Κεφάλαιο 2

Εργαστηριακή Άσκηση 2

Όπου θα δούμε πώς μπορούμε να ορίζουμε δικές μας διαδικασίες και θα παρουσιάσουμε τις primitive διαδικασίες χειρισμού λιστών, τις μεταβλητές και τα side effects.

2.1 Πώς ορίζουμε διαδικασίες

Είδαμε ότι, πέραν των primitives, μπορούν να υπάρχουν και *user-defined διαδικασίες*. Η `DEFUN` είναι μία ειδική διαδικασία-συνάρτηση[1], που καλείται *macro function*, και δεν αποτιμά τα ορίσματά της. Επομένως, αυτά δεν χρειάζεται να συνοδεύονται από απόστροφο. Η συνάρτηση αυτή χρησιμοποιείται για τον ορισμό νέων συναρτήσεων[2].

- Να ορίσετε μία απλοϊκή έκδοση της συνάρτησης `average` (*μέσος όρος*) ως εξής:

  ```
  (defun average (x y)
    (/ (+ x y) 2.0))
  ```

  αφού την αποτιμήσει ο Listener, καλέστε τη με τη μορφή `(average x y)`, όπου στη θέση των `x` και `y` θα βάλετε τους αριθμούς των οποίων αναζητάτε το μέσο όρο.

Πριν σας ζητηθεί να ορίσετε κάποιες δικές σας συναρτήσεις, θα συζητήσουμε εν συντομία τον παραπάνω τύπο της `DEFUN`. Το πρώτο όρισμα που δώσαμε στη `DEFUN`, δηλαδή το `average`, είναι το όνομα της συνάρτησης την οποία ορίζουμε. Το δεύτερο όρισμα που της δώσαμε, δηλαδή η λίστα `(x y)`, είναι η λεγόμενη *λίστα ορισμάτων* (*argument list*): Καθορίζει τόσο το πλήθος των ορισμάτων που θα έχει η νέα συνάρτηση, όσο και τα ονόματα των ορισμάτων έτσι ώστε να μπορούμε να αναφερθούμε σε αυτά στο *σώμα της συνάρτησης*. Το σώμα της συνάρτησης είναι η τελευταία γραμμή του παραπάνω τύπου (το τελευταίο όρισμα), το οποίο

[1] Όπως έχουμε πει, τα primitives είναι διαδικασίες. Οι διαδικασίες που απλώς υπολογίζουν μία τιμή ονομάζονται *συναρτήσεις* (*functions*). Επομένως, δεν είναι όλες οι διαδικασίες συναρτήσεις. Για να κάνουμε τα πράγματα απλούστερα, από εδώ και στο εξής, θα αναφερόμαστε σε όλες τις διαδικασίες ως συναρτήσεις, όπως εξάλλου συνηθίζεται στη βιβλιογραφία.

[2] DEFUN: **Def**ine **Fun**ction.

ορίζει την κυρίως λειτουργία της συνάρτησης που θα δημιουργηθεί. Επομένως, μπορούμε να δώσουμε ένα γενικότερο τρόπο χρήσης της DEFUN:

(defun *function_name* (*arguments*) *body*)

Ασκήσεις:

1. Να γράψετε μία συνάρτηση, που να υπολογίζει το μήκος της υποτείνουσας ορθογωνίου τριγώνου εάν είναι γνωστά τα μήκη των δύο κάθετων πλευρών. Σε περίπτωση που δεν τον θυμόσαστε, ο μαθηματικός τύπος που θα πρέπει να υλοποιήσετε είναι:
$$f(x, y) = \sqrt{x^2 + y^2}$$

2. Να γράψετε μία συνάρτηση, που θα παίρνει τρία ορίσματα και θα επιστρέφει το γινόμενο του κύβου του πρώτου ορίσματος με την απόλυτη τιμή του αθροίσματος των δύο τελευταίων ορισμάτων. Δηλαδή:
$$f(x, y, z) = x^3 * |y + z|$$

3. Να γράψετε μία συνάρτηση που θα υπολογίζει το εμβαδόν ισοσκελούς τριγώνου, εάν είναι γνωστά τα μήκη των πλευρών του. Θα χρειαστείτε πάλι το πυθαγόρειο θεώρημα μόνο που τώρα θα πρέπει να βρίσκει τη μία κάθετη πλευρά και όχι την υποτείνουσα όπως στο 1. Να τροποποιήσετε κατάλληλα την συνάρτηση του 1, έτσι ώστε να μπορείτε να την καλέσετε μέσα απ' τη συνάρτηση του εμβαδού όταν τη χρειαστείτε.

2.2 Μεταβλητές και side effects

Μέχρι τώρα έχουμε δει ότι η Lisp προσπαθεί να αποτιμήσει οτιδήποτε της δίνεται υπό μορφή τύπου[3]. Όπως είδαμε στο προηγούμενο εργαστήριο, αν εισάγουμε ένα μη-δεσμευμένο σύμβολο και πατήσουμε το carriage return, η Lisp θα μας απαντήσει με ένα μήνυμα σφάλματος. Αυτό συμβαίνει διότι προσπαθεί να βρει τιμή για το σύμβολο αυτό, το οποίο όμως δεν είναι δεσμευμένο σε κάποια τιμή.

> *Η διαδικασία κράτησης μιας θέσης μνήμης για την αποθήκευση μίας τιμής για κάποιο σύμβολο καλείται* πρόσδεση *(*binding*). Η διαδικασία της αποθήκευσης μίας τιμής σε αυτή τη θέση καλείται* ανάθεση *(*assignment*). Η διαδικασία της επαναφοράς μίας τιμής από αυτή τη θέση μνήμης είναι μία μορφή* αποτίμησης.

Είδαμε επίσης ότι τα ίδια τα σύμβολα μπορούν να χρησιμοποιηθούν ως μεταβλητές. Όπως σε όλες τις γλώσσες προγραμματισμού, έτσι και στη Lisp , κάθε σύμβολο-μεταβλητή έχει μία *εμβέλεια* (*scope*). Η εμβέλεια αυτή είναι η περιοχή μέσα στην οποία μπορούν να γίνονται αναφορές στη μεταβλητή. Προσοχή, διότι αν αναφερθείτε σε μία μεταβλητή εκτός της εμβέλειάς της, το αποτέλεσμα μπορεί να είναι απροσδιόριστο. Μέχρι τώρα, έχουμε συναντήσει αρκετά σύμβολα ως ορίσματα μίας συνάρτησης. Αν υποθέσουμε ότι είχαμε αναθέσει στα σύμβολα αυτά κάποια τιμή, έτσι ώστε να τα θεωρούμε ως μεταβλητές, τότε θα λέγαμε ότι αυτές είναι *τοπικές μεταβλητές* (*local variables*), καθώς η εμβέλειά τους περιορίζεται στο σώμα της συνάρτησης.

[3] Πρόκειται συνήθως για ό,τι δίνεται χωρίς απόστροφο.

- Σκεφτείτε το παράδειγμα:

```
(defun double (n) (* 2 n))
```

Κάθε φορά που καλούμε τη συνάρτηση `double`, δημιουργείται μία νέα τοπική μεταβλητή ονόματι `n`. Οποιαδήποτε αναφορά στη `n` εντός του σώματος της `double` είναι ορθή καθώς βρισκόμαστε μέσα στην εμβέλειά της. Αντίθετα, εκτός της `double`, η `n` είναι μία τελείως διαφορετική μεταβλητή, η οποία δεν είναι τοπική σε κάποια συγκεκριμένη συνάρτηση, είναι δηλαδή μία *καθολική μεταβλητή* (*global variable*). Αν καλέσουμε την καθολική μεταβλητή `n` χωρίς να την έχουμε δεσμεύσει σε κάποια τιμή, θα πάρουμε ένα μήνυμα σφάλματος. Να επιβεβαιώσετε στο Listener την ορθότητα αυτού του ισχυρισμού. Να σημειώσουμε ότι μπορεί να υπάρχει μόνο μία καθολική μεταβλητή ονόματι `n`, αλλά πολλές τοπικές μεταβλητές με το ίδιο όνομα, αρκεί να μην επικαλύπτονται οι εμβέλειές τους.

Η `SETF` macro function αναθέτει μία τιμή σε μία μεταβλητή[4]. Εάν η μεταβλητή έχει ήδη κάποια τιμή, η νέα τιμή αντικαθιστά την παλιά.

- Εισάγετε στο Listener τον ακόλουθο τύπο:

```
(setf planets-to-visit '(jupiter mars venus saturn pluto))
```

Η `SETF`, επειδή ακριβώς είναι μία macro function, δεν αποτιμά το πρώτο της όρισμα (εδώ είναι το `planets-to-visit`). Επιπρόσθετα, αντί απλώς να επιστρέφει την τιμή του δεύτερου ορίσματός της (πράγμα το οποίο ούτως ή άλλως κάνει, γι' αυτό και σας τυπώνει τη λίστα), αναθέτει επιπλέον την τιμή αυτή στο πρώτο της όρισμα[5]. Αρκεί να καλέσουμε τώρα τη μεταβλητή `planets-to-visit` για να δούμε τί τιμή της έχει ανατεθεί. Οτιδήποτε έχει κάνει κάποια συνάρτηση, το οποίο παραμένει και μετά την επιστροφή τιμής απ' τη συνάρτηση, καλείται *side effect*. Επομένως, μπορούμε να πούμε ότι ο βασικός λόγος για τον οποίο αποτιμάται ο παραπάνω τύπος είναι το side effect του και όχι η τιμή την οποία επιστρέφει. Η συνάρτηση `DEFUN` για παράδειγμα καλείται μόνο για το side effect της, αφού η τιμή την οποία επιστρέφει κάθε φορά είναι απλώς το όνομα της νέας συνάρτησης την οποία όρισε.

Η `SETF` θα μπορούσε να χρησιμοποιηθεί ακόμα και για να αλλάξει την τιμή μιας τοπικής μεταβλητής. Βέβαια, αυτό δεν αποτελεί καλή προγραμματιστική πρακτική γι' αυτό κι εμείς θα τη χρησιμοποιούμε μόνο για να αναθέτουμε ή να αλλάζουμε τιμή σε καθολικές μεταβλητές· πάντως, αξίζει να τρέξετε το ακόλουθο παράδειγμα για να δείτε πώς γίνεται:

```
(defun poor-style (p)
  (setf p (+ p 5))
  (list 'result 'is p))
```

Καλέστε τώρα `(poor-style 8)` και θα καταλάβετε πως λειτουργεί.

Υποθέστε ότι ζείτε σε μια εποχή που τα ταξίδια στο διάστημα είναι απλώς μια καθημερινότητα. Η λίστα που είδαμε πριν είναι οι πλανήτες που σκοπεύετε να επισκεφθείτε για να περάσετε τις χειμερινές σας διακοπές. Ο Πλούτωνας απέχει απ' τη Γη $4,34 \times 10^{12} m$, οπότε ακόμα και αν ταξιδεύατε με την ταχύτητα του φωτός θα χρειαζόσασταν $t = d/c = 1.447 \times 10^4 s \simeq 4 hours$. Γενικά, σας κουράζουν

[4] Είναι εξίσου σωστό είτε πούμε ότι αναθέτουμε μία τιμή σε μία μεταβλητή, είτε ότι δεσμεύουμε μία μεταβλητή σε μία τιμή.

[5] Η `SETF` δεν εφαρμόζεται μόνο σε σύμβολα και γι' αυτό το λόγο προτιμάται απ' τους προγραμματιστές, σε αντίθεση με τις `SETQ` και `SET` που έχουν πιο περιορισμένες δυνατότητες.

τα πολύωρα ταξίδια και θα θέλατε όλο το διαθέσιμο χρόνο να τον ξοδέψετε στον προορισμό σας. Θέλετε επομένως να διαγράψετε τον Πλούτωνα από τη λίστα σας και να εισάγετε στη θέση του κάποιον άλλο, πιο κοντινό πλανήτη, όπως για παράδειγμα τον Ερμή, που πρόσφατα ακούσατε ότι προσφέρει φτηνές και χαλαρωτικές διακοπές. Εκτελείτε, επομένως, στο Listener του Lisp PDA σας τα εξής:

```
> (setf planets-to-visit (remove 'pluto planets-to-visit))
> (setf planets-to-visit (cons 'mercury planets-to-visit))
```

Το πώς κάνουν αυτό που κάνουν οι συναρτήσεις `CONS` και `REMOVE`, μη σας απασχολεί προς το παρόν. Αυτό που θα πρέπει να σας απασχολήσει είναι το πώς θα κάνουμε τα παραπάνω να εκτελούνται κάθε φορά που εμείς καλούμε μια δική μας συνάρτηση, την `change-option`. Η συνάρτηση αυτή θα παίρνει δύο ορίσματα, συγκεκριμένα, δύο πλανήτες x, y· θα διαγράφει τον x απ' τη λίστα που περιέχεται στην καθολική μεταβλητή `planets-to-visit` και θα εισάγει τον y στην ανανεωμένη, μετά τη διαγραφή, λίστα της `planets-to-visit`.

- Να δημιουργήσετε τη συνάρτηση `change-option`, χρησιμοποιώντας τις δύο `setf` που δείξαμε προηγουμένως, χωρίς να ξεχνάτε ότι *κατά τον ορισμό μιας συνάρτησης, στο σώμα αυτής, μπορούμε και πρέπει να αναφερόμαστε στις μεταβλητές/ορίσματα χωρίς απόστροφο.* Σκεφτείτε απλά ότι αν βάλετε απόστροφο, είναι σα να λέτε στη Lisp να ασχοληθεί με το όνομα της μεταβλητής και όχι με την τιμή που της έχει ανατεθεί.

Αξίζει τέλος να σημειώσουμε ότι χρησιμοποιώντας μία μόνο `setf` μπορούμε να πραγματοποιήσουμε πολλές αναθέσεις σε αντίστοιχο αριθμό μεταβλητών. Τα περιττά ορίσματα πρέπει να είναι οι μεταβλητές, οι οποίες δεν αποτιμούνται, ενώ κάθε άρτιο όρισμα, έστω $2n$, πρέπει να είναι η έκφραση που θέλουμε να αναθέσουμε στο $2n-1$ όρισμα-μεταβλητή. Για παράδειγμα:

```
(setf first-variable '(1 2)
      second-variable '(a b)

             ⋮

      final-variable '(nothing important))
```

2.3 Χειρισμός Λιστών

Στην προηγούμενη εργαστηριακή άσκηση, μεταξύ άλλων, είδαμε τί είναι οι λίστες και πώς μπορούμε να τις αναπαριστούμε στην Common Lisp. Αν θυμηθείτε τον ορισμό που δώσαμε, θα δείτε ότι μας δίνει τη δυνατότητα να εισάγουμε λίστες απεριόριστης πολυπλοκότητας που μπορεί να είναι πολύ διαφορετικές μεταξύ τους. Επομένως, εύλογα θα αναρωτηθεί κανείς: ''*Με ποιά εργαλεία θα μπορούμε να χειριζόμαστε τις λίστες που θα δημιουργήσουμε*''; Την απάντηση δίνει η πληθώρα primitive διαδικασιών χειρισμού λιστών που μας παρέχει η Common Lisp και τις οποίες θα αρχίσουμε παρουσιάζουμε ευθύς αμέσως[6].

Έστω, ότι κάποιος φοιτητής έχει κάνει την ακόλουθη λίστα, με τους τομείς που τον γοητεύουν για να ασχοληθεί ερευνητικά:

[6] Θα μιλήσουμε, δηλαδή, για *LISt Processing*, απ' όπου ακριβώς πήρε η Lisp το όνομά της.

```
(theory-of-computation computational-complexity artificial-
intelligence distributed-systems algorithms)
```

Όταν δημιούργησε τη λίστα, κατέταξε τους τομείς κατά φθίνουσα προτεραιότητα, δηλαδή αυτό που τον ενδιέφερε περισσότερο ήταν η *Θεωρία Υπολογισμού* ενώ αυτό που τον ενδιέφερε λιγότερο ήταν οι *Αλγόριθμοι*. Στην πορεία, όμως, διαπίστωσε ότι για να ασχοληθεί με όλα τα υπόλοιπα, μάλλον θα έπρεπε πρώτα να διαβάσει αρκετά καλά το γενικό κομμάτι των αλγορίθμων γεγονός που τον αναγκάζει να το μεταφέρει στην πρώτη θέση της λίστας του[7]. Ή έστω, για παράδειγμα, πως αποφασίζει ξαφνικά, ότι θα ήταν προτιμότερο να ξεκινήσει απ' αυτό που τον ενδιαφέρει λιγότερο και αφού περάσει απ' όλους τους άλλους τομείς να καταλήξει με ένα πολύ γερό υπόβαθρο στον τομέα που τον ενδιαφέρει περισσότερο, όπου θα δώσει και όλες του τις δυνάμεις. Κάτι τέτοιο θα απαιτούσε να αναθεωρήσει τον τρόπο που βλέπει τη λίστα του και να δεχτεί ότι οι θέσεις δηλώνουν απλώς τη σειρά με την οποία θα ασχοληθεί με τους τομείς και όχι το πόσο τον ενδιαφέρει ο καθένας απ' αυτούς. Ακόμη θα έπρεπε με κάποιο τρόπο να αντιστρέψει τη λίστα του, να βρει δηλαδή έναν μέσο και το κάθε στοιχείο της να μεταβεί στη συμμετρική θέση ως προς αυτόν τον μέσο (το πρώτο θα γίνει τελευταίο, το δεύτερο, προτελευταίο κ.ο.κ.). Σε αυτή την εργαστηριακή άσκηση, μεταξύ άλλων, θα δούμε και πώς μπορούμε να χρησιμοποιήσουμε τα `first`, `rest`, `append`, `list`, `cons`, καθώς και άλλα πολύ χρήσιμα primitives για να χειριστούμε τις λίστες μας.

HCONS είναι η βασική συνάρτηση με την οποία μπορούμε να κατασκευάζουμε λίστες. Ας δούμε με ένα template πώς θα μπορούσαμε να καλέσουμε τη συνάρτηση `cons`:

```
(cons new_first_element a_list)
```

Η λειτουργία της έχει ως εξής:

> *Η* `cons` *παίρνει ως ορίσματα μία έκφραση και μία λίστα και επιστρέφει μία νέα λίστα, της οποίας το πρώτο στοιχείο είναι η έκφραση, ενώ τα υπόλοιπα είναι τα στοιχεία της λίστας που είχε πάρει ως όρισμα.*

Το όνομα `cons` προέρχεται απ' το *constructor*, αφού πρόκειται για το βασικό κατασκευαστή λιστών της Lisp. Αξίζει, επίσης, να ξεκαθαρίσουμε ότι τόσο η έκφραση, όσο και η λίστα που δίνονται ως ορίσματα στην `cons` είναι *αντικείμενα* (*objects*), δηλαδή συγκεκριμένα στιγμιότυπα κάποιου τύπου δεδομένων (π.χ. `'3`, `'(symbol1 symbol2)` κ.ο.κ.), τα οποία μπορούν να προκύπτουν και ως αποτέλεσμα της κλήσης κάποιας άλλης συνάρτησης. Για να το καταλάβετε καλύτερα τρέξτε στο Listener τα ακόλουθα:

1. `(cons '1 '(2 3 4))`
2. `(cons (- 2 1) '(2 3 4))`
3. `(cons '1 (cons '2 (cons '3 (cons '4 '()))))`

Παρατηρήστε ότι στο Παράδειγμα 2, το πρώτο όρισμα της `cons` είναι η τιμή την οποία επιστρέφει η `-` , με ορίσματα 2, 1. Φυσικά, η λίστα `(- 2 1)` δεν είναι quoted γιατί θέλουμε η Lisp να εντοπίσει τη συνάρτηση `-` και, χρησιμοποιώντας την, να αποτιμήσει την έκφραση-λίστα. Τρέξτε ξανά το 2, έχοντας κάνει quoted τη λίστα

[7] Εντάξει, κι εγώ ο ίδιος θα του πρότεινα να την ξαναφτιάξει απ' την αρχή, αλλά έτσι δε θα μάθουμε ποτέ πως να χειριζόμαστε τις λίστες.

`(- 2 1)`, τοποθετήστε δηλαδή μία απόστροφο πριν την αριστερή παρένθεση αυτής. Προσέξτε τί επιστρέφει η `cons` και προσπαθήστε με βάση αυτά που ξέρετε ήδη να εξηγήσετε το αποτέλεσμα.
Αν το Παράδειγμα 3 συνιστά για εσάς μπέρδεμα, σκεφτείτε το ως εξής: ''Η `(cons '4 '())` δημιουργεί τη νέα λίστα (4), άρα η `(cons '3 '(cons '4 '()))` είναι τελικά ίδια με την `(cons '3 '(4))` απ' όπου προκύπτει η λίστα `(3 4)` κ.ο.κ. μέχρι να φτάσουμε στο `'1`''. Να εκτελέσετε το συλλογισμό αυτό στο Listener για να επιβεβαιώσετε τα επιμέρους αποτελέσματα.

- Να κατασκευάσετε τη λίστα του φοιτητή που αναφέραμε προηγουμένως, με χρήση της `cons` (να το κάνετε με τουλάχιστον δύο διαφορετικούς τρόπους).

Οι ακόλουθες ασκήσεις στοχεύουν στο να εξοικειωθείτε με κάποιες όχι τόσο εμφανείς πτυχές της `cons`:

1. Τρέξτε στο Listener τον τύπο[8]:

   ```
   (cons '() (artificial-intelligence))
   ```

   Διαβάστε το *error message*· θα σας βοηθήσει να καταλάβετε ποιό είναι το λάθος και πώς να το διορθώσετε. Αφού διορθώσετε και τρέξετε σωστά τον παραπάνω τύπο, προσπαθήστε να εξηγήσετε πώς δικαιολογείται αυτό που επιστρέφει η Lisp. Πώς θα μπορέσουμε τελικά να κατασκευάσουμε τη λίστα `(artificial-intelligence)` με χρήση της `cons`; Αφού μπορούμε πολύ εύκολα να κατασκευάσουμε οποιαδήποτε λίστα με τη μορφή δεδομένων, αρκεί να την κάνουμε quoted, για πιο λόγο πιστεύεται θεωρείται τόσο σημαντική η συνάρτηση `cons`; Πιστεύετε ότι προσφέρει κάτι επιπλέον;

2. Πώς απαντάει η Lisp, εάν δώσουμε ως δεύτερο όρισμα στην `cons` ένα σύμβολο αντί για μία λίστα; Δοκιμάστε για παράδειγμα τον τύπο:

   ```
   (cons 'theory-of-computation 'artificial-intelligence)
   ```

   Αν όλα πήγαν καλά, η Lisp θα έχει απαντήσει με ένα *dotted pair*, δηλαδή ένα ζεύγος αντικειμένων που χωρίζονται από μία τελεία. Το *dotted pair* είναι ένας ακόμα τύπος δεδομένων της Lisp που, όμως, χρησιμοποιείται σπανίως και για το λόγο αυτό δε θα πούμε περισσότερα. Το αναφέρουμε μόνο για να το αναγνωρίζετε, καθώς αν ποτέ το συναντήσετε κατά πάσα πιθανότητα κάπου θα έχετε κάνει λάθος.

Όπως είδαμε, η συνάρτηση `cons` κατασκευάζει μία λίστα από ένα οποιοδήποτε αντικείμενο και μία λίστα. Μπορούμε να επαναφέρουμε τα δύο αυτά ορίσματα απ' τη νέα λίστα χρησιμοποιώντας τις συναρτήσεις `FIRST` και `REST`, οι οποίες ορίζονται ως εξής:

`(first` *`list`*`)` Παίρνει σαν όρισμα μία λίστα και επιστρέφει το πρώτο της στοιχείο. Αν η λίστα-όρισμα είναι η `()`, τότε επιστρέφει `NIL`.

`(rest` *`list`*`)` Παίρνει σαν όρισμα μία λίστα και την επιστρέφει χωρίς το πρώτο της στοιχείο. Αν η λίστα-όρισμα είναι η `()`, τότε επιστρέφει `NIL`.

[8] Από εδώ και στο εξής, όταν ενδιαφερόμαστε για την τιμή μιας έκφρασης, θα καλούμε την έκφραση αυτή *τύπο* (*form*).

Η παρακάτω βηματική εκτέλεση θα σας βοηθήσει να εξοικειωθείτε με τη χρήση των συναρτήσεων αυτών:

- Δημιουργήστε με χρήση της `cons` τη λίστα `(a b c d e f)`. Μπορούμε να το κάνουμε είτε δίνοντας απ' ευθείας τη λίστα quoted, είτε περνώντας τον παραπάνω τύπο της `cons` ως όρισμα στην `first`, είτε αναθέτοντας πρώτα τη λίστα σε κάποια μεταβλητή. Προσέξτε ότι αν δώσετε quoted τον τύπο `(cons 'a '(b c d e f))` (βάζοντάς του, δηλαδή, μία απόστροφο πριν την πρώτη από αριστερά παρένθεση) στη `first`, παύει να είναι τύπος, μετατρέπεται σε δεδομένα και το αποτέλεσμα της `first` θα σας φέρει προ εκπλήξεως. Αν δεν είναι προφανές, δοκιμάστε το για να δείτε τί ακριβώς συμβαίνει[9]. Ο σωστός τύπος, που αποτιμάται στο πρώτο στοιχείο της λίστας `(a b c d e f)` είναι ο

  ```
  (first (cons 'a '(b c d e f)))
  ```

  Τρέξτε τον για να το διαπιστώσετε. Εν συνεχεία, βάλτε τη `rest` στη θέση της `first` και ζητήστε απ' τη Lisp να αποτιμήσει το νέο τύπο. Αν το κάνατε σωστά, θα πρέπει να βλέπετε ήδη τη λίστα `(b c d e f)`, δηλαδή τη λίστα με την οποία ξεκινήσαμε, χωρίς το πρώτο της στοιχείο. Αν καταφέρετε να κάνετε τη Lisp να εκτυπώσει το στοιχείο `b` χρησιμοποιώντας μία `first`, μία `rest` και μία `cons`, τότε είστε απολύτως έτοιμοι για να συνεχίσετε. Αν όχι, μην απογοητεύεστε· μετά το πέρας της παρούσας εργαστηριακής άσκησης θα είστε σε θέση να εκτελέσετε πολύ πιο σύνθετους χειρισμούς απ' αυτόν που μόλις ζητήθηκε.

Η Lisp πέρα από τη `first` παρέχει και τις `second`, `third` ... `tenth`, με προφανείς λειτουργίες[10].

- Να αποτιμήσετε τους ακόλουθους τύπους στο χαρτί (προσοχή, κάποιοι μπορεί να προκαλούν σφάλμα) και εν συνεχεία να ελέγξετε την ορθότητα των επιλογών σας τρέχοντάς τους στο Listener (μην παραλείψετε το πρώτο βήμα· είναι το πιο σημαντικό):

  1. `(first '(1 2 3))`
  2. `(first (1 2 3))`
  3. `(rest '(1 2 3))`
  4. `(rest '((1 2) 3))`
  5. `(second '(cons 'a '(a b c)))`
  6. `(first '((a b) (c d)))`
  7. `(rest '((a b) (c d)))`
  8. `(rest (rest '(a b c d)))`
  9. `(rest '(rest (a b c d)))`

[9] Δεν έχουμε πρόθεση να σας παιδέψουμε, ζητώντας σας να τρέχετε ακόμα και τα σφάλματα. Αντίθετα, πρέπει να έχετε πάντα στο μυαλό σας ότι ο **μόνος** τρόπος για να εξοικειωθείτε με οποιαδήποτε γλώσσα προγραμματισμού είναι η συνεχής γραφή προγραμμάτων και ο πειραματισμός με όλα τα πρέπει και τα μη της γλώσσσας.

[10] Εισήχθησαν στη Lisp προς αντικατάσταση των παλαιότερων `CAR` και `CDR`. Επειδή δε θα αναφερθούμε καθόλου σε αυτές, περισσότερες πληροφορίες μπορείτε να βρείτε στις σημειώσεις του μαθήματος.

10. `(first (rest (rest (rest (rest (rest '(symbol1 symbol2 symbol3 symbol4 symbol5 not-a-symbol)))))))`
11. `'(1r 2r 3r)`
12. `(third (rest (rest '(-> <- 1r 2r 3r))))`
13. `(cons '() '(a b c))`
14. `(cons '(a b c) '())`
15. `(cons '(a b c) '())`
16. `(rest '(a))`
17. `(cons 'NIL 'NIL)`

Η συνάρτηση `EQUAL` ελέγχει την ισότητα δύο λιστών. Θα μπορούσατε για παράδειγμα να τη χρησιμοποιήσετε για να ελέγξετε μία λίστα που δεν είστε σίγουροι αν εκφράζει αυτό που πιστεύετε ή σε περιπτώσεις που θέλετε να ελέγξετε πότε μία λίστα θα έρθει σε κάποια συγκεκριμένη μορφή.

- Να αποτιμήσετε του τύπους:

  1. `(equal '(a (b c) d) '(a (b c) d))`
  2. `(equal '(a (b c) d) '(a b c d))`
  3. `(equal '(a) '((a)))`
  4. `(equal '(a) (first '((a))))`

Υπάρχει επίσης μία άλλη συνάρτηση, η `length`, η οποία επιστρέφει το πλήθος των στοιχείων μίας λίστας.

- Σκεφτείτε και εν συνεχεία ελέγξτε τί τιμή θα επιστρέψει η Lisp για τους ακόλουθους τύπους:

  1. `(length '(1 2 3))`
  2. `(length (1 2 3))`
  3. `(length '(a b c (d)))`
  4. `(length '(this is a list))`
  5. `(length '((this is a list within an other list)))`
  6. `(length '(((a b) (c d) (e f))))`
  7. `(length '(() () ()))`
  8. `(length '())`

Ας δούμε και μία ανακεφαλαιωτική άσκηση:

- Να αναθέσετε στη μεταβλητή `long-list` τη λίστα `(a b c d e f g h i)`. Εν συνεχεία χρησιμοποιώντας τη `long-list` και όχι την ίδια τη λίστα, να αναθέσετε στις μεταβλητές `head`, `tail` το πρώτο και τα υπόλοιπα στοιχεία της λίστας, αντίστοιχα. Να χρησιμοποιήσετε την `cons` για να κατασκευάσετε την αρχική λίστα απ' τις `head`, `tail` και να αναθέσετε το αποτέλεσμα στη μεταβλητή `ReSuLt`. Να συγκρίνεται τέλος, αν οι λίστες που περιέχουν οι μεταβλητές `long-list` και `ReSuLt` είναι ίσες, με χρήση της `equal`.

Δύο ακόμα συναρτήσεις με τις οποίες μπορούμε να κατασκευάζουμε λίστες είναι η `APPEND` και η `LIST`.

(`append` $list_1$ $list_2$ $\ldots list_n$) Παίρνει σαν ορίσματα n λίστες και επιστρέφει μία νέα λίστα που έχει για στοιχεία τα στοιχεία των n λιστών.

(`list` $element_1$ $element_2$ $\ldots element_n$) Παίρνει σαν ορίσματα n στοιχεία και επιστρέφει μία λίστα που έχει για στοιχεία τα στοιχεία αυτά (τα στοιχεία μπορεί να είναι οποιαδήποτε *s-expression*).

- Να αποτιμήσετε τους ακόλουθους τύπους (όπου προκύπτει σφάλμα να διαβάσετε το αντίστοιχο μήνυμα):

  1. `(append '(a b c) '(x y z))`
  2. `(append 'a '(x y))`
  3. `(append '(a b) 'x)`
  4. `(list 'a 'b 'c)`
  5. `(list '(1 2 3) '4 '5)`
  6. `(list (append '(1 2 3) '()) '4 '5)`
  7. `(append '(a b) '(c d) '(e f))`

- Να συγκρίνετε τους ακόλουθους τύπους:

  1. `(cons '1 '(2 3))`
  2. `(append '(1) '(2 3))`
  3. `(list '1 '2 '3)`

- Αφού εκτελέσετε `(setf ab-list '(a b) cd-list '(c d))` να συγκρίνετε και τους ακόλουθους:

  1. `(cons ab-list cd-list)`
  2. `(append ab-list cd-list)`
  3. `(list ab-list cd-list)`

Ασκήσεις κατανόησης:

1. Να αποτιμήσετε τους ακόλουθους τύπους (πρώτα στο χαρτί):

   ```
   (append '(a b c) '())
   (list '(a b c) '())
   (cons '(a b c) '())
   ```

2. Ακολουθεί ένας διάλογος με τη Lisp. Προσπαθήστε να βρείτε τί τιμές επιστρέφει σε κάθε βήμα η Lisp (να σημειώσετε τις απαντήσεις δίπλα σε κάθε τύπο):

   ```
   > (setf tools (list 'hammer 'screwdriver))
   > (cons 'pliers tools)
   > tools
   > (setf tools (cons 'pliers tools))
   ```

```
> tools
> (setf tools (cons 'pliers tools))
> tools
> (setf tools (remove 'pliers tools))
> tools
> (append '(pliers wrench) tools)
> tools
> (setf tools (append '(pliers wrench) tools))
> tools
```

3. Προσπαθήστε να αποτιμήσετε τον ακόλουθο τύπο:

```
(cons (first nil) (rest nil))
```