέτρων δηλώνεται μέσω του αντίστοιχου τύπου: &optional, &rest, &key, &aux. Εδώ θα αναφερθούμε στις δύο πρώτες.

Προαιρετικές Παράμετροι

Δηλώνονται χρησιμοποιώντας το «&optional» μετά από τις κανονικές παραμέτρους/ορίσματα μιας συνάρτησης. Κατά την κλήση της συνάρτησης, μπορεί να υπάρχουν ή να μην υπάρχουν αντίστοιχες πραγματικές παράμετροι (εξ' ου και το όνομα προεραιτικές). Μπορούμε να δηλώσουμε ή να μη δηλώσουμε εξ' ορισμού (default) τιμές για τις προαιρετικές παραμέτρους, οι οποίες χρησιμοποιούνται στην περίπτωση που δεν υπάρχουν πραγματικές παράμετροι στην κλήση της συνάρτησης. Στην περίπωση που δεν δηλώσουμε εξ' ορισμού τιμές, όλες θεωρούνται ότι έχουν εξ' ορισμού τιμή NIL.

Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να δημιουργήσουμε μια συνάρτηση που να υπολογίζει το εμβαδόν ενός τριγώνου ή ενός ορθογωνίου. Για το εμβαδόν ενός τριγώνου χρησιμοποιούμε τον τύπο E=0.5*b*h, ενώ για αυτό ενός ορθογωνίου τον E=b*h. Για να τα συνδυάσουμε σε μια συνάρτηση, κάνουμε χρήση μιας προεραιτικής παραμέτρου, της factor:

```
(defun embadon (b h &optional factor)
  (if (= factor 0.5)
      (* 0.5 b h)
      (* b h)))
```

Η συνάρτηση αυτή όταν δεν δίνεται η προαιρετική παράμετρος, υπολογίζει εμβαδόν ορθογωνίου, ενώ όταν δίνεται ίση με 0.5, υπολογίζει εμβαδόν τριγώνου. Επομένως

* (embadon 4 5 0.5) → 10.0
* (embadon 4 5) → 20

Πολλές φορές χρειάζεται να δώσουμε αρχική τιμή στην προαιρετική παράμετρο είτε για λόγους υλοποίησης είτε για λόγους απλοποίησης της συνάρτησης. Π.χ. η παραπάνω συνάρτηση μπορεί να γραφεί:

```
(defun embadon (b h &optional (factor 1))
  (* factor b h))
```

που είναι μια πιο απλοποιημένη έκδοση, όπου δηλώνουμε ότι η προαιρετική παράμετρος factor, όταν δεν δίνεται παίρνει την τιμή 1. Οι δύο παραπάνω κλήσεις θα δώσουν τα ίδια αποτελέσματα.

Υπόλοιπες Παράμετροι

Μια υπόλοιπη παράμετρος δηλώνεται χρησιμοποιώντας το «&rest» μετά από τις κανονικές παραμέτρους/ορίσματα μιας συνάρτησης. Μπορεί να υπάρχει μόνο μια υπόλοιπη παράμετρος σε μια συνάρτηση. Κατά την κλήση της συνάρτησης, όλες οι πραγματικές παράμετροι μετά τις κανονικες σχηματίζουν μια λίστα που αποδίδεται σαν τιμή στην τυπική υπόλοιπη παράμετρο.

Έστω ότι θέλουμε να δημιουργήσουμε μια συνάρτηση που να υπολογίζει τα εμβαδά ενός ή περισσοτέρων τριγώνων που έχουν κοινή βάση και να επιστρέφει τα αποτελέσματα σε μια λίστα. Η συνάρτησή μας μπορεί να έχει τον εξής ορισμό:

```
(defun m-embadon (b &rest hs)
  (let ((e-list nil))
    (dolist (elem hs (reverse e-list))
      (push (* 0.5 b elem) e-list))))
```

Τότε

* (m-embadon 5 4 6 8) → (10.0 15.0 20.0)

Αυτό που γίνεται είναι ότι η πραγματική παράμετρος ‘5’ αποδίδεται στην b, ενώ οι απομένουσες παράμετροι ‘4’, ‘6’ και ‘8’ σχηματίζουν μια λίστα που αποδίδεται στην υπόλοιπη παράμετρο hs. Στη συνέχεια εκτελείται η m-embadon.

Επίσης

* (m-embadon 4 3 7) → (6.0 14.0)

Δηλ. μετά το πρώτο, που αντιστοιχεί στην κανονική παράμετρο b, δίνουμε όσα επι πλέον ορίσματα θέλουμε, για τον υπολογισμό των αντίστοιχων εμβαδών.

7. ΕΙΣΟΔΟΣ-ΕΞΟΔΟΣ

Η LISP όπως όλες οι γλώσσες προγραμματσιμού προσφέρει και συναρτήσεις για είσοδο δεδομένων και έξοδο αποτελεσμάτων. Για είσοδο δεδομένων από τον χρήστη, διατίθεται η συνάρτηση read. Όταν στη ροή εκτέλεσης βρεθεί η read, τότε σταματά η εκτέλεση και περιμένει μέχρις ότου ο χρήστης πληκτρολογήσει κάποια έκφραση. Π.χ. η παρακάτω έκφραση

* (read) →

αναγκάζει το σύστημα να περιμένει κάποια είσοδο από το πληκτρολόγιο

five → FIVE

Για να ανατεθεί σε κάποια μεταβλητή αυτό που θα πληκτρολογηθεί, χρησιμοποιούμε μια από τις συναρτήσεις ανάθεσης:

* (setf x (read))

Καλό είναι πριν από κάθε read να υπάρχει εκτύπωση κάποιου μηνύματος που ναπληροφορεί τον χρήστη ότι πρέπει κάτι να πληκτρολογήση. Αυτό γίνεται με τη χρήση της συνάρτησης εξόδου print. Αν εκτελέσουμε την παρακάτω έκφραση

* (print ‘(Give a number))

το αποτέλεσμα θα είναι

(GIVE A NUMBER) (αυτό είναι το αποτέλεσμα της ενέργειας της print)
(GIVE A NUMBER) (αυτό είναι η τιμή του συναρτησιακού τύπου)

Ας δούμε την παρακάτω συνάρτηση:

```
(defun print-what-type ()
  (print '(please type a number :))
  (setf num (read))
  (print (append '(you typed) (list num))))
```

Αν την καλέσουμε, θα έχουμε τα παρακάτω (με έντονα αυτά που πληκτρολογεί ο χρήστης):

* (print-what-type) →
(PLEASE TYPE A NUMBER) **45** →

(YOU TYPED 45)
(YOU TYPED 45)

Βέβαια η LISP διαθέτει και δυνατότητες για πιο κομψές εξόδους. Αυτό επιτυγχάνεται με τη συνάρτηση format. Η πιο απλή εφαρμογή της format απλώς εκτυπώνει μια ακολουθία χαρακτήρων (string) στην οθόνη χωρις μετακίνηση σε άλλη γραμμή:

(format t "Hello!") → Hello! (αυτό είναι το αποτέλεσμα της ενέργειας της format)
NIL (αυτό είναι η τιμή του συναρτησιακού τύπου)

όπου το 't' υποδηλώνει την οθόνη (θα μπορούσε να είναι και κάτι άλλο, π.χ. για εκτύπωση σε αρχείο).

Για να τυπώσουμε μια φράση σε νέα γραμμή τοποθετούμε το σύμπλεγμα '~%' εκεί ακριβώς που θέλουμε αλλαγή γραμμής:

(format t “~%Hello!~%What’s your name?”) →
Hello!
What's your name?
NIL

Αν θέλουμε όχι απλώς ν’ αλλάξουμε γραμμή, αλλά ν’αφήσουμε και κενές γραμμές τότε τοποθετούμε ένα ακέραιο αριθμό n πριν το ‘%’, οπότε γίνεται ακκαγή γραμμής και αφήνονται (n-1) κενές γραμμές. Π.χ. η

```
(progn (format t “~%HELLO~3%”)
       (format t “HELLO~%”))
```

θα έχει σαν αποτέλεσμα:

HELLO
HELLO
NIL

δηλ. το δεύτερο HELLO εκτυπώνεται αφού αφεθούν δύο κενές γραμμές από το προηγούμενο.

Το ‘~’ ουσιαστικά πληροφορεί ότι ακολουθεί κάποια *οδηγία εκτύπωσης*. Στην παραπάνω περίπτωση είναι το ‘%’ που σημαίνει αλλαγή γραμμής. Υπάρχουν όμως και άλλες πολλές οδηγίες εκτύπωσης, που ουσιαστικά αποτελούν μια μικρή γλώσσα. Εδώ θα αναφέρουμε μόνο άλλη μια οδηγία, την ‘a’, η οποία σημαίνει τη θέση στην οποία πρέπει να μπει η τιμή ενός ορίσματος που ακολουθεί τη φράση που βρίσκεται σε εισαγωγικά. Π.χ.

```
(setf name ‘giannis)
(format t “~%My name is ~a.” name)
```

Το αποτέλεσμα θα είναι:

My name is GIANNIS.

Μπορούμε να έχουμε περισσότερα του ενός ‘a’ σε μια φράση. Π.χ.

(format t “~%My name is ~a and my hoby is ~a” name hoby)

Μπορούμε επίσης να αφήσουμε συγκεκριμένο αριθμό κενών σε κάθε εκτύπωση τιμής που αντιστοιχεί σε ‘a’. Π.χ. ‘~8a’ σημαίνει να εκτυπωθεί η τιμή του ορίσματος που αντιστοιχεί στο ‘a’ και μετά να αφεθούν 8 κενά πριν την επόμενη εκτύπωση κάποιου στοιχείου.

8. ΣΥΝΑΡΤΗΣΕΙΣ ΜΕ ΟΡΙΣΜΑΤΑ ΣΥΝΑΡΤΗΣΕΙΣ

Ένα κοινό χαρακτηριστικό των συναρτήσεων αυτών είναι ότι χρησιμοποιούν σαν όρισμα κάποια συνάρτηση-διαδικασία. Κάποιες απο αυτές ονομάζονται και συναρτήσεις αντιστοίχησης (mapping functions), διότι αναφέρονται σε αντιστοιχίες μεταξύ λιστών, που υλοποιούνται είτε ως κάποιος μετασχηματισμός είτε ως φιλτράρισμα μιας λίστας.

mapcar

Είναι μια συνάρτηση μετασχηματισμού λίστας. Έχει την εξής σύνταξη:

(mapcar #'<function-name> <list form>*)

όπου τα <list form> είναι τόσα, όσα και τα ορίσματα που παίρνει η <function-name>.

Η λειτουργία της έχει ως εξής: Εφαρμόζεται η <function-name> διαδοχικά με ορίσματα κάθε φορά τα ομοθέσια στοιχεία των λιστών που προκύπτουν από την εκτίμηση των <list form>. Σαν αποτέλεσμα, επιστρέφεται μια λίστα με τα αποτελέσματα των παραπάνω εφαρμογών. Π.χ.

* (mapcar # 'oddp '(1 2 3)) → (T NIL T)
* (mapcar # '= '(1 2 3) '(4 2 1)) → (NIL T NIL)

Στην δεύτερη από τις παραπάνω προτάσεις, εφαρμόζεται η συνάρτηση '=' διαδοχικά στα ζεύγη τιμών (1 4), (2 2) και (3 1) και τα αποτελέσματα των εφαρμογών αυτών επιστρέφονται σε μια λίστα.

lambda

Χρησιμοποιείται για τον ορισμό μιας συνάρτησης-διαδικασίας που δεν χρειάζεται (ή δεν θέλουμε) να έχει κάποιο συγκεκριμένο όνομα. Συνήθως αφορά διαδικασίες που χρησιμοποιούνται μόνο σ' ένα σημείο του προγράμματος και δεν χρειάζονται πουθενά αλλού, ονομάζονται δε *συναρτήσεις lambda*.

Η σύνταξη της lambda είναι η ίδια με αυτή της defun, μόνο που δεν υπάρχει όνομα συνάρτησης και το defun έχει αντικατασταθεί από το lambda:

(lambda (<$param_1$> … <$param_n$>)
    <$form_1$>
      …
    <$form_m$>)

Οι συναρτήσεις lambda χρησιμοποιούνται σε συναρτήσεις που έχουν σαν όρισμα μια συνάρτηση, οπότε στη θέση του ονόματος της συνάρτησης μπαίνει ο ορισμός μιας συνάρτησης lambda. Π.χ. μπορούμε να χρησιμοποιήσουμε μια συνάρτηση lambda σαν όρισμα της mapcar:

- (mapcar # '(lambda (x) (if (numberp x) x)) '(1 a b 2)) → (1 NIL NIL 2)

Εδώ, η συνάρτηση εφαρμογής ορίζεται επί τόπου και ο ορισμός της χάνεται μετά την εκτέλεση της πρότασης. Συναρτήσεις lambda μπορούν να χρησιμοποιηθούν σε όλες τις συναρτήσεις που ακολουθούν.

remove-if-not

Είναι μια συνάρτηση φίλτρου μιας λίστας. Έχει την ακόλουθη σύνταξη:

(remove-if-not #'<function-name> <list form>)

όπου το <function-name> είναι το όνομα μιας συνάρτησης ελέγχου/φίλτρου και το <list form> πρέπει νάχει σαν αποτέλεσμα μια λίστα. Το αποτέλεσμα είναι η λίστα χωρίς στοιχεία που δεν ικανοποιούν τη συνάρτηση ή με άλλα λόγια η λίστα με τα στοιχεία που ικανοποιούν τη συνάρτηση. Π.χ.

* (remove-if-not #'oddp '(1 2 3 4 5)) → (1 3 5)

remove-if

Είναι η συμμετρική της προηγούμενης. Έχει την ακόλουθη σύνταξη:

(remove-if #'<function-name> <list form>)

Το αποτέλεσμα είναι η λίστα χωρίς στοιχεία που ικανοποιούν τη συνάρτηση ή με άλλα λόγια η λίστα με τα στοιχεία που δεν ικανοποιούν τη συνάρτηση. Π.χ.

* (remove-if #'oddp '(1 2 3 4 5)) → (2 4)

find-if

Συνάρτηση αναζήτησης. Σύνταξη:

(find-if #'<function-name> <list form>)

Επιστρέφει το πρώτο στοιχείο της λίστας που ικανοποιεί τη συνάρτηση. Π.χ.

* (find-if #'evenp '(1 2 3 4 5)) → 2

count-if

Συνάρτηση αναζήτησης-απαρίθμησης. Σύνταξη:

(count-if #'<function-name> <list form>)

Επιστρέφει τον αριθμό των στοιχείων της λίστας που ικανοποιούν τη συνάρτηση. Π.χ.

* (count-if #'symbolp '(1 a 3 b 5)) → 2

funcall

Χρησιμοποιείται συνήθως όταν θέλουμε να καλέσουμε μια συνάρτηση που έχει καταχωρηθεί σαν τιμή σε μια μεταβλητή. Επίσης, χρησιμοποιείται για τον ορισμό συναρτήσεων που έχουν για ορίσματα συναρτήσεις. Σύνταξη:

(funcall #'<function-name> <arg1> <arg2> … <argn>)

Εφαρμόζει τη συνάρτηση στα ορίσματα που ακολουθούν. Οπότε, τα ορίσματα πρέπει να είναι τόσα τον αριθμό όσα απαιτούνται από τη συνάρτηση. Π.χ.

* (funcall #'list 'a 'b 'c) → (A B C), δηλαδή είναι το ίδιο σα να είχαμε

* (list 'a 'b 'c)

Ας δούμε τώρα πως μπορούμε να καλέσουμε μια συνάρτηση που έχει καταχωρηθεί σε μια μεταβλητή:

* (setf x 'list) → LIST
* (funcall x 'a 'b 'c) → (A B C)

Τέλος, ας δούμε πως μπορούμε να ορίσουμε συναρτήσεις με ορίσματα συναρτήσεις. Π.χ.

```
(defun test-fun (argx funx)
  (funcall funx argx))
```

Τώρα μπορούμε να καλέσουμε τη συνάρτηση:

* (test-fun ‘(1 a b 2) #’last) → (2)
* (test-fun ‘(1 a b 2) #’first) → 1

apply

Είναι αντίστοιχη της funcall με κάπως διαφορετική σύνταξη:

(apply #’<function-name> ‘(<arg1> <arg2> … <argn>))

όπου δηλ. τα ορίσματα είναι σε μια λίστα. Εφαρμόζει κι’ αυτή τη συνάρτηση στα ορίσματα. Π.χ.

* (apply #’list ‘(a b c)) → (A B C)

Επίσης, αντίστοιχα και στην κλήση συνάρτησης μέσω μεταβλητής και στη δημιουργία συνάρτησης με ορίσματα συναρτήσεις:

* (funcall x ‘(a b c)) → (A B C)

```
(defun test-fun (argx funx)
  (apply funx (list argx)))
```

9. ΛΙΣΤΕΣ ΙΔΙΟΤΗΤΩΝ-ΠΙΝΑΚΕΣ

9.1 Λίστα ιδιοτήτων

Η LISP επιτρέπει σε ένα σύμβολο να έχει *τιμές ιδιοτήτων* (property values). Δηλαδή, μπορούμε σ' ένα σύμβολο να προσαρτήσουμε *ιδιοτήτες* (properties) και να αποδώσουμε σ' αυτές τιμές. Το σύνολο των ιδιοτήτων και των τιμών τους που είναι προσαρτημένο σε ένα σύμβολο λέγεται *λίστα ιδιοτήτων* (property list) του συμβόλου. Π.χ. μπορούμε να θεωρήσουμε ότι ένα σύμβολο παριστάνει το όνομα ενός φοιτητή που έχει σαν ιδιότητες τις 'τμήμα', 'γονείς', 'αδέλφια', που έχουν κάποιες τιμές.

Ο τρόπος για να δημιουργήσουμε ένα σύμβολο με λίστα ιδιοτήτων είναι ο ακόλουθος:

(setf (get <symbol> <property name>) <property value>)

Π.χ.

* (setf (get 'giannis 'department) 'computer-engineering) →
COMPUTER-ENGINEERING
* (setf (get 'giannis 'parents) '(kostas maria)) → (KOSTAS MARIA)
* (setf (get 'giannis 'siblings) '(giorgos eleni)) → (GIORGOS ELENI)

Ο τρόπος να προσπελάσουμε τις τιμές των ιδιοτήτων ενός τέτοιου συμβόλου είναι ο εξής:

(get <symbol> <property name>)

Π.χ.

* (get ‘giannis ‘department) → COMPUTER-ENGINEERING
* (get ‘giannis ‘parents) → (KOSTAS MARIA)

Αν κάποια ιδιότητα δεν υπάρχει, τότε επιστρέφεται NIL. Έτσι, δεν μπορούμε να διακρίνουμε μεταξύ μιας ιδιότητας που δεν υπάρχει και μιας που έχει τιμή NIL.

Για να διαγράψουμε μια ιδιότητα και την αντίστοιχη τιμή της, χρησιμοποιούμε το remprop ως εξής:

(remprop <symbol> <property>)

Π.χ.

* (remprop ‘giannis ‘parents) → T

Οπότε

* (get ‘giannis ‘parents) → NIL

9.2 Πίνακες

Η δημιουργία ενός μονοδιάστατου πίνακα στη LISP γίνεται ως εξής:
(setf <array name> (make-array <dimension>))

Π.χ. η

* (setf students (make-array 4)) → #(0 0 0 0) (αυτός είναι ο τρόπος που η LISP παρουσιάζει ένα πίνακα στην οθόνη)

δημιουργεί ένα μονοδιάστατο πίνακα με όνομα students που έχει 4 στοιχεία. Τα στοιχεία ενός πίνακα στη LISP αριθμούνται από το 0 (εδώ από 0-3). Αν δεν αρχικοποιήσουμε ένα πίνακα, τότε δίνονται μηδενικά στοιχεία (βλ. παραπάνω αποτέλεσμα). Επίσης, πρέπει να σημειώσουμε ότι τα στοιχεία ενός πίνακα στη LISP μπορεί να είναι οτιδήποτε, δηλ. και ετερογενή. Δεν απαιτείται όπως σε άλλες γλώσσες να είναι κοινού τύπου.

Η αρχικοποίηση ενός πίνακα μπορεί να γίνει με διάφορους τρόπους.

(α) Με τη χρήση της παραμέτρου :initial-element κατά τη δημιουργία του πίνακα.
Π.χ.

* (setf students (make-array 4 :initial-element NIL)) → #(NIL NIL NIL NIL) ή
* (setf students (make-array 4 :initial-element ‘a)) → #(A A A A)

(β) Με τη χρήση της παραμέτρου :initial-contents κατά τη δημιουργία του πίνακα.
Π.χ.

* (setf students (make-array 4 :initial-contents ‘(a b c d))) → #(A B C D)

(γ) Με την ανάθεση τιμών σε κάθε στοιχείο του μέσω της aref.
Π.χ.

* (setf (aref students 0) ‘a) → A
* (setf (aref students 1) ‘b) → B κλπ.

Η προσπέλαση των στοιχείων ενός πίνακα γίνεται ως εξής:

(aref <array name> <element index>)

Π.χ.

* (aref students 1) → B (αναφερόμενοι στην τελευταία αρχικοποίηση)

Ο ορισμός δισδιάστατου πίνακα γίνεται κατά αντίστοιχο τρόπο:

(setf <array name> (make-array <dimensions>))

Π.χ. η

* (setf marks (make-array '(2 3))) → #2A((0 0 0) (0 0 0))

δημιουργεί ένα δισδιάστατο πίνακα με όνομα marks που έχει 2x3=6 στοιχεία.

Η αρχικοποίηση ενός δισδιάστατου πίνακα μπορεί να γίνει με τους ίδιους ακριβώς τρόπους, όπως και ενός μονοδιάστατου. Π.χ.

* (setf marks (make-array '(2 3) :initial-element NIL)) →
#2A((NIL NIL NIL) (NIL NIL NIL))

* (setf marks (make-array '(2 3) :initial-contents '((7 6 8) (9 5 6)))) →
#2A((7 6 8) (9 5 6))

* (setf (aref marks 0 0) 7) → A (ανάθεση τιμής στο στοιχείο (0,0))
* (setf (aref marks 0 1) 6) → B (ανάθεση τιμής στο στοιχείο (0,1)) κλπ.

Η προσπέλαση των στοιχείων ενός δισδιάστατου πίνακα γίνεται παρόμοια με αυτή του μονοδιάστατου:

(aref <array name> <element index>)

Π.χ.

* (aref marks 1 2) → 6 (αναφερόμενοι στην τελευταία αρχικοποίηση)

Δύο συναρτήσεις που συνδέονται με τους πίνακες είναι οι array-dimension και array-dimensions, που επιστρέφουν την(τις) διασταση(εις) ενός πίνακα.

array-dimension

(array-dimension <array name> <axis num>)

όπου το <axis num> είναι ‘0’ για την πρώτη (ή μια) διάσταση, ‘1’ για τη δεύτερη κ.ο.κ. διάσταση. Το αποτέλεσμα είναι το μέγεθος της συγκεκριμένης διάστασης. Π.χ.

* (array-dimension students 0) → 4

* (array-dimension marks 0) → 2
* (array-dimension marks 1) → 3

array-dimensions

(array-dimensions <array name>)

όπου το αποτέλεσμα είναι μια λίστα με τα μεγέθη των διαστάσεων του πίνακα. Π.χ.

* (array-dimensions students) → (4)

* (array-dimensions marks) → (2 3)

10. ΔΟΜΕΣ ΣΤΗ LISP

Η LISP επιτρέπει τη δημιουργία νέων τύπων δεδομένων με τη μορφή των λεγόμενων δομών (structures). Μια δομή αποτελείται από πεδία (fields) και τιμές (values) των πεδίων. Η δημιουργία μιας δομής γίνεται με τη βοήθεια του defstruct ως εξής:

(defstruct <structure name>
    (<field1> <value1)
    (<field2> <value2)

    (<fieldn> <valuen))

Π.χ. η

(defstruct student
    (year nil)
    (sex nil)
    (performance nil))

δημιουργεί τη δομή student με πεδία τα ‘name’, ‘sex’ και ‘performance’ και αρχική (ή εξ’ορισμού) τιμή γι’ αυτά nil.

Η defstruct δημιουργεί μια δομή, δηλ. ένα καλούπι, αλλά όχι στιγμιότυπα της δομής. Δημιουργεί όμως μαζί με τη δομή και μια *συνάρτηση-δημιουργό* στιγμιοτύπων. Αυτή η συνάρτηση έχει όνομα make-<structure name>, δηλ. για την παραπάνω δομή είναι η make-student. Μ’ αυτή μπορούμε να δημιουργήσουμε στιγμιότυπα της δομής student. Π.χ. η

* (setf maria (make-student)) →
#S(STUDENT :YEAR NIL :SEX NIL :PERFORMANCE NIL)

δημιουργεί ένα στιγμιότυπο με όνομα MARIA με τιμές πεδίων τις αρχικές (εξ'ορισμού) τιμές, ενώ η

* (setf giannis (make-student :year 'b :sex 'male :performance 'good)) →
#S(STUDENT :YEAR B :SEX MALE :PERFORMANCE GOOD)

δημιουργεί ένα στιγμιότυπο με όνομα 'giannis' και με τιμές 'b', 'male' και 'good' στα τρία πεδία αντίστοιχα.

Επίσης, η defstruct δημιουργεί και *συναρτήσεις ανάγνωσης*, για την προσπέλαση των τιμών των πεδίων. Αυτές έχουν σαν όνομα το <structure name>-<field name>, δηλ. στην περίπτωσή μας θα είναι οι student-year, student-sex και student-performance. Τώρα

* (student-sex maria) → NIL
* (student-year giannis) → B

Επί πλέον, γενικεύεται η χρήση της setf ώστε να λειτουργεί και για τα πεδία των στιγμιοτύπων. Έτσι, μπορούμε να εισάγουμε ή να αλλάζουμε τις τιμές των πεδίων. Π.χ.

* (setf (student-year maria) 'a) → A
* (setf (student-sex maria) 'female) → FEMALE
* (setf (student-performance maria) 'very-good) → VERY-GOOD

Επιπρόσθετα, η defstruct δημιουργεί και μια *συνάρτηση αναγνώρισης τύπου* με όνομα <structure name>-p, οπότε μπορούμε να ελέγξουμε αν ένα αντικείμενο είναι του τύπου της δομής που δημιουργήθηκε. Π.χ.

* (student-p maria) → T

Είναι δυνατόν να δημιουργήσουμε δομές που να περιέχουν άλλες δομές, δηλ. να υλοποιήσουμε σχέσεις εξειδίκευσης μεταξύ δομών. Π.χ. έστω η δομή student, όπως την ορίσαμε παραπάνω. Τότε μπορούμε να ορίσουμε μια νέα δομή univ-student, ως εξής:

```
* (defstruct (univ-student (:include student))
    (institution 'university))
```

Τώρα η δομή univ-student έχει τα πεδία που έχει η student συν το πεδίο 'institution'. Μπορούμε να δημιουργήσουμε ένα στιγμιότυπο της univ-student:

```
* (setf giorgos (make-univ-student)) →
#S(UNIV-STUDENT :YEAR NIL :SEX NIL :PERFORMANCE NIL
:INSTITUTION UNIVERSITY)
```

Οπότε

```
* (univ-student-year giorgos) → NIL
* (univ-student-institution giorgos) → UNIVERSITY
```

Μπορούμε όμως να δώσουμε και τιμές στα πεδία όπως και στη δομή student:

```
* (setf (univ-student-year giorgos) 'c) → C
```

οπότε

```
* (univ-student-year giorgos) → C
```

Τέλος, η συνάρτηση describe τα περιεχόμενα ενός στιγμιότυπου, για λόγους ελέγχου. Π.χ.

```
* (describe maria) →
```

#S(STUDENT :YEAR A :SEX FEMALE :PERFORMANCE VERY-GOOD) is a named structure of type STUDENT.

It is included in the structures:

UNIV-STUDENT

Its slot names and values are:

YEAR - A

SEX - FEMALE

PERFORMANCE - VERY-GOOD

Βιβλιογραφία

1. G. L. Steele JR, “Common Lisp, The Language”, Digital Press, 1984 (διαθέσιμο on-line στη διεύθυνση http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html)
2. P. H. Winston and B. K. P. Horn, LISP, 3rd Edition, Addison Wesley, 1989.
3. G. F. Luger and W. A. Stubblefield, Artificial Intelligence and the Design of Expert Systems (Κεφ. 7), Benjamin/Cummings, 1989.