6

Εργαστηριακή Άσκηση 6

Όπου θα ασχοληθούμε, όπως γίνεται πάντα στις γλώσσες προγραμματισμού, με αυτό που συνηθίζουμε να ονομάζουμε Input/Output.

6.1 Input/Output

Ως γνωστόν, το ''i/o'' είναι ο τρόπος με τον οποίο ο υπολογιστής και πιο συγκεκριμένα ένα πρόγραμμα, επικοινωνεί με τον ''έξω κόσμο''. Η Lisp από μόνη της παρέχει μία πολύ απλή μορφή i/o, καθώς όπως έχουμε ήδη δει εκτελεί συνεχώς ένα read-eval-print loop, το οποίο εκτυπώνει στην οθόνη τα αποτελέσματα των εκφράσεων που αποτιμά. Παρ' όλα αυτά, συνήθως, ως απαιτητικοί προγραμματιστές που είμαστε, ζητάμε αρκετά περισσότερα απ' αυτό και θα θεωρούσαμε, χωρίς δεύτερη σκέψη, άχρηστη μια γλώσσα που δε θα μπορούσε να μας τα παρέχει. Ακόμα χειρότερα, κάποιοι από εμάς θα αναγκάζονταν να δημιουργήσουν βιβλιοθήκες με τις απαραίτητες i/o συναρτήσεις για να μπορούμε να κάνουμε σωστά τη δουλειά μας. Επειδή σε κάποιους αυτά θα ακούγονται ως τρομακτικά σενάρια, αρκεί να πούμε ότι κάποιοι προνόησαν για τα προφανή και ως εκ τούτου οι απαραίτητες i/o συναρτήσεις είναι ήδη υλοποιημένες και εσείς το μόνο που έχετε να κάνετε είναι να μάθετε να τις χειρίζεστε και να τις καλείτε όταν τις χρειάζεστε. Έτσι, με τη βοήθειά τους, μπορείτε να εκτυπώσετε οποιοδήποτε μήνυμα επιθυμείτε στην οθόνη, ακόμα και να παρέχετε μία ερώτηση και να περιμένετε για απάντηση απ' το χρήστη δημιουργώντας έτσι αλληλεπιδραστικά προγράμματα. Μία ακόμα χρήση των i/o συναρτήσεων είναι η ανάγνωση και εγγραφή δεδομένων από και προς αρχεία του δίσκου κάνοντας έτσι δυνατή την προσωρινή αλλά και μακροπρόθεσμη αποθήκευση δεδομένων.

Ιστορικά, το i/o έχει αποτελέσει θέμα ευρείας διαφωνίας μεταξύ των διαφόρων συστημάτων Lisp. Για παράδειγμα, μέχρι και σήμερα δεν έχει αποφασιστεί κάποιο στάνταρ window system interface, όπως επίσης και κάποιος στάνταρ τρόπος χειρισμού του mouse ή ανάπτυξης γραφικών. Για τις λειτουργίες αυτές, συνηθίζεται ο κάθε κατασκευαστής περιβάλλοντος ανάπτυξης εφαρμογών Lisp να παρέχει τα δικά του εργαλεία. Το θετικό της υπόθεσης είναι ότι οι βασικότερες και πιο συχνά χρησιμοποιούμενες συναρτήσεις για i/o έχουν προτυποποιηθεί και αυτές είναι που θα παρουσιάσουμε στην παρούσα εργαστηριακή άσκηση.

6.1.1 Συμβολοσειρές

Όπως θα γνωρίζετε ήδη, μία συμβολοσειρά είναι μια μορφή ακολουθίας (παράθεση συμβόλων ενός αλφαβήτου με τη διάταξη να έχει σημασία). Μοιάζουν κατά κάποιο τρόπο με τις λίστες, αλλά έχουν ένα διαφορετικό σύνολο primitive συναρτήσεων για το χειρισμό τους.

Οι συμβολοσειρές αποτιμούν στον εαυτό τους ακριβώς όπως και οι αριθμοί. Παρατηρήστε στα ακόλουθα παραδείγματα ότι οι χαρακτήρες που αποτελούν μια συμβολοσειρά δε μετατρέπονται στους αντίστοιχους κεφαλαίους όταν αποτιμούνται, αντίθετα με το τί ίσχυε για τα σύμβολα. Είναι ένας κομψός τρόπος για να κατανοήσουμε ότι οι συμβολοσειρές δεν είναι σύμβολα. Το κατηγόρημα `STRINGP` επιστρέφει `T` αν η είσοδός του είναι μία συμβολοσειρά.

- Να εκτελέσετε στο Listener τα ακόλουθα:

```
> ‘‘strings are strings’’
> (setf a ‘‘This object is a string.’’)
> (stringp a)
> a
> (setf b 'this-object-is-a-symbol)
> (stringp b)
```

Προσέξτε ότι οι συμβολοσειρές πρέπει να περικλείονται από τα διπλά εισαγωγικά του λατινικού αλφαβήτου (`"`), χαρακτήρας ο οποίος είναι διαφορετικός από το χαρακτήρα (`'`) που χρησιμοποιούσαμε για τα σύμβολα. Δύο φορές ο ' αντί για του " (με Shift+") δεν πρόκειται να δουλέψει στη Lisp.

6.1.2 Η συνάρτηση FORMAT

Η συνάρτηση `FORMAT` υπό φυσιολογικές συνθήκες επιστρέφει `NIL`, αλλά ως side effect προκαλεί την εγγραφή πραγμάτων στην οθόνη ή σε κάποιο αρχείο. Εάν επιθυμούμε την εκτύπωση στην οθόνη, αρκεί να δώσουμε ως πρώτο όρισμα στη `format` το σύμβολο `T`. Το δεύτερο όρισμα πρέπει να είναι μία συμβολοσειρά, που καλείται *format control string*.

- Πληκτρολογήστε τον ακόλουθο τύπο:

```
(format t "Hello World!")
```

Παρατηρήστε ότι η `format` εκτυπώνει τη συμβολοσειρά χωρίς τα quotes και εν συνεχεία επιστρέφει `NIL`.

Το format control string μπορεί επίσης να περιέχει ειδικούς χαρακτήρες διαμόρφωσης οι οποίοι ξεκινούν με τον χαρακτήρα tilde (∼). Παραδείγματος χάριν, το ∼ % κάνει τη `format` να αρχίσει να γράφει σε μία νέα γραμμή.

- Δοκιμάστε τα ακόλουθα:

1. `(format t "After this~%a new line will start.")`
2. `(format t "Now we leave an~%~%empty line!")`

Το ~ & κάνει τη `format` να αλλάξει γραμμή, εκτός κι αν γνωρίζει ότι βρίσκεται ήδη στην αρχή μίας νέας γραμμής. Δοκιμάστε να προσθέσετε σε ένα από τα προηγούμενα παραδείγματα δύο ή και περισσότερες συνεχόμενες φορές το ~ & και προσπαθήστε να ερμηνεύσετε το αποτέλεσμα. Προσθέστε το επίσης μία φορά μετά απ' το ~ % directive[1]. Μπορείτε να σκεφτείτε ότι το το ~ & directive υλοποιεί κατά κάποιο τρόπο την προτροπή: ''Εάν δεν έχεις αλλάξει, για κάποιο λόγο που δε γνωρίζω, γραμμή, άλλαξέ τη, αλλιώς μην κάνεις τίποτα''. Σε προγράμματα που παράγουν αρκετές γραμμές εξόδου στην οθόνη, είναι συνήθης πρακτική κάθε `format` να ξεκινάει με το ~ & directive, έτσι ώστε να είμαστε βέβαιοι ότι θα αλλάζει πάντα γραμμή πριν την εκτύπωση του τρέχοντος μηνύματος (εάν φυσικά θέλουμε να αλλάζει γραμμή).

- Η ακόλουθη συνάρτηση είναι ένα τέτοιο παράδειγμα:

```
(defun print_fun ()
  (format t "~&First line to screen.")
  (format t "~&Second line to screen.")
  (format t "~&Third line to screen.")
  (format t "~&Fourth line to screen.")
  (format t "~&Fifth line to screen."))
```

Ένα ακόμα πολύ σημαντικό formatting directive είναι το ~S, το οποίο εισάγει την εκτυπώσιμη αναπαράσταση ενός αντικειμένου Lisp στο μήνυμα που τυπώνει η `format`[2]. Όπως περίπου συμβαίνει και με την `printf` στην C, έτσι και εδώ, για κάθε εμφάνιση του ~S στο format control string απαιτείται και ένα ακόμα όρισμα στο τέλος της `format`.

- Εκτελέστε για παράδειγμα τον ακόλουθο διάλογο με τη Lisp:

```
> (format t "A ~S is a connected, acyclic graph" 'tree)
> (setf forest 'forest tree 'tree)
> (format t "A ~S is an acyclic graph.~%A ~S is a conne
   cted ~S." 'forest 'tree 'forest)
> (defun my-gcd (x y)
    ;;Euclidean Algorithm for greatest common divisor
    ;;gcd in common lisp but we redefine it here
    (let ((temp nil)
          (num2 y))
      (do ((result x))
          ((zerop num2) result)
          (setf result (mod result num2));modulo function
          (setf temp num2 num2 result result temp))))
> (format t "The greatest common divisor of ~S and ~S is:
   ~S" 5 6 (my-gcd 5 6))
```

- Δύο αριθμοί είναι *σχετικά πρώτοι* (*relatively prime*), εάν το 1 είναι ο μέγιστος ακέραιος που διαιρεί ακριβώς και τους δύο. Να ορίσετε μία συνάρτηση, ονόματι `rel-prime`, η οποία θα παίρνει ως ορίσματα δύο αριθμούς και με χρήση της `format` θα ενημερώνει το χρήστη για το εάν είναι σχετικά πρώτοι ή όχι.

[1] Οι ειδικοί αυτοί χαρακτήρες καλούνται και *special formatting directives*.
[2] Το S προέρχεται απ' το γνωστό μας *s-expression*, που σημαίνει symbolic expression.

Το ~A directive εκτυπώνει ένα αντικείμενο χωρίς να χρησιμοποιεί χαρακτήρες διαφυγής, όπως π.χ. τα quotes.

- Δοκιμάστε την ακόλουθη συνάρτηση με είσοδο `"Hello World"`:

```
(defun test (x)
  (format t "~&With escape characters: ~S" x)
  (format t "~&Without escape characters: ~A" x))
```

Οι ακόλουθες ασκήσεις θα σας βοηθήσουν να εξοικειωθείτε περαιτέρω με τη χρήση της `format`:

1. Γράψτε μία αναδρομική συνάρτηση `draw-line`, η οποία θα εκτυπώνει μία γραμμή με βάση το ζητούμενο μήκος, χρησιμοποιώντας την `(format t "*")` τον κατάλληλο αριθμό φορών. Για παράδειγμα, η κλήση `(draw-line 10)` θα πρέπει να εκτυπώσει:

```
**********
```

2. Γράψτε μία αναδρομική συνάρτηση `draw-box`, η οποία θα καλεί τη `draw-line` n φορές με είσοδο m, έτσι ώστε να κατασκευάσει ορθογώνιο παραλληλόγραμμο διαστάσεων (m, n), όπου m το μήκος και n το πλάτος.

3. Να ορίσετε μία συνάρτηση `combinations`, η οποία θα παίρνει δύο ορίσματα, n και r, και θα επιστρέφει το $C(n, r)$ (δηλαδή, το $\binom{n}{r}$), το οποίο είναι όλοι οι δυνατοί τρόποι με τους οποίους μπορούμε να επιλέξουμε r αντικείμενα από μία συλλογή n αντικειμένων χωρίς να μας ενδιαφέρει η διάταξη. Το αποτέλεσμα θα πρέπει να εκτυπώνεται σε ψηφιακή μορφή, π.χ. το $C(6, 2) = 15$ θα πρέπει να εκτυπώνεται ως:

```
 |
 |
 _
|_
 _|
```

(Υπόδειξη: $C(n, r) = \frac{n!}{r!(n-r)!}$ ο αριθμός των συνδυασμών r αντικειμένων από n αντικείμενα[3]).

6.1.3 Η συνάρτηση READ

Η `READ` είναι μία συνάρτηση που μπορεί να χρησιμοποιηθεί για την ανάγνωση δεδομένων εισόδου που δίνονται απ' το πληκτρολόγιο. Οποτεδήποτε χρησιμοποιήσουμε τη `read` σε μία συνάρτηση, ο έλεγχος σταματάει στη `read` και περιμένει για δεδομένα εισόδου απ' τον χρήστη. Μόλις ο χρήστης εισάγει τα δεδομένα και πατήσει

[3] Ο αριθμός των τρόπων να διατάξω r αντικείμενα από n είναι $P(n, r) = n!/(n-r)!$, όμως το να διατάξω r αντικείμενα από n είναι το ίδιο με το να επιλέξω πρώτα r αντικείμενα από n και εν συνεχεία να διατάξω τα r αντικείμενα που επέλεξα με όλους τους δυνατούς τρόπους (αντιμεταθέσεις r αντικειμένων)· επομένως, $P(n, r) = C(n, r)P(r, r)$ και έτσι προκύπτει εύκολα το $C(n, r)$, αφού $P(r, r) = r!$.

το carriage return, η `read` τα διαβάζει και τα επιστρέφει. Εφόσον επιστρέφει τα δεδομένα, μπορούμε είτε να τα χρησιμοποιήσουμε απ' ευθείας, εάν είχαμε τοποθετήσει τη `read` ως όρισμα μιας άλλης συνάρτησης, είτε να τα αποθηκεύσουμε σε κάποια μεταβλητή με χρήση της `setf` ή της `let`.

- Δοκιμάστε για παράδειγμα την ακόλουθη συνάρτηση στο Listener:

  ```
  > (setf read-data (read))
  ```

  Καλέστε εν συνεχεία τη μεταβλητή `read-data` και θα σας επιστραφεί η τιμή που της δώσατε μέσω του πληκτρολογίου.

- Να μετατρέψετε τη συνάρτηση `combinations` που ορίσατε στην προηγούμενη υποενότητα, έτσι ώστε να παίρνει ένα μόνο όρισμα έστω max_val. Η συνάρτηση θα παράγει μία τυχαία τιμή w, όπου $0 < w < max\_val$ και θα ζητάει από το χρήστη να δώσει μία τιμή από 1 έως w για το n (πλήθος αντικειμένων συλλογής) και μία τιμή από 1 έως n για το r (ο χρήστης θα πρέπει να μπορεί να δει στην οθόνη τα πεδιά ορισμού απ' τα οποία μπορεί να επιλέξει). Επιπρόσθετα θα πρέπει να ελέγχονται τα δεδομένα που εισάγει ο χρήστης για την περίπτωση που είναι εκτός πεδίου ορισμού, περίπτωση κατά την οποία ο χρήστης θα ενημερώνεται με κατάλληλο μήνυμα και θα μπορεί να προσπαθήσει και πάλι. Όταν τα n και r εισαχθούν επιτυχώς, τα υπόλοιπα θα γίνονται όπως και πριν.

Σε περιπτώσεις που επιθυμείτε να πάρετε μία απάντηση της μορφής ''ναι'' ή ''όχι'', μπορείτε να χρησιμοποιήσετε τις συναρτήσεις `yes-or-no-p` ή `y-or-n-p` στη θέση της `read`. Εάν η `yes-or-no-p` πάρει "yes" επιστρέφει `T`, ενώ αν πάρει "no" επιστρέφει `NIL`. Αντίστοιχα, η `y-or-n-p` περιμένει "y" ή "n". Αν τις δοκιμάσετε στο Lispworks, θα παρατηρήσετε ότι στο περιβάλλον αυτό δεν έχουν καμία ουσιαστική διαφορά. Βλέπετε γιατί;

6.1.4 Η συνάρτηση WITH-OPEN-FILE

Η `WITH-OPEN-FILE` μας παρέχει έναν εύκολο τρόπο για να διαχειριζόμαστε αρχεία. Η γενική της σύνταξη έχει ως εξής:

`(with-open-file (`*var pathname*`) `*body*`)` Η `with-open-file`, όπως και η `let`, δημιουργεί την τοπική μεταβλητή *var* και της αναθέτει ένα *stream object* που αναπαριστά τη σύνδεση με το αρχείο που δείχνει το *pathname*.

Μόλις ανοίξουμε το αρχείο με τη `with-open-file`, μπορούμε είτε να διαβάσουμε απ' αυτό είτε να γράψουμε σε αυτό. Το τί θα κάνουμε βέβαια οφείλουμε να το έχουμε ξεκαθαρίσει στη Lisp με κάποια ειδικά ορίσματα που πρέπει να προηγούνται όλων των συναρτήσεων στο σώμα της `with-open-file`. Πάντως, στην περίπτωση που θέλουμε να διαβάσουμε από αρχείο δεν χρειάζεται τίποτα απ' αυτά καθώς η ανάγνωση είναι default. Ας δούμε ένα απλό παράδειγμα ανάγνωσης από αρχείο για να γίνουν κατανοητά τα προαναφερθέντα.

- Να δημιουργήσετε το αρχείο timber.dat σε κάποιο directory του δίσκου (εμείς εδώ υποθέτουμε ότι έχει δημιουργηθεί κάτω απ' το δίσκο C:) και να του εισάγετε τα ακόλουθα:

```
"The North Slope"
((45 redwood) (12 oak) (43 maple))
100
```

Αφού το αποθηκεύσετε, να ορίσετε στο Lispworks την ακόλουθη συνάρτηση:

```
(defun get-tree-data ()
  (with-open-file (stream "C:/timber.dat")
    (let* ((tree-loc (read stream))
           (tree-table (read stream))
           (num-trees (read stream)))
      (format t "~&There are ~S trees on ~S." num-trees
        tree-loc)
      (format t "~&They are: ~S" tree-table))))
```

και αφού τη μελετήσετε προσεκτικά, να την τρέξετε για να δείτε πως λειτουργεί.

Υπάρχουν αρκετά πράγματα τα οποία αξίζει να συζητήσουμε σχετικά με την παραπάνω συνάρτηση. Το πρώτο πράγμα που βλέπουμε είναι ότι το σώμα της `with-open-file` αποτελείται από όλες τις συναρτήσεις που θέλουμε να εκτελέσουμε σχετικά με το αρχείο. Στην πραγματικότητα εδώ αποτελείται μόνο απ' τη `let*`, η οποία και καλεί όλες τις άλλες συναρτήσεις. Φυσικά, δε θα μπορούσαμε να έχουμε κλείσει τη `with-open-file` πριν εκτελέσουμε όλες τις ενέργειες που σχετίζονται με το αρχείο, διότι πολύ απλά θα βγαίναμε εκτός της εμβέλειας της μεταβλητής `stream` και επομένως δεν θα μπορούσαμε να επικοινωνήσουμε με το αρχείο. Αυτό μας λέει το εξής πολύ σημαντικό: ''Μετά το πέρας του τύπου της `with-open-file`, το αρχείο που είχαμε ανοίξει κλείνει αυτόματα''. Αν η `with-open-file` δε βρει το αρχείο το οποίο προσπαθεί να ανοίξει για ανάγνωση, επιστρέφει ένα μήνυμα σφάλματος. Αν το αρχείο βρεθεί και η σύνδεση γίνει επιτυχώς, μπορούμε να καλέσουμε τη `read` τόσες φορές, όσους τύπους δεδομένων θέλουμε να διαβάσουμε απ' την αρχή του αρχείου. Κάθε φορά που καλείται η `read` με όρισμα τη μεταβλητή που περιέχει το stream object (εδώ είναι η μεταβλητή `stream`), διαβάζεται ο επόμενος τύπος δεδομένων του αρχείου.

Ας δούμε τώρα πώς μπορούμε να γράψουμε σε ένα αρχείο χρησιμοποιώντας τη `with-open-file` για να το ανοίξουμε. Το πρώτο καινούριο πράγμα που πρέπει να κάνουμε είναι να πούμε με κάποιο τρόπο στη `with-open-file` ότι ανοίγουμε το αρχείο για εγγραφή. Αυτό μπορεί να γίνει με τη χρήση του ορίσματος λέξεων-κλειδιών `:DIRECTION :OUTPUT`. Η λέξη-κλειδί `:OUTPUT` μπορεί να θεωρηθεί ως όρισμα της `:DIRECTION`. Άλλα πιθανά ορίσματα μπορεί να είναι τα `:INPUT` και `:IO` με το τελευταίο να δηλώνει ταυτόχρονη εγγραφή και ανάγνωση.

- Να πληκτρολογήστε την ακόλουθη συνάρτηση:

```
(defun save-tree-data (tree-loc tree-table num-trees)
  (with-open-file (stream "C:/timber.newdat" :direction
                   :output)
    (format stream "~S~%" tree-loc)
    (format stream "~S~%" tree-table)
    (format stream "~S~%" num-trees)))
```

Τώρα να την καλέσετε ως εξής:

```
(save-tree-data
  "The West Ridge"
  '((45 redwood) (22 oak) (43 maple))
  110)
```

Να εντοπίσετε το αρχείο `timber.newdat` και να το ανοίξετε για να επαληθεύσετε ότι όλα έχουν εγγραφεί σωστά. Καλέστε τώρα και πάλι τη συνάρτηση με παρόμοια ορίσματα. Τί παρατηρείτε;

Το μήνυμα σφάλματος που προέκυψε όταν προσπαθήσατε να τρέξετε και πάλι τη `save-tree-data` οφείλεται στο γεγονός ότι το αρχείο υπήρχε ήδη από το προηγούμενο τρέξιμο. Υπάρχουν κάποιες λέξεις−κλειδιά, οι οποίες μας επιτρέπουν να χειριστούμε τέτοιες καταστάσεις. Για παράδειγμα, προσθέστε μετά το `:direction :output` στη `save-tree-data` το `:if-exists :append` και τρέξτε αρκετές φορές τη συνάρτηση. Με τον τρόπο αυτό, λέμε στη `with-open-file` τί να κάνει εάν το αρχείο υπάρχει ήδη. Το default είναι `:error` γι' αυτό και σας επεστράφη μήνυμα σφάλματος προηγουμένως. Αντίθετα, όταν αλλάξαμε το `:error` με το `:append`, δηλώσαμε ότι αν το αρχείο υπάρχει ήδη, θέλουμε τα νέα δεδομένα να προστεθούν στο τέλος του. Άλλα πιθανά ορίσματα για την `:if-exists` είναι τα `:overwrite` και `NIL`. Η `:overwrite` αντικαθιστά τα περιεχόμενα του αρχείου με τις νέες εγγραφές, ενώ η `NIL` δεν κάνει τίποτα και επιστρέφει `NIL` ως ένδειξη σφάλματος. Αντίστοιχα, μπορούμε να χρησιμοποιήσουμε τη λέξη-κλειδί `:if-does-not-exist` σε συνδυασμό με ένα απ' τα ορίσματα `:error`, `:create` και `NIL`, για να δηλώσουμε τί επιθυμούμε να γίνει αν το αρχείο δεν υπάρχει.

Οι ακόλουθες ανακεφαλαιωτικές ασκήσεις θα σας βοηθήσουν να εξοικειωθείτε περαιτέρω με όσα είδαμε στην παρούσα εργαστηριακή άσκηση:

1. Να ορίσετε μία συνάρτηση `arb-num`, η οποία θα παίρνει ως ορίσματα δύο αριθμούς n, x και ένα $filename$ και θα δημιουργεί ένα νέο αρχείο σύμφωνα με το $filename$ στο οποίο θα τυπώνει n τυχαίους αριθμούς από το διάστημα $[0, x]$.

2. Να ορίσετε μία συνάρτηση `my-sort`, η οποία θα διατάσει με αποδοτικό τρόπο και κατά αύξουσα σειρά μία λίστα με αριθμούς.

3. Να ορίσετε μία συνάρτηση `find-num`, η οποία θα δημιουργεί ένα αρχείο με n τυχαίους αριθμούς από το διάστημα $[0, x]$ (με χρήση της 1) και αφού τους διατάξει (με χρήση της 2), θα μπαίνει σε ένα loop όπου θα γίνονται τα εξής: Ο χρήστης θα δίνει είτε έναν αριθμό είτε ένα filename απ' το οποίο το πρόγραμμα θα πρέπει να πάρει τον αριθμό και εν συνεχεία θα ελέγχει εάν ο αριθμός υπάρχει στο αρχείο με τους διατεταγμένους αριθμούς. Εάν υπάρχει, θα ενημερώνει το χρήστη με κατάλληλο μήνυμα, ενώ αν δεν υπάρχει θα τον προσθέτει στην κατάλληλη θέση του αρχείου έτσι ώστε να διατηρείται η διάταξη. Η συνάρτηση θα πρέπει να είναι σε θέση να ενημερώσει τον χρήστη ότι όλοι οι αριθμοί από $[0, x]$ έχουν εισαχθεί, αν κάποια στιγμή γίνει αυτό, έτσι ώστε να σταματήσει το παιχνίδι. Επιπρόσθετα, ο χρήστης θα πρέπει να μπορεί να τερματίσει το loop , όποτε το θελήσει, εισάγοντας αντί για αριθμό τη συμβολοσειρά exit. Τέλος, θα πρέπει να κοινοποιείται στο χρήστη σε ποιά θέση εισήχθη ο αριθμός, εάν δεν υπήρχε, ενώ εάν υπήρχε,

σε ποιά θέση υπήρχε. Περαιτέρω λεπτομέρειες της υλοποίησης επαφίενται στην κρίση σας.

4. Σε μία συνέντευξη πρόσληψης μηχανικών υπολογιστών, γνωστής πολυεθνικής εταιρίας, οι υποψήφιοι καλούνταν να δώσουν λύση στο ακόλουθο πρόβλημα:

   > ''Έστω ότι διαθέτετε δύο αυγά και βρίσκεστε σε μία πολυκατοικία με 100 ορόφους. Θέλετε με αποδοτικό τρόπο να διαπιστώσετε από ποιόν όροφο και πάνω θα σπάσει ένα αυγό, εάν το αφήσετε να πέσει στο έδαφος. Εάν πετάξετε ένα αυγό από κάποιο όροφο και δε σπάσει, μπορείτε να επαναχρησιμοποιήσετε το αυγό''.

   Να σχεδιάσετε ένα πρόγραμμα Lisp, το οποίο θα εξομοιώνει το φυσικό περιβάλλον μέσα στο οποίο θα μπορεί κάποιος να δοκιμάσει μια λύση που σκέφτηκε. Το πρόγραμμα θα πρέπει να διαλέγει τυχαία το πλήθος των ορόφων της πολυκατοικίας, καθώς και τον όροφο απ' τον οποίο και πάνω τα αυγά θα σπάνε. Θα πρέπει, επίσης, να αφήνει το χρήστη να κάνει ρίψεις από τους διάφορους ορόφους, αφού τον ενημερώσει για το ύψος της πολυκατοικίας. Εάν κάποια στιγμή σπάσουν και τα δύο αυγά, το πρόγραμμα θα ζητάει απ' το χρήστη να απαντήσει ποιός είναι ο όροφος απ' τον οποίο και πάνω τα αυγά σπάνε. Φυσικά, ο χρήστης θα πρέπει να μπορεί να απαντήσει και πιο νωρίς, εάν το επιθυμεί. Εάν η απάντησή του είναι λανθασμένη, το πρόγραμμα θα τον ενημερώνει ότι έχασε και θα του δίνει τη δυνατότητα να προσπαθήσει και πάλι. Εάν η απάντησή του είναι σωστή, το πρόγραμμα θα τον ενημερώνει για αυτό, θα του εμφανίζει το πλήθος των προσπαθειών που έκανε και θα του αντιπαραβάλει το πλήθος των βημάτων με τα οποία θα έλυνε το πρόβλημα ένας αποδοτικός αλγόριθμος. Τον αλγόριθμο αυτό θα πρέπει να τον υλοποιήσετε με μία ή περισσότερες συναρτήσεις Lisp και το ποιός θα μπορούσε να είναι αυτός ο αλγόριθμος επαφίεται στην κρίση σας. Όλη η αλληλεπίδραση με το χρήστη θα μπορεί να γίνει και μέσω αρχείων. Δηλαδή, ο χρήστης μπορεί να έχει εισαγάγει όλες τις προσπάθειές του σε ένα αρχείο απ' όπου και θα πρέπει να τις διαβάζει το πρόγραμμα ή να τις δίνει σε κάθε βήμα απ' το πληκτρολόγιο. Επίσης, θα πρέπει να μπορεί να πάρει την απάντηση του προγράμματος είτε στην οθόνη είτε σε κάποιο αρχείο είτε και στα δύο, ανάλογα με το τί επιθυμεί κάθε φορά.