Κεφάλαιο 3

Εργαστηριακή Άσκηση 3

Όπου θα δούμε τις λογικές συναρτήσεις και θα εμβαθύνουμε λίγο περισσότερο στις λίστες και τις μεταβλητές.

3.1 Λογικές Συναρτήσεις

Οι *λογικές συναρτήσεις* (*logical* ή *boolean functions*[1]) είναι οι συναρτήσεις εκείνες των οποίων οι είσοδοι και οι έξοδοι είναι τιμές αλήθειας, δηλαδή `T` ή `NIL`. Πιο τυπικά, μια λογική συνάρτηση είναι μια αντιστοιχία $f : B^k \rightarrow B$, με το B^k εννοούμε

$$\underbrace{B \times B \times \ldots \times B}_{k \text{ φορές}}$$

, όπου $B = \{T, NIL\}$ και $k : k \in \mathbb{Z}, k \geq 0$ (επομένως, $k \in \{0\} \cup \mathbb{Z}^+$) είναι η *τάξη* (*arity*) της συνάρτησης και εκφράζει το πλήθος των ορισμάτων της (πλήθος συνόλων που συμμετέχουν στο καρτεσιανό γινόμενο που ορίζει το πεδίο ορισμού της). Επομένως, αν θεωρήσουμε για f την `and` τριών εισόδων (έχει τάξη ίση με τρία), θα μπορούσαμε να πούμε ότι $f(T, T, T) = T$, ενώ $f(T, T, NIL) = NIL$, αφού για την `and` ισχύει ότι εάν έστω και μία είσοδος είναι `NIL`, δηλαδή `False`, τότε και η έξοδος είναι `NIL`. Για να το καταλάβετε, αρκεί να τις αντιμετωπίσετε όπως τις λογικές συναρτήσεις που χρησιμοποιούμε σε τομείς όπως η Λογική Σχεδίαση, με τη μόνη διαφορά ότι αντί για 0, 1 εδώ χρησιμοποιούμε `NIL`, `T`, αντίστοιχα[2]. Βέβαια, δεν πρέπει να ξεχνάμε την *prefix notation* που έχουμε ήδη δει ότι είναι πανταχού παρούσα στη Lisp. Στη Lisp, θα καλούσαμε την παραπάνω λογική συνάρτηση ως `(and T T T)` και `(and T T NIL)` για τα αντίστοιχα παραδείγματα. Να τα καλέσετε κι εσείς για να αρχίσετε να εξοικειώνεστε.

Οι βασικές λογικές συναρτήσεις είναι η `and` , η `or` και η `not`. Με χρήση αυτών μπορείτε να υλοποιήσετε και όλες τις υπόλοιπες γνωστές λογικές συναρτήσεις όπως η `nand` και η `xor`. Για παράδειγμα:

[1] Προς τιμήν ενός Βρετανού μαθηματικού, εφευρέτη της Boolean Algebra, του George Boole που έζησε το 19ο αιώνα [1815-1864].

[2] Γι' αυτό τις καλούμε τόσο λογικές όσο και boolean συναρτήσεις, αφού και στη Λογική Σχεδίαση οι δυαδικές τιμές 0, 1 παίζουν το ρόλο των τιμών ψεύδους και αλήθειας, αντίστοιχα και όχι το συνήθη αριθμητικό τους ρόλο. Οπότε, να έχετε υπ' όψιν σας ότι δεν υπάρχει καμία ουσιαστική διαφορά.

$nand(x, y) = \neg(x \wedge y)$
$xor(x, y) = (x \wedge \neg y) \vee (y \wedge \neg x)$

- Να ορίσετε τις λογικές συναρτήσεις `nand`, `xor` και `nor` (not or), χρησιμοποιώντας τις τρεις βασικές λογικές συναρτήσεις (όποιες χρειάζονται κάθε φορά). Να τις τρέξετε με διάφορες εισόδους για να επιβεβαιώσετε την ορθή λειτουργία τους (θα σας βοηθούσε να κατασκευάσετε τους πίνακες αληθείας τους).

- Η λογική συνάρτηση `nand` καλείται *λογικά πλήρης* (*logically complete*), επειδή, παίρνοντας διάφορους συνδυασμούς αυτής, μπορούμε να κατασκευάσουμε όλες τις άλλες λογικές συναρτήσεις. Για παράδειγμα, δοκιμάστε τον ακόλουθο όχι και τόσο συνηθισμένο ορισμό της `not`:

  ```
  (defun not2 (x) (nand x x))
  ```

  και βεβαιωθείτε ότι κατανοείτε τον τρόπο με τον οποίο λειτουργεί.

Οι *νόμοι του DeMorgan* (γνωστοί και ως *θεώρημα του DeMorgan*) είναι κανόνες της *τυπικής λογικής* (*formal logic*) που συσχετίζουν ζεύγη δυαδικών λογικών τελεστών (συγκεκριμένα των `and` και `or` δύο εισόδων), με ένα συστηματικό τρόπο και με τη βοήθεια του λογικού τελεστή `not`. Με αυστηρούς μαθηματικούς όρους το θεώρημα λέει ότι κάθε ένας απ' τους ακόλουθους τύπους είναι λογικά ισοδύναμος με τον διπλανό του και μπορεί να μετασχηματιστεί σε αυτόν (και προς τις δύο κατευθύνσεις):

$\neg(p \vee q) \iff (\neg p) \wedge (\neg q)$
$\neg(p \wedge q) \iff (\neg p) \vee (\neg q)$

- Να ορίσετε δύο συναρτήσεις `check-de-morgan1` και `check-de-morgan2`, οι οποίες θα ελέγχουν την ισχύ των νόμων του DeMorgan (μία για τον καθένα). (*Υπόδειξη*: Θα υλοποιήσετε και τα δύο μέλη κάθε ισοδυναμίας και θα ελέγχετε με την `eql` αν είναι ίδια τα αποτελέσματα που επιστρέφουν).

3.2 Περαιτέρω Χειρισμός Λιστών

3.2.1 PUSH και POP

Είδαμε στο προηγούμενο εργαστήριο ότι μπορούμε να χρησιμοποιήσουμε τις συναρτήσεις `cons`, `append` και `list` για να κατασκευάσουμε λίστες. Βέβαια, οι συναρτήσεις αυτές μπορούν μεν να κατασκευάζουν λίστες, αλλά η λίστα που επιστρέφουν κάθε φορά θα χάνεται, αν δε χρησιμοποιήσουμε τη συνάρτηση `setf` για να την αναθέσουμε σε μία μεταβλητή. Υπάρχουν δύο συναρτήσεις, η `PUSH` και η `POP` οι οποίες μπορούν και να αποθηκεύουν το αποτέλεσμα[3].

`(push` *`new_first_element a_symbol_bound_to_a_list`*`)` Παίρνει σαν ορίσματα ένα στοιχείο και ένα σύμβολο ήδη δεσμευμένο σε μία λίστα. Επιστρέφει μία νέα λίστα ακριβώς όπως και η `cons`, μόνο που επιπλέον αναθέτει τη λίστα αυτή στο σύμβολο (το 2ο όρισμά της).

[3] Μπορεί να χρησιμοποιούνται σπάνια, αλλά είναι καλό να τις έχετε υπ' όψιν σας.

(pop *a_symbol_bound_to_a_list*) Παίρνει σαν ορίσμα ένα σύμβολο ήδη δεσμευμένο σε μία λίστα. Επιστρέφει το πρώτο στοιχείο της λίστας και την υπόλοιπη την αναθέτει στο σύμβολο.

Λόγω των αυτόματων αναθέσεων που πραγματοποιούν οι δύο αυτές συναρτήσεις, είναι απαραίτητο να τους δίνουμε ως όρισμα *ένα σύμβολο που έχει δεσμευτεί σε λίστα* και όχι απλώς μια λίστα, καθώς στη δεύτερη περίπτωση οι συναρτήσεις δε θα βρίσκουν μεταβλητή/σύμβολο για να αναθέσουν το αποτέλεσμα.
Ο ακόλουθος διάλογος θα σας βοηθήσει να εξοικειωθείτε με τις push και pop:

```
> (setf new-front 'a
        list-to-be-changed '(b c))
> (push new-front list-to-be-changed)
> list-to-be-changed
> (pop list-to-be-changed)
> list-to-be-changed
```

3.2.2 REVERSE

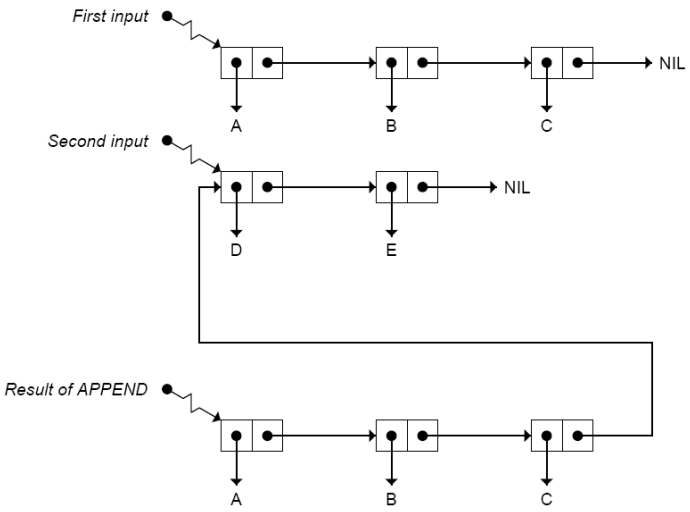
Η συνάρτηση REVERSE ορίζεται ως εξής:

(reverse *a_list*) Παίρνει για όρισμα μία λίστα και επιστρέφει τη λίστα ανεστραμμένη.

- Να δοκιμάσετε τα ακόλουθα παραδείγματα:

  1. `(reverse '(one two three four five)`
  2. `(reverse '(e v i l))`
  3. ```
     (reverse '((first list)
                (second list)
                (third list)))
     ```
  4. `(reverse 'live)`
  5. `(reverse '(one-element-cannot-be-reversed))`
  6. ```
     > (setf a-list '(a b c d))
     > (reverse a-list)
     > a-list
     > (defun store-reverse (x)
         (setf a-list (reverse x)))
     > (store-reverse '(a b c d))
     > a-list
     ```

Απ' το Παράδειγμα 6 αξίζει να παρατηρήσουμε ότι η reverse δεν αλλάζει την τιμή καμίας μεταβλητής ούτε τροποποιεί κάποιο απ' τα cons cells που αναπαριστούν τη λίστα εσωτερικά. Για το λόγο αυτό καλείται *nondestructive function* (και η append για τον ίδιο λόγο είναι μία nondestructive function). Για να καταλάβουμε τί πραγματικά γίνεται εσωτερικά, ας εξετάσουμε λίγο την append. Όταν της δώσουμε σαν ορίσματα τις λίστες (a b c) και (d e), αντιγράφει την πρώτη είσοδο, αλλά όχι τη δεύτερη. Εν συνεχεία, κάνει τη δεύτερη θέση μνήμης του τελευταίου κελιού της λίστας που αντέγραψε να δείχνει στην πρώτη θέση μνήμης του πρώτου κελιού της λίστας που δεν αντέγραψε και που ήταν το δεύτερο όρισμά της (Μελετήστε το Σχήμα 3.1 για να το καταλάβετε). Μετά απ' αυτή τη συζήτηση είμαστε



Σχήμα 3.1: Η εσωτερική αναπαράσταση του τύπου `(append '(a b c) '(d e))`. *Παρατηρήστε το κάτω αντίγραφο της πρώτης εισόδου. Η δεύτερη θέση μνήμης του τελευταίου του κελιού δείχνει στην πρώτη θέση μνήμης του πρώτου κελιού της δεύτερης εισόδου (που δεν έχει αντιγραφεί αλλά είναι η αρχική). Έτσι δημιουργείται μία νέα λίστα (αλυσίδα από cons cells) που ξεκινάει απ' το κάτω* `A` *και καταλήγει στο* `NIL`*. Αυτή δεν είναι άλλη απ' τη λίστα* `(a b c d e)` *.*

σε θέση να κατανοήσουμε το λόγο για τον οποίο ο τύπος `(append 'a '(b c d))` προκαλεί σφάλμα, ενώ αντίθετα ο `(append '(b c d) 'a)` δεν προκαλεί αλλά επιστρέφει `(b c d . a)`. Στην πρώτη περίπτωση, η `append` προσπαθώντας να αντιγράψει τα cons cells που συνθέτουν το πρώτο όρισμά της, αποτυγχάνει, και αυτό συμβαίνει διότι το πρώτο όρισμά της δεν είναι λίστα, επομένως δεν συντίθεται από cons cells. Αντίθετα, στη δεύτερη περίπτωση το πρώτο όρισμα είναι λίστα επομένως αντιγράφεται κανονικά και το *dotted pair* `(b c d . a)` προκύπτει απλώς διότι η δεύτερη θέση μνήμης του τελευταίου κελιού της αντεγραμμένης λίστας αντί να δείχνει σε κάποιο (cons cell) το οποίο να δείχνει με τη σειρά του στο `NIL` (αυτή θα ήταν η φυσιολογική περίπτωση του τύπου `(append '(b c d) '(a))`), δείχνει απ' ευθείας σε ένα σύμβολο.

- Να ορίσετε μία δική σας συνάρτηση, ονόματι `add-to-end`, η οποία θα έχει ως εξής:

  `(add-to-end` ***`element list`***`)` Παίρνει σαν ορίσματα ένα στοιχείο και μία λίστα και επιστρέφει τη λίστα έχοντας προσθέσει το στοιχείο στο τέλος της (λέγοντας στοιχείο εννοούμε οποιαδήποτε s-expression).

  και για να το κάνει αυτό θα χρησιμοποιεί υποχρεωτικά τη συνάρτηση `reverse`, καθώς και όποια συνάρτηση κατασκευής λιστών θεωρείτε ότι ταιριάζει. Αφού την υλοποιήσετε με αυτό τον τρόπο, υλοποιήστε τη με μία μόνο `append`. Ο τελευταίος τρόπος είναι και ο πιο αποδοτικός, αφού δημιουργεί μόνο ένα αντίγραφο της λίστας (εσωτερικά), ενώ ο προηγούμενος δημιουργεί δύο.

3.2.3 NTH και NTHCDR

Η συνάρτηση NTHCDR επιστρέφει το επίθεμα n μίας λίστας, δηλαδή το επίθεμα εκείνο της λίστας που ξεκινάει απ' τη θέση n. Το επίθεμα 0 είναι όλη η λίστα, το 1 είναι η λίστα χωρίς το 1ο της στοιχείο, ενώ το επίθεμα n είναι η λίστα χωρίς τα πρώτα της n στοιχεία. Μπορείτε να το καταλάβετε ακόμα πιο καλά, αν σκεφτείτε τη λίστα ως cons cells και παρατηρήσετε ότι η `nthcdr` σας μεταφέρει στη δεύτερη θέση μνήμης του n-οστού cons cell που δεν είναι τίποτα άλλο από ένας δείκτης που δείχνει στην πρώτη θέση μνήμης του cons cell υπ' αριθμόν $n+1$. Επομένως αν την καλέσετε με πρώτο όρισμα τον αριθμό 1, σας πάει στον δείκτη του πρώτου cons cell , ο οποίος δείχνει στα περιεχόμενα του δεύτερου cons cell που δεν είναι άλλα απ' το δεύτερο στοιχείο της λίστας. Για το λόγο αυτό σας επιστρέφεται η λίστα που είχατε δώσει ως δεύτερο όρισμα στη `nthcdr` από το δεύτερο στοιχείο της και μετά, χωρίς δηλαδή το πρώτο της στοιχείο. Η συνάρτηση ορίζεται ως εξής:

(`nthcdr` *`a_number a_list`*) Παίρνει σαν ορίσματα έναν αριθμό, έστω n, και μία λίστα και επιστρέφει το επίθεμα n της λίστας.

- Να καλέσετε την `nthcdr` με δεύτερο όρισμα μία λίστα n στοιχείων (το n της επιλογής σας) και πρώτο όρισμα τον αριθμό n (π.χ. μία λίστα 5 στοιχείων και τον αριθμό 5). Πώς εξηγείτε το αποτέλεσμα με βάση τα όσα είπαμε για τον τρόπο με τον οποίο λειτουργεί η `nthcdr`;

- Να αποτιμήσετε τους ακόλουθους τύπους:

  1. `(nthcdr 0 '(a b c d e))`
  2. `(nthcdr 1 '(a b c d e))`
  3. `(nthcdr 4 '(a b c d e))`
  4. `(nthcdr 5 '(a b c d e))`
  5. `(nthcdr 9 '(a b c d e))`

- Στο ακόλουθο παράδειγμα η λίστα που δίνουμε σαν όρισμα είναι ένα *dotted pair*, επομένως, ο τελευταίος δείκτης στην αλυσίδα από cons cells που την αναπαριστά δείχνει σε ένα άτομο και όχι στο `NIL`. Να το τρέξετε για να δείτε τί αποτέλεσμα μας επιστρέφει η Lisp.

```
(nthcdr 4 '(a b c .  d))
```

Η συνάρτηση NTH επιστρέφει το πρώτο στοιχείο του n επιθέματος μιας λίστας.

(`nth` *`a_number a_list`*) Παίρνει σαν ορίσματα έναν αριθμό, έστω n, και μία λίστα και επιστρέφει το πρώτο στοιχείο του επιθέματος n της λίστας. Πιο απλά, μπορούμε να πούμε ότι επιστρέφει το πρώτο στοιχείο της λίστας που θα επέστρεφε η `nthcdr`, αν είχε κληθεί με τα ίδια ορίσματα.

Μετά την παραπάνω συζήτηση μπορούμε εύκολα να ορίσουμε μόνοι μας την `my-nth`, τη δική μας δηλαδή εκδοχή για την `nth`, ως εξής:

```
(defun my-nth (n x)
    "My version of nth function"
    (first (nthcdr n x)))
```

- Αφού ορίσετε την `my-nth` στο Listener, να αποτιμήσετε τους ακόλουθους τύπους (στους δύο τελευταίους να διαβάσετε προσεκτικά τα μηνύματα των σφαλμάτων):

  1. `(my-nth 0 '(a b c d))`
  2. `(nth 0 '(a b c d))`
  3. `(my-nth 3 '(a b c d))`
  4. `(nth 3 '(a b c d))`
  5. `(my-nth 5 '(a b c d))`
  6. `(nth 5 '(a b c d))`
  7. `(my-nth 2 'a)`
  8. `(nth 2 'a)`

3.2.4 BUTLAST

Η συνάρτηση `BUTLAST` παίρνει σαν όρισμα μία λίστα και επιστρέφει τη λίστα χωρίς το τελευταίο της στοιχείο.

`(butlast` *a_list*`)` Παίρνει σαν όρισμα μία λίστα και επιστρέφει τη λίστα χωρίς το τελευταίο της στοιχείο.

- Δοκιμάστε τους ακόλουθους τύπους:

  1. `(butlast '(a b c 1 2 3 this-will-be-removed))`
  2. `(butlast '(list))`
  3. `(butlast '((list)))`
  4. `(butlast '())`
  5. `(butlast (nthcdr 3 '(a b c 1 2 3 this-will-be-removed)))`

3.2.5 LAST

Η `LAST` παίρνει σαν όρισμα μία λίστα και επιστρέφει μια νέα λίστα που περιέχει μόνο το τελευταίο στοιχείο της λίστας που πήρε σαν όρισμα. Για να είμαστε πιο ακριβείς, μπορούμε να πούμε ότι επιστρέφει το cons cell του οποίου το `car` (αριστερή θέση μνήμης κελιού) είναι το τελευταίο στοιχείο της λίστας.

`(last` *a_list*`)` Παίρνει σαν όρισμα μία λίστα και επιστρέφει μια νέα λίστα με ένα μόνο στοιχείο, που είναι το τελευταίο στοιχείο της λίστας την οποία πήρε ως όρισμα.

- Τί τιμή επιστρέφουν οι παρακάτω τύποι;

  1. `(last '(this is a list))`
  2. `(last '(list))`
  3. `(last '())`
  4. `(last nil)`
  5. `(last '(a b c .  d))`
  6. `(last '((this is strange-can you find out why?)))`
  7. `(last 'Produces-an-error-Read-the-error-message)`

3.2.6 REMOVE

Τη REMOVE την έχουμε ήδη δει. Αυτό που κάνει είναι να αφαιρεί ένα στοιχείο από μία λίστα και μάλιστα να αφαιρεί όλες τις εμφανίσεις του στοιχείου από τη λίστα αυτή[4]. Το αποτέλεσμα που επιστρέφεται απ' τη **remove** είναι μία νέα λίστα που δεν περιέχει τα διεγραμμένα στοιχεία.

(remove *element_to_remove a_list*) Παίρνει σαν όρισμα μία λίστα και ένα στοιχείο και διαγράφει όλες τις εμφανίσεις του στοιχείου από τη λίστα επιστρέφοντας με μία νέα λίστα το αποτέλεσμα.

- Να αποτιμήσετε τους παρακάτω τύπους:

  1. `(remove 'e '(E v e r y o c c u r e n c e o f t h e l e t t e r e w i l l b e l o s t))`
  2. `(remove 'f '(r f e v f e a f l f e f d))`
  3. `(remove '1 '(3 1 4 1 5 9))`
  4. `(remove 'nothing-left '(nothing-left))`
  5. `(remove 'everything-left '((everything-left)))`

- Ο ακόλουθος διάλογος με τη Lisp θα σας βοηθήσει να καταλάβετε γιατί λέμε ότι και η **remove** είναι μία nondestructive συνάρτηση:

```
> (setf bounded-symbol '(some letters will
    be temporaly removed))
> (remove 'e bounded-symbol)
> bounded-symbol
```

Οι ακόλουθες ασκήσεις θα σας βοηθήσουν να εξοικειωθείτε ακόμα περισσότερο με το χειρισμό των λιστών:

1. Να ορίσετε μία συνάρτηση, ονόματι **change-first**, η οποία θα αλλάζει το πρώτο στοιχείο μιας λίστας με ένα στοιχείο της επιλογής σας. Θα παίρνει επομένως δύο ορίσματα, ένα στοιχείο και μια λίστα και θα αλλάζει το πρώτο στοιχείο της λίστας με το στοιχείο που δώσατε ως όρισμα.

2. Προσπαθήστε, χωρίς να την τρέξετε, να καταλάβετε τί κάνει η ακόλουθη συνάρτηση:

```
(defun change-n (x n y)
  (append (reverse
            (cons x
                  (nthcdr (- (length y) (- n 1))
                          (reverse y))))
          (nthcdr n y)))
```

   Εν συνεχεία, για να την κατανοήσετε καλύτερα, να τρέξετε κάθε τύπο που την αποτελεί ξεχωριστά. Τέλος, τρέξτε τη συνάρτηση για διάφορετικές εισόδους και παρατηρήστε τα αποτελέσματα. Κάνει τελικά αυτό που αναμένατε;

[4] Θα δούμε αργότερα ότι μπορούμε να της ζητήσουμε να αφαιρέσει μόνο ορισμένες εμφανίσεις.

3. Να ορίσετε μία συνάρτηση, ονόματι `my-butlast`, η οποία θα είναι η δική σας εκδοχή της συνάρτησης `butlast`. Θα παίρνει δηλαδή σαν όρισμα μία λίστα και θα την επιστρέφει χωρίς το τελευταίο της στοιχείο.

4. Να ορίσετε μία συνάρτηση, ονόματι `palindrome`, η οποία θα ανιχνεύει τις *παλίνδρομες* λίστες. Μία παλίνδρομη λίστα είναι μια λίστα τις οποίας τα στοιχεία διαβάζονται το ίδιο τόσο από αριστερά τόσο και από δεξιά. Για παράδειγμα, η λίστες `(a b c d c b a)` και `(hello world central-point world hello)` είναι παλίνδρομες, ενώ η λίστα `(a b c a b c)` δεν είναι. Επομένως, η συνάρτησή σας θα πρέπει να παίρνει σαν όρισμα μία λίστα και να απαντάει με `T` αν η λίστα είναι παλίνδρομη, ενώ αντίθετα με `NIL` εάν δεν είναι.

5. Ποιά primitive συνάρτηση θεωρείτε ότι ορίζει η παρακάτω συνάρτηση;

```
(defun unknown-function (x)
  (first (last (reverse x))))
```

3.3 Οι λίστες ως σύνολα

Ως γνωστών, ένα *σύνολο* (*set*) είναι μία συλλογή ξεχωριστών αντικειμένων που την αντιλαμβανόμαστε ως μία ολότητα. Επομένως, εξ' ορισμού, στα σύνολα δε μας ενδιαφέρει η διάταξη και κάθε αντικείμενο μπορεί να εμφανίζεται μόνο μία φορά. Τα στοιχεία του συνόλου καλούνται *μέλη* (*members*). Οι ημέρες της εβδομάδας, για παράδειγμα, αποτελούν ένα σύνολο το ίδιο και οι ακέραιοι αριθμοί (άπειρο σύνολο). Πρέπει να είναι προφανές ότι οι λίστες δεν είναι σύνολα· έχουμε δει ότι διαφορετικές διατάξεις δημιουργούν διαφορετικές λίστες και ότι τα στοιχεία της λίστας μπορούν να επαναλαμβάνονται. Βέβαια, αυτό δεν μας απαγορεύει να χρησιμοποιήσουμε τις λίστες για να κατασκευάζουμε σύνολα.

Αναμφίβολα, τα σύνολα είναι ένας απ' τους πιο χρήσιμους τύπους δεδομένων που μπορεί να κατασκευάσει κάποιος χρησιμοποιώντας τις λίστες. Οι βασικές πράξεις που μπορούμε να εκτελέσουμε σε ένα σύνολο είναι ο έλεγχος για το εάν ένα αντικείμενο είναι μέλος του συνόλου (`member`), η ένωση δύο συνόλων (`union`), η τομή τους (`intersection`), η διαφορά τους (`set-difference`) και τέλος ο έλεγχος του εάν ένα σύνολο είναι υποσύνολο κάποιου άλλου συνόλου (`subsetp`). Στη συνέχεια περιγράφουμε τις primitive συναρτήσεις της Lisp για τις παραπάνω πράξεις.

3.3.1 MEMBER

Η `MEMBER` είναι στην ουσία ένα *κατηγόρημα* (*predicate*), το οποίο ελέγχει εάν ένα αντικείμενο είναι στοιχείο μιας λίστας. Εάν το αντικείμενο βρεθεί μες στη λίστα, τότε επιστρέφεται το επίθεμα της λίστας που ξεκινάει απ' το αντικείμενο αυτό. Εάν το αντικείμενο εμφανίζεται περισσότερες από μία φορές, περίπτωση κατά την οποία μιλάμε αμιγώς για λίστα και όχι για σύνολο, τότε επιστρέφεται το επίθεμα που ξεκινάει απ' την πρώτη εμφάνιση του αντικειμένου (ο έλεγχος γίνεται από αριστερά προς τα δεξιά). Αν δε βρεθεί το αντικείμενο, η `member` επιστρέφει `NIL`[5].

[5] Η `member` δεν είναι ακριβώς κατηγόρημα, καθώς δεν επιστρέφει ποτέ την τιμή `T`. Είθισται, πάντως, να προσμετράται ως κατηγόρημα.

- Να αποτιμήσετε τους ακόλουθους τύπους:

  1. `(member 'y '(a b c d e))`
  2. `(member 1 '(3 b 1 d e))`
  3. `(member 'b (remove 'b '(3 b 1 b d e b b b)))`
  4. `(member 'b (last '(3 b 1 b d e b b b)))`
  5. `(first (member 'b (last '(3 b 1 b d e b b b))))`

- Να ορίσετε μία συνάρτηση `before` η οποία θα καλείται ως εξής:

  `(before` *element*$_1$ *element*$_2$ *a_list*`)` Παίρνει ως ορίσματα δύο στοιχεία και μία λίστα και ελέγχει εάν το πρώτο όρισμα εμφανίζεται πριν το δεύτερο (από αριστερά) στη λίστα.

  Θα πρέπει να χρησιμοποιήσετε δύο φορές τη συνάρτηση `member`. Αφού την ορίσετε να πειραματιστείτε με τη λίστα `(and therefore never send to know for whom the bell tolls-it tolls for thee)`[6].

3.3.2 INTERSECTION

Η συνάρτηση `INTERSECTION` παίρνει ως ορίσματα δύο σύνολα, πάντα υπό την μορφή λιστών, και επιστρέφει την τομή τους, ένα σύνολο, δηλαδή, με τα στοιχεία που εμφανίζονται και στα δύο σύνολα. Η σειρά με την οποία θα επιστραφούν τα στοιχεία είναι ακαθόριστη και εξ' άλλου δε μας ενδιαφέρει, αφού μιλάμε για σύνολα.

- Στα ακόλουθα, να αποτιμήσετε τους τύπους και να απαντήσετε στις ερωτήσεις:

  1. `(intersection '(c++ java lisp prolog) '(c asp php lisp prolog haskell))`
  2. `(intersection '(b c f a) '(c a b d))`
  3. Ποιό είναι το αποτέλεσμα της τομής ενός συνόλου με το `NIL`;
  4. Ποιό είναι το αποτέλεσμα της τομής ενός συνόλου με τον εαυτό του;

3.3.3 UNION

Η συνάρτηση `UNION` επιστρέφει την ένωση δύο συνόλων. Αν ένα αντικείμενο εμφανίζεται και στα δύο σύνολα, στο νέο σύνολο θα εμφανίζεται μόνο μία φορά. Επίσης, η σειρά εμφάνισης των αντικειμένων στο αποτέλεσμα είναι ακαθόριστη.

- Στα ακόλουθα, να αποτιμήσετε τους τύπους και να απαντήσετε στις ερωτήσεις:

  1. ```
     (union '(a b c d)
            (e f g h))
     ```
  2. ```
     (union '(1 3 5 7 9)
            (0 2 4 6 8))
     ```
  3. Ποιά είναι η ένωση ενός συνόλου με το `NIL`;
  4. Ποιά είναι η ένωση ενός συνόλου με τον εαυτό του;

[6] John Donne [Devotions upon Emergent Occasions, XVII, 1624]· αναγράφεται επίσης στην πρώτη σελίδα του μυθιστορήματος ''Για ποιόν χτυπά η καμπάνα'' του Έρνεστ Χέμινγουεϊ.

3.3.4 SET-DIFFERENCE

Η συνάρτηση `SET-DIFFERENCE` επιστρέφει αυτό που στη θεωρία συνόλων καλούμε *σχετικό συμπλήρωμα* (*relative complement* ή *set theoretic difference*) δύο συνόλων. Το σχετικό συμπλήρωμα δύο συνόλων A, B, που γράφεται ως A/B ή $B - A$ είναι ένα σύνολο, του οποίου τα μέλη είναι μέλη του B, αλλά όχι του A. Επομένως, η `set-difference` επιστρέφει ό,τι απομένει στο πρώτο σύνολο αν διαγραφούν απ' αυτό όσα μέλη του περιέχονται και στο δεύτερο σύνολο. Η λέξη 'πρώτο' έχει σημασία, καθώς η `set-difference` υλοποιεί αυστηρά την πράξη $argument_1 - argument_2$, αν τα ορίσματα είναι σύνολα και όχι την πράξη $argument_2 - argument_1$. Τέλος, αξίζει να αναφέρουμε ότι τα στοιχεία επιστρέφονται με αντίστροφη σειρά απ' αυτή που εμφανίζονταν αρχικά στο πρώτο σύνολο, αλλά αφού μιλάμε για σύνολα δεν έχει καμία σημασία.

- Να αποτιμήσετε τους ακόλουθους τύπους:

  1. `(set-difference '(d e f g 1 2 3 4) '(1 2 3 4))`
  2. `(set-difference '(d e f g 1 2 3 4) '(d e f g))`
  3. `(set-difference '(d e f g 1 2 3 4) '(a b c))`
  4. `(set-difference '(unchanged members) '())`
  5. `(set-difference '(everything removed) '(everything removed))`

3.3.5 SUBSETP

Το κατηγόρημα `SUBSETP` επιστρέφει `T`, εάν το σύνολο που του δίνεται ως πρώτο όρισμα είναι υποσύνολο του συνόλου που του δίνεται ως δεύτερο όρισμα. Το αντίστροφο δεν ισχύει. Με πιο απλά λόγια, επιστρέφει `T` εάν και μόνον εάν όλα τα στοιχεία του πρώτου του ορίσματος περιέχονται στο δεύτερό του όρισμα.

- Να αποτιμήσετε τους ακόλουθους τύπους:

  1. `(subsetp '(a b) '(a b c d e))`
  2. `(subsetp '(a b c d e) '(a b))`
  3. `(subsetp '(a b c) '(a b d e f))`

Ακολουθούν δύο γενικές ασκήσεις πάνω στα σύνολα:

1. Λέμε ότι δύο σύνολα A, B είναι ίσα εάν έχουν ακριβώς τα ίδια μέλη, δηλαδή κάθε μέλος του A να είναι μέλος του B και αντίστροφα. Να ορίσετε μία συνάρτηση, ονόματι `set-equality`, η οποία θα παίρνει ως ορίσματα δύο σύνολα και θα επιστρέφει `T` αν τα σύνολα αυτά είναι ίσα και `NIL` σε αντίθετη περίπτωση (*Υποδείξεις*: 1. Αν δύο σύνολα είναι ίσα, τότε το κάθε ένα είναι υποσύνολο του άλλου. 2. Θα χρειαστεί να χρησιμοποιήσετε τη λογική συνάρτηση `and`, π.χ. `(and` $s-expression_1$ $s-expression_2$`)`).

2. Λέμε ότι ένα σύνολο A είναι ένα κανονικό υποσύνολο ενός συνόλου B (γράφουμε $A \subset B$), εάν το A είναι υποσύνολο του B (γράφουμε $A \subseteq B$), αλλά το A δεν είναι ίσο με το B, ή αλλιώς: $A \subset B$, εάν $A \subseteq B$ και $\exists b \in B$, τ.ω. $b \notin A$. Να ορίσετε μία συνάρτηση, ονόματι `proper-subset`, η οποία θα παίρνει ως ορίσματα δύο σύνολα και θα επιστρέφει `T` αν το πρώτο είναι γνήσιο υποσύνολο του δεύτερου, ενώ `NIL` εάν δεν είναι (*Υπόδειξη*: Εδώ θα χρειαστεί να χρησιμοποιήσετε και τη λογική συνάρτηση `not`).

3.4 Περισσότερα για τις μεταβλητές

Ξεκινώντας, ας δούμε δύο ακόμα ορισμούς για τις μεταβλητές, οι οποίοι είναι πιο δόκιμοι ως προς την ορολογία που χρησιμοποιείται στη Lisp και έρχονται να συμπληρώσουν τη συζήτηση σχετικά με τις τοπικές και τις καθολικές μεταβλητές:

- Μία μεταβλητή είναι μία *λεξικολογική μεταβλητή* (*lexical variable*), εάν εμφανίζεται στο σώμα μίας συνάρτησης στην οποία είναι όρισμα/παράμετρος[7].
- Μία μεταβλητή είναι μία *ειδική μεταβλητή* (*special variable*), εάν εμφανίζεται στο σώμα μίας συνάρτησης στην οποία δεν είναι όρισμα.

Εμείς, πάντως, θα συνεχίσουμε να χρησιμοποιούμε τους όρους τοπική και καθολική μεταβλητή, αναφερόμενοι σε λεξικολογικές και ειδικές μεταβλητές, αντίστοιχα.

3.4.1 Η ειδική συνάρτηση LET

Η ειδική συνάρτηση `LET` μπορεί να χρησιμοποιηθεί για τη δημιουργία τοπικών μεταβλητών. Ας δούμε πρώτα ένα παράδειγμα και εν συνεχεία θα πούμε περισσότερα για τον τυπικό τρόπο κλήσης της συνάρτησης αυτής. Έστω, ότι θέλουμε να χρησιμοποιούμε πολύ συχνά το μέσο όρο δύο αριθμών και για να μην είμαστε αναγκασμένοι να ορίζουμε ξανά και ξανά το μέσο όρο, αποφασίζουμε να δημιουργήσουμε μία συνάρτηση η οποία θα κάνει τη δουλειά για εμάς. Σκεφτόμαστε ότι αντί να υπολογίζουμε ένα τύπο της μορφής $(x+y)/2$, θα μπορούσαμε να αναθέτουμε το άθροισμα (αριθμητής) σε μία τοπική μεταβλητή, έστω sum και ακολούθως να υπολογίζουμε τον τύπο $sum/2$. Για να το κάνουμε αυτό μπορούμε να χρησιμοποιήσουμε την ειδική συνάρτηση `let` ως εξής:

```
(defun average (x y)
  (let ((sum (+ x y)))
    (list 'The 'average 'of x 'and y 'is (/ sum 2.0))))
```

Να ορίσετε και να τρέξετε την παραπάνω συνάρτηση. Το σώμα της `average` περιέχει ένα `let` τύπο ο οποίος μπορεί να διαβαστεί ως εξής: ''Έστω ότι το `sum` είναι `(+ x y)`, τότε επέστρεψε `(list 'The 'average 'of x 'and y 'is (/ sum 2.0))`''. Η γενική σύνταξη της `let` είναι η εξής:

```
(let ((variable_1 value_1)
      (variable_2 value_2)
                  ⋮
      (variable_n value_n))
  form_1
    ⋮
  form_m)
```

Αξίζει να παρατηρήσετε ότι το πλήθος των τύπων που ζητάμε να επιστραφούν, δηλαδή το m, είναι ανεξάρτητο απ' το πλήθος των τοπικών μεταβλητών που ορίσαμε, δηλαδή το n. Επίσης, το σύνολο των m τύπων αποτελούν το σώμα της συνάρτησης. Κάθε τοπική μεταβλητή που ορίζουμε και αρχικοποιούμε με τη `let` έχει

[7] Εμείς θα συνεχίσουμε να χρησιμοποιούμε τον όρο *όρισμα*, αλλά καλό είναι να ξέρετε ότι στη βιβλιογραφία αναφέρεται συχνά και ως *παράμετρος συνάρτησης*.

εμβέλεια μόνο μέχρι το τέλος της `let`[8]. Αν αφαιρούσαμε τον τύπο του προηγούμενου παραδείγματος, `(list 'The 'average 'of x 'and y 'is (/ sum 2.0))`, απ' το σώμα της `let` και τον τοποθετούσαμε στο σώμα της `defun`, τότε, όταν θα τρέχαμε τη συνάρτηση για κάποια ορίσματα, θα παίρναμε ένα μήνυμα σφάλματος, αφού η Lisp δε θα μπορούσε να εντοπίσει τιμή για τη μεταβλητή `sum`, μιας και θα είχε κληθεί έξω απ' την εμβέλειά της. Στην πραγματικότητα, αυτό που έγινε είναι ότι, κατά τον ορισμό της συνάρτησης, η Lisp υπέθεσε ότι αναφερόμαστε σε μία ειδική μεταβλητή `sum` την οποία πιθανώς να έχουμε ορίσει κάπου έξω απ' τον εικονικό φράχτη που δημιουργεί η `defun`. Ο λόγος που το υπέθεσε αυτό είναι ότι ούτε μπορεί να δει τη `sum` που ορίσαμε στη `let` γιατί είμαστε έξω απ' την εμβέλειά της, ούτε βρίσκει τη `sum` μεταξύ των ορισμάτων της συνάτησης. Το πιο σημαντικό πράγμα που δε θα πρέπει ποτέ να ξεχνάμε όταν χρησιμοποιούμε τη `let`, είναι ότι αποτιμάει τους *τύπους τιμών* (*value forms*) των n μεταβλητών παράλληλα. Αυτό σημαίνει ότι πρώτα αποτιμάει όλους τους τύπους τιμών και μετά πηγαίνει να αναθέσει τις τιμές αυτές στις αντίστοιχες μεταβλητές.

- Έχοντας κατά νου τον τρόπο με τον οποίο αποτιμάει τους τύπους τιμών η `let`, προσπαθήστε να σκεφτείτε γιατί ο ακόλουθος τύπος προκαλεί ένα σφάλμα:

```
(let ((x 'outside)
      (y x))
  (list x y))
```

- Να ορίσετε μία συνάρτηση `first-last`, η οποία θα παίρνει ως όρισμα μία λίστα, θα χρησιμοποιεί τη `let` για να ορίσει δύο τοπικές μεταβλητές και στο σώμα της θα επιστρέφει, χρησιμοποιώντας τις μεταβήτές που όρισε, μία νέα λίστα που θα περιέχει μόνο το πρώτο και το τελευταίο στοιχείο της λίστας που πήρε ως όρισμα.

3.4.2 Η ειδική συνάρτηση LET*

Είδαμε ότι με τη `let` δε μπορούμε να ορίσουμε και να αρχικοποιήσουμε μία μεταβλητή και εν συνεχεία να χρησιμοποιήσουμε τη μεταβλητή αυτή ως τιμή για μία άλλη μεταβλητή που ορίζουμε με τη `let`. Αυτό συμβαίνει πολύ απλά διότι η `let` αποτιμάει πρώτα όλους τους τύπους τιμών, επομένως όταν προσπαθήσει να βρει τιμή για τη μεταβλητή αυτή, η μεταβλητή δε θα έχει ακόμα πάρει τιμή και θα προκύψει σφάλμα.

Το πρόβλημα αυτό το λύνει η ειδική συνάρτηση `LET*` η οποία αποτιμάει τους τύπους τιμών ακολουθιακά. Πιο συγκεκριμένα, αποτιμάει τον πρώτο τύπο τιμών και την τιμή που βρίσκει την αναθέτει στην πρώτη μεταβλητή, μετά αποτιμάει το δεύτερο τύπο τιμών και την τιμή που βρίσκει την αναθέτει στη δεύτερη μεταβλητή κ.ο.κ. Επομένως, παρατηρούμε ότι το παράδειγμα που δείξαμε προηγουμένως, δεν έχει κανένα λόγο να προκαλεί σφάλμα, αν αντικαταστήσουμε τη `let` με τη `let*`, αφού τώρα η μεταβλητή μας θα έχει πάρει τιμή, όταν τη συναντήσει η Lisp σε έναν άλλο τύπο, και έτσι θα μπορεί να αναθέσει την τιμή της σε κάποια άλλη μεταβλητή.

- Να τρέξετε το παράδειγμα που οδηγούσε σε σφάλμα στην προηγούμενη υποενότητα, αλλάζοντας τη `let` με τη `let*`. Παρατηρείτε κάποια διαφορά;

[8] Πιο σωστά θα λέγαμε ότι η εμβέλειά της είναι ο *εικονικός φράχτης* που ορίζεται από τη `let`.

Μπορείτε να εξηγήσετε γιατί παίρνουμε τα αντίστοιχα αποτελέσματα, τόσο στην προηγούμενη όσο και σε αυτή την περίπτωση;

- Ζητείστε από τη Lisp να αποτιμήσει τις μεταβλητές που ορίσατε μες στο σώμα μιας `let`, αλλά αφού έχετε καλέσει τη `let`, δηλαδή έξω απ' τον εικονικό φράχτη που ορίζει. Καταλαβαίνετε γιατί τις καλούμε τοπικές μεταβλητές;

- Τώρα προσπαθήστε να κάνετε το αντίστροφο. Αναθέστε τιμή σε μία καθολική μεταβλητή με τη βοήθεια της `setf` και μετά ζητείστε από τη `let` να αναθέσει την τιμή της καθολικής μεταβλητής που ορίσατε σε μία άλλη τοπική μεταβλήτη (θα μπορούσατε, για παράδειγμα, να το κάνετε πάνω στο παράδειγμα που είδαμε προηγουμένως) επιτελώντας ταυτόχρονα και κάποιες άλλες λειτουργίες. Τί παρατηρείτε; Βρίσκει η Lisp τιμή για τη μεταβλητή αυτή, τη θεωρεί δηλαδή δεσμευμένη σε κάποια τιμή, ή επιστρέφει μήνυμα σφάλματος; Τί συμπέρασμα βγάζετε για τις καθολικές μεταβλητές;