Κεφάλαιο 5

Εργαστηριακή Άσκηση 5

Όπου θα εξηγήσουμε πώς μπορεί να γίνει εφικτή η επαναληπτική εκτέλεση μιας ενέργειας και πώς μια ενέργεια μπορεί να εφαρμοσθεί σε κάθε στοιχείο μιας λίστας στοιχείων.

5.1 Εντολές Επανάληψης

Η επανάληψη (iteration) μιας ενέργειας για κάποιο αριθμό φορών, μπορεί να γίνει με χρήση του DOTIMES primitive, ενώ η επανάληψη με την έννοια της εφαρμογής της ίδιας ενέργειας σε πολλά στοιχεία μιας λίστας, μπορεί να πραγματοποιηθεί μέσω του DOLIST. Τέλος, η Lisp παρέχει και τις πιο ισχυρές συναρτήσεις DO και DO* που καλύπτουν τις αδυναμίες των DOTIMES και DOLIST, καθώς και δύο συναρτήσεις για ακολουθιακή αποτίμηση πολλαπλών τύπων, τις PROG1 και PROGN.

5.1.1 DOTIMES

Η DOTIMES χρησιμοποιείται ευρέως, καθώς παρέχει έναν εύκολο τρόπο για να γράφουμε διαδικασίες επανάληψης χρησιμοποιώντας μετρητή. Ο γενικός ορισμός έχει ως εξής:

`(dotimes (`*`counter upper_bound result`*`) `*`body`*`)` Αρχικά αποτιμάει τον τύπο *upper_bound* και παράγει έναν αριθμό, έστω n. Εν συνεχεία, οι αριθμοί $0 \ldots n-1$ ανατίθενται ο ένας μετά τον άλλο στον *counter*. Για κάθε τιμή εκτελείται ο τύπος *body*, δηλαδή n φορές. Το αποτέλεσμα επιστρέφεται μέσω του τύπου *result*. Αν δεν έχει καθοριστεί τύπος *result*, η dotimes επιστρέφει NIL και η χρησιμότητά της έγγυται στα side effects που πιθανώς να αφήνει.

Για παράδειγμα, τρέξτε τον ακόλουθο τύπο (μη δίνετε ιδιαίτερη σημασία στον τύπο format t, καθώς χρησιμοποιείται για output και θα συζητηθεί σε επόμενο εργαστήριο):

```
(dotimes (i 4)
  (format t "~&I is ~S." i))
```

Ας δούμε ένα πιο εξεζητημένο παράδειγμα χρήσης της dotimes, το οποίο θα σας βοηθήσει να κατανοήσετε τα προαναφερθέντα. Έστω, ότι θέλουμε να ορίσουμε μία

συνάρτηση η οποία θα υπολογίζει δυνάμεις αριθμών. Με λίγα λόγια θα υπολογίζει τύπους της μορφής

$$m^n = m \times m \times \ldots \times m \tag{5.1}$$

Βέβαια, αυτό μπορεί να γίνει πολύ εύκολα και αναδρομικά, αλλά εμείς θα προτιμήσουμε την επαναληπτική εκτέλεση στην παρούσα φάση. Η ακόλουθη συνάρτηση υπολογίζει δυνάμεις αριθμών χρησιμοποιώντας τη `dotimes`:

```
(defun dotimes-expt (m n)
  (let ((result 1))
    (dotimes (count n result)
      (setf result (* m result)))))
```

Η συνάρτηση αυτή παίρνει τα ορίσματα `m` και `n`, αρχικοποιεί μια τοπική μεταβλητή `result` στο 1, για n φορές εκτελεί $result \leftarrow m \times result$ και επιστρέφει την τιμή του `result`.

- Να ορίσετε τη συνάρτηση `dotimes-factorial`, χρησιμοποιώντας ένα `dotimes` τύπο, η οποία θα υπολογίζει το παραγοντικό του n. (*Υπόδειξη*: Θυμηθείτε ότι το παραγοντικό του n είναι 1, αν $n = 0$ και $n * (n-1)!$, αλλιώς· να την υλοποιήσετε τόσο αναδρομικά όσο και χωρίς χρήση αναδρομής).

5.1.2 DOLIST

Η `DOLIST` είναι παρόμοια με τη `DOTIMES` με τη διαφορά ότι αντί να αναθέτει n τιμές σε ένα μετρητή, αναθέτει σε ένα στοιχείο/παράμετρο το ένα μετά το άλλο τα στοιχεία μιας λίστας. Επομένως, καταλαβαίνουμε ότι είναι ιδιαίτερα χρήσιμη όταν θέλουμε να εφαρμόσουμε κάτι σε κάθε στοιχείο μιας λίστας. Η γενική σύνταξη έχει ως εξής:

`(dolist (`*`element a_list result`*`)` *`body`*`)` Αρχικά αποτιμάται η λίστα, παράγοντας μία λίστα στοιχείων. Εν συνεχεία, τα στοιχεία ανατίθενται το ένα μετά το άλλο στο *element*. Για κάθε στοιχείο αποτιμάται το σώμα της `dolist` και όταν τα στοιχεία εξαντληθούν επιστρέφεται το *result*. Εάν το *result* δεν έχει καθοριστεί, επιστρέφεται `NIL`.

Ας κατασκευάσουμε μία συνάρτηση για να δείξουμε τα παραπάνω. Η συνάρτηση αυτή, έστω `count-outlyers`, θα παίρνει ως όρισμα μία λίστα με θερμοκρασίες και θα μετράει πόσες απ' αυτές είναι κάτω απ' το σημείο πήξης του νερού ($0\,^\circ\mathrm{C}$) ή πάνω απ' το σημείο βρασμού του ($100\,^\circ\mathrm{C}$).

```
(defun count-outlyers (list-of-elements)
  (let ((result 0))
    (dolist (element list-of-elements result)
      (when (or (> element 100)
                (< element 0))
        (setf result (+ result 1))))))
```

Η συνάρτηση `when`, που χρησιμοποιήσαμε εκεί που μάλλον θα περιμένατε να δείτε την `if`, μπορεί όντως να χρησιμοποιηθεί στη θέση της, αν γνωρίζουμε εκ των προτέρων ότι το `else form` θα είναι `NIL`. Αν κοιτάξετε προσεκτικά το παράδειγμα, θα παρατηρήσετε ότι στην πραγματικότητα δε θέλουμε να γίνει τίποτα όταν δεν ισχύει καμία απ' τις συνθήκες ελέγχου (`or forms`), καθώς επιθυμούμε το `result`

να παραμείνει ανεπηρέαστο. Έτσι μπορούμε να χρησιμοποιήσουμε τη `when` που θεωρεί το `else form` εκ των προτέρων `NIL`. Υπάρχει και η συνάρτηση `unless` η οποία θεωρεί εκ των προτέρων `NIL` το `then form` της αντίστοιχης `if`, επομένως, καταλαβαίνουμε ότι εκτελεί κάτι όταν η συνθήκη που της δώσουμε είναι ψευδής. Ας δούμε τη σύγκριση με την `if`, για να απλοποιήσουμε τα πράγματα:

```
(if test then_form nil)
          ≡
(when test then_form)
```

και

```
(if test nil else_form)
          ≡
(unless test else_form)
```

Επίσης, αξίζει να αναφέρουμε, ότι μεταξύ του `test` και του τερματικού τύπου στις `when` και `unless`, μπορεί να υπάρχει οποιοσδήποτε αριθμός τύπων, οι οποίοι αποτιμώνται μόνο για τα side effects τους.

1. Να τροποποιήσετε την `count-outlyers`, έτσι ώστε να κάνει τα ίδια με χρήση της `unless` αντί της `when`.

2. Να ορίσετε μία συνάρτηση, ονόματι `increase-numbers`, η οποία θα παίρνει ως όρισμα μία λίστα και έναν αριθμό n και για κάθε στοιχείο της που είναι αριθμός θα του προσθέτει ένα τυχαίο αριθμό από $-n$ μέχρι n. Θα πρέπει να επιστρέφει το συνολικό άθροισμα των αριθμών μετά την τροποποίησή τους, διαιρεμένο δια το πλήθος τους. Η συνάρτηση `increase-numbers` θα πρέπει να καλείται από μια άλλη συνάρτηση, ονόματι `check-result`, η οποία θα ελέγχει εάν το αποτέλεσμα είναι θετικός αριθμός, αρνητικός ή μηδέν και θα επιστρέφει ανάλογο μήνυμα. Μην ξεχάσετε να ελέγχετε για πιθανή διαίρεση με το 0, την οποία, προφανώς, θα πρέπει να χειρίζεστε ξεχωριστά.

Ας δούμε τώρα τη `first-n-outlyers`, για την οποία, αν δεν είναι προφανές απ' το όνομα, θα πρέπει να καταλάβετε τί κάνει. Σε κάθε περίπτωση θα πρέπει να προσπαθήσετε να καταλάβετε πώς κάνει αυτό που κάνει. Πριν την τρέξετε, να μην ξεχάσετε να αναθέσετε σε δύο global μεταβλητές `boiling` και `freezing`, τις τιμές βρασμού και πήξης του νερού, αντίστοιχα.

```
(defun first-n-outlyers (n list-of-elements)
  (let ((result 0)
        (outlyers nil))
    (if (<= n 0)
        nil
        (dolist (element list-of-elements outlyers)
          (cond ((or (> element boiling)
                     (< element freezing))
                 (setf result (+ result 1))
                 (push element outlyers)
                 (when (= n result) (return outlyers))))))))
```

Ας δούμε άλλο ένα ενδιαφέρον παράδειγμα:

- Ένας γράφος $G(V, E)$ είναι πλήρης, εάν $\forall x, y \in V(G) : x \neq y \Rightarrow xy \in E(G)$, το οποίο σημαίνει ότι για οποιουσδήποτε διακριτούς κόμβους x και y του γράφου G, υπάρχει η αντίστοιχη ακμή που τους συνδέει στο σύνολο των ακμών του G, ή με άλλα λόγια, όλοι οι κόμβοι είναι γειτονικοί μεταξύ τους. Η ακόλουθη συνάρτηση ελέγχει αν ένας γράφος είναι πλήρης ή όχι, παίρνοντας ως είσοδο τα σύνολα των κόμβων και των ακμών του, V και E, αντίστοιχα. Βέβαια, για να απλοποιήσουμε τα πράγματα υποθέτουμε ότι η είσοδος είναι ένας νόμιμος γράφος, καθώς αποφεύγουμε οποιονδήποτε έλεγχο για κάτι τέτοιο.

```
(defun check-completeness (V E)
  (let ((result1 0)
        (result2 0))
        (if (equal nil
          (dolist (current-node V result1)
            (if (<
                 (dolist (current-edge E result2)
                   (when (member current-node current-edge)
                         (setf result2 (+ result2 1))))
                 (- (length V) 1))
                 (return (setf result1 nil))
                 (setf result2 0))))
           'not-complete 'complete)))
```

Αφού μελετήσετε προσεκτικά και τρέξετε για αρκετές εισόδους τη συνάρτηση αυτή, προσπαθήστε να τη διασπάσετε σε όλα τα επιμέρους στοιχεία της και να δημιουργήσετε πολλές διαφορετικές συναρτήσεις, που καλώντας η μία την άλλη θα ελέγχουν και πάλι αν η αρχική είσοδος αντιπροσώπευε ένα πλήρη γράφο. Με άλλα λόγια προσπαθήστε να κάνετε το πρόγραμμα πιο κομψό, καθώς η παραπάνω συνάρτηση δεν είναι ούτε *portable* ούτε ευανάγνωστη. Να θυμάστε ότι όσο πιο στοιχειώδεις συναρτήσεις δημιουργούμε, τόσο πιο συχνά θα μπορούμε να τις χρησιμοποιήσουμε ως επιμέρους τμήματα μιας υψηλότερου επιπέδου λειτουργίας.

- Αγνοώντας, τέλος, την ύπαρξη της `reverse` να γράψετε μία συνάρτηση, ονόματι `dolist-reverse`, η οποία θα αντιστρέφει τα στοιχεία μιας λίστας χρησιμοποιώντας τη `dolist`.

5.1.3 DO και DO*

Η συνάρτηση `DO` μπορεί να χρησιμοποιηθεί για επαναληπτική εκτέλεση στις περιπτώσεις όπου οι `DOTIMES` και `DOLIST` δεν είναι αρκετά ευέλικτες. Σε άλλες περιπτώσεις θα πρέπει να αποφεύγεται, καθώς δεν είναι εύκολη στη χρήση.

- Ας δούμε ένα παράδειγμα που θα διευκολύνει τη συζήτηση σχετικά με τη `do`.

```
(defun do-expt (m n)
  (do ((result 1)
       (exponent n))
      ((zerop exponent) result)
```

```
        (setf result (* m result))
        (setf exponent (- exponent 1))))
```

Όπως είναι προφανές, πρόκειται για μία ακόμα εκδοχή του m^n. Παρατηρούμε ότι το πρώτο τμήμα αποτελείται από μία λίστα παραμέτρων (τοπικών μεταβλητών), οι οποίες αρχικοποιούνται σε κάποια τιμή. Συγκεκριμένα, δηλώνονται οι μεταβλητές `result` και `exponent` και αρχικοποιούνται στις τιμές 1 και `n`, αντίστοιχα. Εάν κάποια απ' αυτές τις μεταβλητές είχε ήδη δηλωθεί και είχε κάποια τιμή πριν από την κλήση της `do`, τότε σε αυτή, κατά την έξοδο, ανατίθεται η παλιά της τιμή. Εν συνεχεία, μέσω μιας λίστας, δηλώνονται η συνθήκη τερματισμού και η επιστρεφόμενη τιμή. Εδώ, η συνθήκη τερματισμού είναι, να έχει τιμή μηδέν η μεταβλητή `exponent` και η επιστρεφόμενη τιμή είναι το `result`. Μεταξύ τους θα μπορούσαμε να έχουμε τοποθετήσει οποιοδήποτε πλήθος τύπων, οι οποίοι θα αποτιμούνταν μόνο για τα side effects τους. Επιπρόσθετα, παρατηρούμε ότι το σώμα της `do` αποτελείται από δύο ανεξάρτητες λίστες οι οποίες αποτιμούνται ακολουθιακά. Επειδή ακριβώς αποτιμούνται ακολουθιακά, οι τιμές που επιστρέφουν αγνοούνται και η αποτίμηση γίνεται καθαρά και μόνο για τα side effects τους. Τα side effects εδώ είναι η ανάθεση στη `result` του γινομένου του εαυτού της επί `m` και η μείωση κατά 1 της `exponent`. Τέλος, αξίζει να σημειώσουμε ότι όταν συναντηθεί μία `return` στο σώμα της `do`, η `do` τερματίζει αμέσως τη λειτουργία της και επιστρέφει την τιμή της `return`. Φυσικά στην πλειονότητα των περιπτώσεων η `return` θα πρέπει να χρησιμοποιείται σε συνδυασμό με μια συνθήκη ελέγχου.

- Αφήστε κενή τη λίστα του δεύτερου μέρους της `do` (δηλαδή, τη `((zerop exponent) result)`) και προσπαθήστε με μία `return` στο σώμα της `do` να επιτύχετε το ίδιο ακριβώς αποτέλεσμα, δηλαδή, όταν η μεταβλητή `exponent` γίνει 0, η `do` να τερματίζει και να επιστρέφεται το `result`.

- Αφού εισάγετε και πάλι την προηγούμενη εκδοχή της `do-expt`, να αντικαταστήσετε τη δήλωση `(result 1)` με τη `(result 1 (* m result))`, την `(exponent n)` με την `(exponent n (- exponent 1))` και να διαγράψετε και τους δύο `setf` τύπους από το σώμα της `do`. Τί παρατηρείτε;

Στην πραγματικότητα κάναμε το ίδιο ακριβώς πράγμα, μόνο που το κάναμε μέσω *update* τύπων κατά τη δήλωση των μεταβλητών. Ας πάρουμε για παράδειγμα τον τύπο `(result 1 (* m result))`. Αυτός λέει στη `do` να κάνει 1 την τιμή της `result` στην πρώτη επανάληψη και `(* m result)` σε κάθε άλλη επανάληψη. Αντίστοιχα και για την `exponent`. Αυτό που δε θα πρέπει να ξεχνάτε, είναι ότι η `do` αποτιμάει τη λίστα των παραμέτρων της παράλληλα. Επομένως, δε μπορούμε να αρχικοποιήσουμε μια μεταβλητή και εν συνεχεία η τιμή μιας άλλης μεταβλητής να εξαρτάται απ' την πρώτη. Τρέξτε για παράδειγμα την ακόλουθη συνάρτηση και διαβάστε προσεκτικά το μήνυμα σφάλματος που θα πάρετε:

```
(defun do-expt (m n)
  (do ((result m (* m result))
       (exponent n (- exponent 1))
       (counter (- exponent 1)
                (- exponent 1)))
      ((zerop counter) result)))
```

Το πρόβλημα λύνεται αν χρησιμοποιήσουμε τη `do`*, η οποία διαφέρει από τη `do` μόνο στο ότι αποτιμάει τις παραμέτρους της ακολουθιακά.

- Χρησιμοποιήστε τη do* αντί της do στο προηγούμενο παράδειγμα. Διορθώθηκε το πρόβλημα; Αν ναι, μπορείτε να καταλάβετε γιατί;

5.1.4 PROG1 και PROGN

Στο σημείο αυτό καλό θα ήταν να ξεκαθαρίσουμε κάτι. Αρκετές συναρτήσεις μπορούν έμμεσα να αποτιμούν ακολουθιακά πολλούς τύπους. Μεταξύ αυτών είναι η defun, η let, η let*, η when και η unless, οι οποίες επιστρέφουν την τιμή του τελευταίου τους τύπου, αλλά μπορούν να αποτιμήσουν για τα side effects τους απεριόριστο αριθμό τύπων. Δοκιμάστε για παράδειγμα τη συνάρτηση:

```
(defun new-prog (a b)
  (let ((c 1))
    (setf c (+ a b))
    (* c 2)))
```

Παρατηρούμε ότι το σώμα της let αποτελείται από δύο τύπους οι οποίοι αποτιμούνται ακολουθιακά. Ο πρώτος έχει σημασία μόνο για το side effect του, ενώ ο δεύτερος είναι αυτός που επιστρέφει τιμή.

Εάν θέλουμε να συνδυάσουμε τύπους άμεσα, μπορούμε να το κάνουμε με τη βοήθεια των συναρτήσεων PROG1 και PROGN. Και οι δύο αποτιμούν ακολουθιακά όλους τους τύπους απ' τους οποίους αποτελούνται και επιστρέφουν την τιμή ενός απ' αυτούς. Η prog1 επιστρέφει την τιμή του πρώτου τύπου, ενώ η progn του τελευταίου, εξ ου και τα ονόματά τους.

- Να αποτιμήσετε τους ακόλουθους τύπους:

  1. `(progn (setf a 1) (setf b 2) (setf c (+ a b)))`
  2. `(prog1 (setf a 1) (setf b 2) (setf c (+ a b)))`

Βέβαια, επειδή ακριβώς οι περισσότερες συναρτήσεις επιτρέπουν την ακολουθιακή αποτίμηση πολλών τύπων, η χρήση των prog1 και progn δεν είναι τόσο συχνή. Μία πολύ χρήσιμη εφαρμογή τους είναι σε συνδυασμό με το if conditional. Ανάλογα με το αν αληθεύει ή όχι η συνθήκη της if, μπορούμε να αποτιμήσουμε αρκετούς τύπους και όχι μόνο έναν, αν χρησιμοποιήσουμε μία εκ των prog1 και progn. Για παράδειγμα:

```
(defun new-prog2 (n)
  (let ((a 0) (b 0))
    (if (> n 0)
        (progn (setf a n) (setf b (- 0 n)))
        (prog1 (setf a (- 0 n)) (setf b n)))))
```

Ακολουθεί μία ανακεφαλαιωτική άσκηση:

- Να ορίσετε μία συνάρτηση η οποία θα παίρνει ως όρισμα μία λίστα και θα ελέγχει αν το πλήθος των στοιχείων της είναι άρτιος αριθμός. Τον έλεγχο θα τον αναλαμβάνει μία άλλη συνάρτηση η οποία θα πρέπει να χρησιμοποιεί τη dolist. Εάν το πλήθος είναι άρτιο, από το then τύπο της if θα καλείται μία άλλη συνάρτηση που θα χωρίζει στη μέση τα στοιχεία της λίστας και θα δημιουργεί μία νέα λίστα της μορφής `((...) (...))` και η λίστα αυτή θα αποθηκεύεται σε μία μεταβλητή η οποία και θα επιστρέφεται. Έπειτα, στο

`then` τύπο της `if`[1], της βασικής συνάρτησης του προγράμματός σας, οι δύο εσωτερικές λίστες θα αποθηκεύονται η μία μετά την άλλη σε δύο μεταβλητές (χρησιμοποιήστε την `progn` στην `if`) και οι μεταβλητές αυτές θα περνούν μία μία ως είσοδος σε μία άλλη συνάρτηση, που θα επιστρέφει με χρήση της `do` το άθροισμα των στοιχείων της λίστας που παίρνει ως όρισμα. Εν συνεχεία, και πάλι στην `progn` του `then` τύπου της `if`, θα πρέπει να ελέγχεται ποιό είναι το μεγαλύτερο από τα δύο αθροίσματα και με ένα τελευταίο τύπο αυτό να επιστρέφεται. Αν τώρα το πλήθος είναι περιττό, θα πρέπει να γίνονται ακριβώς τα ίδια, αφού πρώτα αφαιρεθεί ένα τυχαίο στοιχείο της λίστας.

[1] Εάν έχετε χρησιμοποιήσει την `if` για το βασικό έλεγχο στη συνάρτησή σας.