Τ.Ε.Ι ΠΑΤΡΑΣ

Τμήμα Επιχειρηματικού Σχεδιασμού και Πληροφοριακών Συστημάτων

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ευφυής Προγραμματισμός: Γλώσσες Προγραμματισμού που υποστηρίζουν τον Προγραμματισμό καθώς και Προγραμματιστικά Περιβάλλοντα.

Εισηγητής: **Λουκάς Μάνδαλος**

Σπουδαστές

Επώνυμο: Γιωργαλλέττος

Όνομα : Μάριος

Επώνυμο: Παπαγεωργίου

Όνομα : Γεώργιος

ΠΕΡΙΕΧΟΜΕΝΑ

Σελ

Πρόλογος……………………………………………………………………..6

Εισαγωγή.. ..7

Κεφάλαιο Πρώτο – LISP

1.1 Εισαγωγή στη LISP 8

1.2 Βασικά στοιχεία - Ορισμοί 11

1.3 Ανάγνωση – Εκτίμηση - Εκτύπωση... 12

1.3.1 Κανόνες Εκτίμησης.. 12

1.4 Θεμελιώδεις Συναρτήσεις.. 15

1.5 Συναρτήσεις Χρήστη – Συνάρτηση LET 23

1.6 Ειδικές (Καθολικές) Μεταβλητές.. 25

1.7 Έλεγχος Ροής Προγράμματος.. 26

1.8 Αναδρομή και Παράμετροι Ειδικού Σκοπού............................... 33

1.8.1 Αναδρομικές Συναρτήσεις 33

1.8.2 Παράμετροι Ειδικού Σκοπού 34

1.9 Είσοδος – Έξοδος.. 37

1.10 Συναρτήσεις με Ορίσματα Συναρτήσεις.................................... 39

1.11 Λίστες Ιδιοτήτων – Πίνακες.. 44

1.11.1 Λίστα Ιδιοτήτων 44

1.11.2 Πίνακες.. 45

1.12 Δομές στη LISP 47

Κεφάλαιο Δεύτερο – ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

2.1 Εισαγωγή στην ΤΝ 51

2.2 Ιστορική εξέλιξη της ΤΝ………………………………………………...53

2.3 Συμβατική Τεχνητή Νοημοσύνη 56

2.4 Υπολογιστική Τεχνητή Νοημοσύνη 57

Κεφάλαιο Τρίτο – ΠΡΟΒΛΗΜΑΤΑ ΤΝ ΚΑΙ LISP

3.1 Αναζήτηση και Στρατηγικές Αναζήτησης 58
3.2 Το πρόβλημα των Δύο Δοχείων 63

Κεφάλαιο Τέταρτο – ΤΟ ΕΡΓΑΛΕΙΟ CLIPS

4.1 Περιβάλλον CLIPS 66
4.2 Η Δομή του CLIPS 67
4.3 Η Σύνταξη του CLIPS 68
4.4 Τα Δομικά Στοιχεία 69
4.5 Οι Μεταβλητές 69
4.6 Τα Γεγονότα 70
4.7 Οι Κανόνες 73
4.8 Ταυτοποίηση 75
4.9 Οι Βασικές Εντολές στο Περιβάλλον 77
4.10 Οι Συναρτήσεις 78
4.10.1 Ο Ορισμός Συναρτήσεων στο CLIPS 90
4.11 Οι Περιορισμοί στις Συνθήκες των Κανόνων 91
4.12 Πρότυπα Γεγονότων 95
4.13 Ενεργοποίηση / Απενεργοποίηση Τιμών Ελέγχου 101
4.14 Αλλαγή Τιμής μιας Ιδιότητας 101
4.15 Στρατηγικές Επίλυσης Συγκρούσεων 102
4.16 Προτεραιότητα Κανόνων 103
4.17 Στρατηγικές Επίλυσης Συγκρούσεων 104

Κεφάλαιο Πέμπτο – ΕΜΠΕΙΡΑ ΣΥΣΤΗΜΑΤΑ

5.1 Ορισμός, Δομή και Λειτουργία Ε.Σ. 106
5.2 Βάση Γνώσης 107
5.3 Βάση Εργασίας 107
5.4 Μηχανισμός Εξαγωγής Συμπερασμάτων (ΜΕΣ) 108
5.5 Συνιστώσα Επικοινωνίας Χρήστη 109

5.6 Συνιστώσα Επεξηγήσεων (ΣΕΠ) 109
5.7 Κριτήρια Καταλληλότητας.. 110
5.8 Τύποι Έμπειρων Συστημάτων.. 112
5.9 Μηχανολογία της Γνώσης 113
5.10 Απόκτηση Γνώσης 114
5.11 Μεθοδολογία Ανάπτυξης ΕΣ 116

Κεφάλαιο Έκτο – ΕΠΙΛΟΓΟΣ
6.1 Επίλογος.. 119

ΒΙΒΛΙΟΓΡΑΦΙΑ... 120

<u>ΠΕΡΙΕΧΟΜΕΝΑ ΣΧΗΜΑΤΩΝ</u>

Σχήμα 3.1 Αναζήτηση κατά πλάτος …………………………….59
Σχήμα 3.2 Αναζήτηση κατά βάθος και οπισθοδρόμηση………61
Σχήμα 3.3 Το πρόβλημα των δύο δοχείων……………………..63
Σχήμα 4.1 Το περιβάλλον του Clips……………………………..67
Σχήμα 5.1 Δομή έμπειρου συστήματος…………………………108
Σχήμα 5.2 Κατηγορίες εμπλεκόμενων ανθρώπων
στην κατασκευή ΕΣ……………………………………113
Σχήμα 5.3 Στάδια μηχανολογίας της γνώσης…………………..114
Σχήμα 5.4 Κύκλος ζωής έμπειρων συστημάτων……………….117

ΠΕΡΙΕΧΟΜΕΝΑ ΠΙΝΑΚΩΝ

Πίνακας 1 Τελεστές μετάβασης…………………………………64
Πίνακας 2 Παραδείγματα ταυτοποίησης γεγονότων…………76
Πίνακας 3 Παραδείγματα μη ταυτοποίησης γεγονότων………76
Πίνακας 4 Παράδειγμα ταυτοποίησης μεταβλητών
Πολλαπλών τιμών……………………………………..77
Πίνακας 5 Βασικές αριθμητικές συναρτήσεις…………………..79
Πίνακας 6 Συναρτήσεις σύγκρισης αριθμών…………………...80
Πίνακας 7 Λογικές συναρτήσεις…………………………………81
Πίνακας 8 Συναρτήσεις ελέγχου τύπου…………………………83
Πίνακας 9 Συνδετικά συνθηκών κανόνων………………………91
Πίνακας 10 Τύποι τιμών για τις ιδιότητες των προτύπων………98

ΠΡΟΛΟΓΟΣ

Θέμα της Πτυχιακής Εργασίας είναι ο Ευφυής Προγραμματισμός. Γλώσσες Προγραμματισμού που τον υποστηρίζουν καθώς και η εξέλιξή τους στο πέρασμα του χρόνου. Χρόνος ανάθεσης της εργασίας αυτής ξεκινά από τον Σεπτέμβριο του 2006 έως τον Μάρτιο του 2008 από τους σπουδαστές Γιώργο Παπαγεωργίου και Μάριο Γιωργαλλέττο. Ως αρχικός επιβλέπων καθηγητής είχε ορισθεί η κα Σωτηροπούλου και η διεκπεραίωση έγινε από τον κο Μάνδαλο.

ΕΙΣΑΓΩΓΗ

Ευφυής Προγραμματισμός, γλώσσες που υποστηρίζουν τον προγραμματισμό, είναι το θέμα που αναπτύσσεται στα παρακάτω κεφάλαια. Lisp, Τεχνητή Νοημοσύνη, Clips και Έμπειρα Συστήματα μας δίνουν να καταλάβουμε και να κατανοήσουμε τον ρόλο και την ανάγκη που υπήρχε, από την δεκαετία του '50 έως τώρα, έναν «αγώνα» αναζήτησης επιστημών για την εύρεση μορφών αποκωδικοποίησης, αν θα μπορούσαμε να το πούμε έτσι, της καθημερινότητας μέσα από τις επιστήμες και τις τέχνες. Ο ρόλος των επιστημόνων στην εξέλιξη ενός συστήματος «μετάδοσης» της ανθρώπινης γνώσης σε μηχανές για την πιο γρήγορη εξαγωγή συμπερασμάτων και λύσεως αλλά και η αντίδραση, μέσα από κακή πληροφόρηση, του ανθρώπου που βλέποντας ένα «θολό τοπίο» φοβάται την εξέλιξη των μηχανών. Αυτό οδηγεί σε μια αποστασιοποίηση των χρηματοδοτών των ερευνών και σε μια συνεχή προσπάθεια απόδειξης του αντιθέτου από του επιστήμονες. Παρακάτω παρουσιάζονται προγράμματα ευφυούς προγραμματισμού, η εξέλιξή τους και εφαρμογές αυτών.

ΚΕΦΑΛΑΙΟ 1^{ο} LISP

1.1 ΕΙΣΑΓΩΓΗ ΣΤΗ LISP

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. Βασική δομή δεδομένων στη LISP είναι η λίστα, απ' όπου πήρε και το όνομα της (LISt Processing). Δηλαδή οι επεξεργασίες σ' ένα πρόγραμμα LISP είναι κατά βάση 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. Ενώ ο αριθμητικός υπολογισμός 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 αποδεικτές θεωρημάτων κ.ά.

Ιστορική εξέλιξη

- LISP 1.5 Η πρώτη έκδοση με ευρεία διανομή, αναπτύχθηκε από τον McCarthy και άλλους στο MIT. Ονομάστηκε έτσι επειδή περιέχει αρκετές βελτιώσεις σε σχέση με τον αρχικό ερμηνευτή της "LISP 1," αλλά χωρίς να απαιτεί βασική αναδιοργάνωση, όπως θα έκανε η προγραμματιζόμενη

LISP 2. (Η LISP 2 χρησιμοποιούσε σύνταξη βασισμένη σε M-εκφράσεις και δεν χρησιμοποιήθηκε ευρέως.)

- Stanford LISP 1.6 – Η διάδοχος της LISP 1.5, αναπτύχθηκε στο Stanford AI Lab, και εξαπλώθηκε πολύ στα συστήματα PDP-10 τα οποία έτρεχαν το λειτουργικό σύστημα TOPS-10. Ξεπεράστηκε από τις Maclisp και InterLisp.
- MACLISP– αναπτύχθηκε για το Project MAC του MIT (άσχετο με το Macintosh της Apple και με τον McCarthy), και είναι άμεσος απόγονος της LISP 1.5. Έτρεχε σε PDP-10 και Multics συστήματα.
- InterLisp– αναπτύχθηκε στο BBN για συστήματα PDP-10 με το Tenex λειτουργικό σύστημα, και αργότερα υιοθετήθηκε ως Lisp της "δυτικής ακτής" για μηχανές Xerox Lisp. Μια μικρή έκδοση με το όνομα "InterLISP 65" δημοσιεύθηκε με τη γραμμή υπολογιστών της Atari που βασιζόταν στο 6502. Για αρκετό καιρό η MacLisp και η InterLisp ήταν ανταγωνιστές.
- Franz Lisp – αρχικά μια απόπειρα του Berkeley, αργότερα αναπτύχθηκε από τη Franz Inc. Το όνομα αποτελεί χιουμοριστικό λογοπαίγνιο του 'Franz Liszt'.
- ZetaLisp – χρησιμοποιήθηκε στα Lisp machine, άμεσος απόγονος της MacLisp.
- EuLisp – απόπειρα να αναπτυχθεί μια νέα, αποδοτική και "καθαρή" Lisp.
- ISLisp – απόπειρα να αναπτυχθεί μια νέα, αποδοτική και "καθαρή" Lisp. Καθορίστηκε ως στάνταρ.
- IEEE Scheme – IEEE standard, 1178-1990 (R1995)
- ANSI Common Lisp – ως επί το πλείστον μια καθαρή έκδοση της ZetaLisp που εμπεριείχε το CLOS.

Η Lisp σήμερα

Έχοντας παρακμάσει στη δεκαετία 1990, η Lisp συναντά αναζωογόνηση του ενδιαφέροντος από το 2000. Το μεγαλύτερο κομμάτι της νέας δραστηριότητας συγκεντρώνεται γύρω από υλοποιήσεις της Common Lisp από το Ανοικτό λογισμικό, και συμπεριλαμβάνει την ανάπτυξη νέων φορητών βιβλιοθηκών και

εφαρμογών. Πολλοί νέοι προγραμματιστές Lisp έχουν πεισθεί από συγγραφείς όπως ο Paul Graham και ο Eric S. Raymond να ασχοληθούν με μια γλώσσα που άλλοι θεωρούν απαρχαιωμένη. Οι νέοι αυτοί προγραμματιστές Lisp συχνά περιγράφουν τη γλώσσα ως διαφωτιστική εμπειρία και ισχυρίζονται ότι είναι αρκετά πιο παραγωγικοί από ότι σε άλλες γλώσσες. Ο Graham αναπτύσσει μια νέα διάλεκτο Lisp που λέγεται Arc.

Κύριες σύγχρονοι διάλεκτοι

Οι δύο κύριες διάλεκτοι της Lisp που χρησιμοποιούνται σήμερα για γενικής χρήσης προγραμματισμό είναι η Common Lisp και η Scheme. Αυτές οι γλώσσες αντιπροσωπεύουν σημαντικά διαφορετικές επιλογές στο σχεδιασμό. Η Common Lisp, προερχόμενη κυρίως από τις MacLisp, Interlisp και Lisp Machine Lisp, είναι ένα εκτεταμένο υπερσύνολο παλιότερων διαλέκτων Lisp, με ένα ογκώδες πρότυπο γλώσσας που περιλαμβάνει πολλούς ενσωματωμένους τύπους δεδομένων και συντακτικές μορφές, καθώς και ένα σύστημα αντικειμένων. Η Scheme έχει ένα μινιμαλιστικότερο σχεδιασμό, με πολύ μικρότερο σύνολο λειτουργιών αλλά με κάποια επιπλέον στοιχεία υλοποίησης (όπως tail-call optimization και πλήρη υποστήριξη continuation) που δεν υπάρχουν στην Common Lisp. Επίσης, η Common Lisp έχει δανειστεί κάποια στοιχεία από τη Scheme, όπως lexical scoping και lexical closures. Επιπλέον, οι διάλεκτοι Lisp χρησιμοποιούνται ως scripting γλώσσες σε αρκετές εφαρμογές, η πιο γνωστή εκ των οποίων είναι η Emacs Lisp στον Emacs συντάκτη κειμένου και η Autolisp στο AutoCAD.

Lisp και Τεχνητή Νοημοσύνη

Από την δημιουργία της, η Lisp συνδεόταν στενά με την ερευνητική κοινότητα της τεχνητής νοημοσύνης, ειδικά στα συστήματα PDP-10. Η Lisp χρησιμοποιήθηκε ως η υλοποίηση της γλώσσας Micro Planner που έθεσε τα θεμέλια για το διάσημο σύστημα τεχνητής νοημοσύνης SHRDLU. Τη δεκαετία 1970, ενώ η έρευνα στην τεχνητή νοημοσύνη δημιουργούσε εμπορικές εταιρίες, η απόδοση

των υπαρχόντων συστημάτων Lisp έγινε σημαντικό ζήτημα. Η Lisp ήταν δύσκολο σύστημα στο να υλοποιηθεί με τις τεχνικές μεταγλώττισης και το υλικό της δεκαετίας του 1970. Οι ρουτίνες συλλογής απορριμμάτων που αναπτύχθηκαν από τον τότε μεταπτυχιακό φοιτητή του MIT, Daniel Edwards, έκαναν πρακτική την εκτέλεση της Lisp σε υπολογιστικά συστήματα γενικής χρήσης, αν και η απόδοση ήταν ακόμα προβληματική. Αυτό οδήγησε στην δημιουργία των μηχανών Lisp: ειδικό υλικό για την εκτέλεση περιβάλλοντος και προγραμμάτων Lisp. Σύντομα οι εξελίξεις τόσο στον τομέα του υλικού υπολογιστών όσο και στην τεχνολογία μεταγλώττισης έκαναν τις μηχανές Lisp ξεπερασμένες, πράγμα επιζήμιο για την αγορά της Lisp. Κατά τις δεκαετίες 1980 και 1990 έγινε μεγάλη προσπάθεια να ενοποιηθούν οι πολυάριθμες διάλεκτοι Lisp (κυρίως οι InterLisp, Maclisp, ZetaLisp και Franz Lisp) σε μία και μόνη γλώσσα. Η νέα αυτή γλώσσα, η Common Lisp, ήταν ουσιαστικά ένα συμβατό υποσύνολο των διαλέκτων που αντικατέστησε. Το 1994, η ANSI δημοσίευσε το πρότυπο της Common Lisp, "ANSI X3.226-1994 Information Technology Programming Language Common Lisp." Εκείνη την περίοδο η παγκόσμια αγορά για τη Lisp ήταν πολύ μικρότερη από ότι είναι σήμερα.

1.2 ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ – ΟΡΙΣΜΟΙ

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 O3P είναι η συνάρτηση. Επίσης, 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 εξαγόμενο 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 O3P γίνεται με τις συμβολικές εκφράσεις ή σ-εκφράσεις (s-expressions). Αυτές αποτελούν τα δομικά στοιχεία της γλώσσας, δηλ. τα στοιχεία από 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 αποτελέσματος της) ονομάζεται συναρτησιακός τύπος (functional form) ή απλώς τύπος (form). 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 λίστα '()' και αντιπροσωπεύει το ψεύδος μιας πρότασης, ενώ το T την αλήθεια. Μια λίστα είναι της μορφής (e1 e_2 · e n), όπου ei 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. Οι λίστες και οι συμβολοσειρές αποτελούν τις ακολουθίες Το μόνο που διαθέτει είναι τα ειδικά σύμβολα T (true) και NIL. Το NIL είναι ταυτόσημο με την κενή λίστα '()' και αντιπροσωπεύει το ψεύδος μιας πρότασης, ενώ το T την αλήθεια. `Μια λίστα είναι της μορφής (e1 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 . Οι λίστες και οι συμβολοσειρές αποτελούν τις ακολουθίες. Ένας συναρτησιακός τύπος (ή τύπος) είναι μια έκφραση που έχει τη μορφή:

(<function-name> <arg>…<argn>)

όπου <function-name> είναι το όνομα μιας συνάρτησης (είτε θεμελιώδους-ενσωματωμένης είτε ορισμένης από τον χρήστη) και κάθε <argi> είναι μια συμβολική έκφραση ή άλλος συναρτησιακός τύπος. Τα <arg_i> αποτελούν τα ορίσματα ή παράμετροι της συνάρτησης. Η LISP είναι κυρίως γλώσσα διερμήνευα μένη (παρ' ότι συνήθως υπάρχει ταυτόχρονα και μεταγλωττιστής). Ο κύκλος εκτέλεσης του διερμηνευτή της LISP είναι ο εξής:

1.3 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.

1.3.1 Κανόνες Εκτίμησης

Για την εκτίμηση (evaluation) των σ-εκφράσεων υπάρχουν συγκεκριμένοι κανόνες, που συνοψίζονται παρακάτω, όπου το σύμβολο '*' παριστάνει την προτροπή (prompt) του περιβάλλοντος O3P (άλλη συνήθης προτροπή είναι το '>') και το '—' σημαίνει «εκτιμάται σε» και υπονοεί το πάτημα του

• Κάθε αριθμός εκτιμάται στον εαυτό του. Π.χ.

* 5^5

•Τα ειδικά σύμβολα εκτιμώνται στον εαυτό τους. Π.χ.
*î^T

* nil -> NIL

- Κάθε σύμβολο εκτιμάται στην τιμή του, εκτός αν υπάρχει το quote (') μπροστά απ' αυτό, οπότε εκτιμάται, στον εαυτό του. Ουσιαστικά, 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)

  * (ςιιοίε πιαΓκ) —» ΜΑΚΚ ή

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 τ/μτ/.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)->(12 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

1.4 ΘΕΜΕΛΙΩΔΕΙΣ ΣΥΝΑΡΤΗΣΕΙΣ

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)

* (car ((a b) c d))-> (C D)

CXXXX (x = a,d)

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

(caard 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

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.
*(NTHCDR2’(12 3 4))->(3 4)

LENGTH

Σύνταξη: (length <list>)

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

Σύνταξη: (reverse <list>)

Επιστρέφει: Την ανάστροφη λιστα. Π.χ.

* (reverse ‘(1 2 3)) -> (3 2 1)

REVERSE

Σύνταξη: (reverse <list>)

Επιστρέφει: Την ανάστροφη λίστα. Π.χ.

* (length ‘(1 2 3))-> (3 2 1)

SUBSEQ

Σύνταξη: (subseq <index1> [<index2>])

Επιστρέφει: Τη λίστα από <index1>+1 μέχρι και <index2>. Αν το <index2> λείπει μέχρι το τέλος. Π.χ.

*(subseq ‘(1 2 3 4 5 6) 2 4) -> (3 4)

*(subseq ‘(1 2 3 4 5 6) 3) -> (4 5 6)

Συναρτήσεις σύνθεσης

CONS

Σύνταξη: (cons <s-expression> <list>)

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

*(cons ‘x’ (a b)) -> (A X 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)) ->(ABCD)

Συναρτήσεις Τροποποίησης

PUSH

Σύνταζη: (push <s-expression> <symbol>), όπου το <symbol> έχει σαν τιμή μια λίστα.
Επιστρέφει: Τη λίστα με επί πλέον πρώτο στοιχείο την <sexpression>
Πλευρικό αποτέλεσμα: Η τιμή του <symbol> γίνεται η νέα λίστα. Π.χ.
* X ->(ABC)
* (push 1 x) -> (1 A B C)* χ-> (1ABC)

•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->(IABC)
* (pop χ) ->(AB C)* χ->(ABC)

Συναρτήσεις Διαχείρισης Λιστών-Ζευγών (ή Λιστών Συσχέτισης)

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))

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.

*(setqx'(12 3))->(12 3)

*χ^(12 3)

(setq x '(1 2 3) y 5 z (+ 5 4)) ->9

*χ^(12 3)

*y -> 5

*Z->9

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)

Αριθμητικές Συναρτήσεις Βασικών πράξεων

• + , - , * , / , float

Π.χ.

* (/ 27 9) → 3

* (/ 4.16 1.3) → 3.2

* (/ 22 7) → 22/7 12

Συναρτήσεις 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

Συναρτήσεις Αναγνώρισης

zerop: T αν το όρισμα του έχει τιμή '0'

plusp: T αν το όρισμα του είναι θετικός αριθμός.

minusp: T αν το όρισμα του είναι αρνητικός αριθμός

evenp: T αν το όρισμα του είναι άρτιος (ακέραιος) αριθμός.

oddp: T αν το όρισμα του είναι περιττός (ακέραιος) αριθμός > : T αν τα ορίσματα του (τουλάχιστον 2) είναι κατά φθίνουσα σειρά < : T αν τα ορίσματα του (τουλάχιστον 2) είναι κατ' αύξουσα σειρά.

Συναρτήσεις Αναγνώρισης Τύπων

atom: T αν το όρισμα του είναι άτομο (δηλ. αριθμός ή σύμβολο)
numberp: T αν το όρισμα του είναι αριθμός
symbolp: T αν το όρισμα του είναι σύμβολο
listp: T αν το όρισμα του είναι λίστα
null: T αν το όρισμα του είναι μια κενή λίστα
endp: T αν το όρισμα 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).

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

* (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

<u>Ανισότητας</u>

< , <= , > , >= (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

<u>Συνάρτηση Συμμετοχής</u>

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)

1.5 ΣΥΝΑΡΤΗΣΕΙΣ
ΧΡΗΣΤΗ-ΣΥΝΑΡΤΗΣΗ LET

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> (<param1> … <paramn>)
<form1>
…
<formm>)

όπου <function-name> είναι το όνομα της συνάρτησης που θέλουμε να ορίσουμε, τα <parami> είναι τα *ορίσματα* ή *τυπικές παράμετροι* της συνάρτησης και τα <formi> 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))
```

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))
```

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)
```

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.

1.6 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>)
```

1.7 ΕΛΕΓΧΟΣ ΡΟΗΣ ΠΡΟΓΡΑΜΜΑΤΟΣ

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

Διατάξεις επιλογής

Διάταξη if

Η διάταξη if έχει την παρακάτω σύνταξη:

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

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

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)
```

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 καμία από τις προηγούμενες δεν έχει εκτελεστεί.

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.

Διατάξεις επανάληψης

(διάταξη dotimes)(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 αποτέλεσμα. 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)

Διάταξη dolist

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 (elm 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.

counter	Εκτέλεση push	new-list
0 1 2	1η 2η 3η	(2) (3 2) (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 μια λίστας μέχρι όμως να συναντήσει 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 σύνταξη της είναι η εξής:

```
(do ((<parm1> [<init-val1> [<update-form1>]])
```

```
… (<paramn> [<init-valn> [<update-formn>]]))
(<termin-test> [[<intermed-form>] <result-form>1)
<form1>
…
<formn>)
```

όπου ένα ή περισσότερα <intermed-form> εν 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. εν 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 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 ο let, έτσι και για το do, υπάρχει το do*, στο οποί 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, ου 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 restlist (cdr rest-list))
(push (+ 1 elem) new-list))))
```

1.8 ΑΝΑΔΡΟΜΗ ΚΑΙ ΠΑΡΑΜΕΤΡΟΙ ΕΙΔΙΚΟΥ ΣΚΟΠΟΥ

1.8.1 Αναδρομικές Συναρτήσεις

Μια συνάρτηση ονομάζεται αναδρομική (recursive) όταν ορίζεται μέσω του εαυτού της (αναδρομική κλήση). Ένα βασικό στοιχείο σε μια αναδρομική συνάρτηση είναι να υπάρχει μα συνθήκη τερματισμού, 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 σε αυτούς:

```
(defun filter-negs (num-list)
(cond (null num-list) nil)    Συνθήκη τερματισμού
(plusp (c num-list))
(cons (car num-list)
(filtr-negs (cdrnum-list))))
(t (filter-negs (cdr num-list)))))
```

Filter negs= αναδρομικές κλήσεις

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

* (filtr-negs ‘(1-1 2-5)) -> (1 5)

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. Επίσης, η χρήσης 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.

1.8.2 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, οι υπόλοιπες, οι κλειδιά και οι βοηθητικές. Κάθε κατηγορία παραμέτρων δηλώνεται μέσω του αντίστοιχου τύπου: &optional, &rest, &key, &aux. Εδώ θα αναφερθούμε στις δύο πρώτες.

Προεραιτικές Παράμετροι

Δηλώνονται χρησιμοποιώντας το «&optional" μετά από τις κανονικές παραμέτρους/ορίσματα μιας συνάρτησης. Κατά την κλήση μιας συνάρτησης, μπορεί να υπάρχουν ή να μην υπάρχουν αντίστοιχες πραγματικές παράμετροι (εξ' ου και το όνομα προαιρετικές). Μπορούμε να δηλώσουμε ή να μην δηλώσουμε εξ' ορισμού τιμές για τις προαιρετικές παραμέτρους, οι οποίες χρησιμοποιούνται στην περίπτωση που δεν υπάρχουν πραγματικές παράμετροι στην κλήση της συνάρτησης. Στην περίπτωση 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 γι' αυτό ενός ορθογωνίου τον E=b*h. Για να τα συνδυάσουμε σε μια συνάρτηση, κάνουμε χρήση μιας προαιρετικής παραμέτρου, της factor:

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

Η συνάρτηση αυτή όταν δεν γίνεται 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 (factor1))
(*factor b h))
```

που είναι μια 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 κλήσης της
συνάρτησης, όλες οι πραγματικές παράμετροι με τις κανονικές 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)

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

1.9 ΕΙΣΟΔΟΣ-ΕΞΟΔΟΣ

Η Lisp όπως όλες οι γλώσσες προγραμματισμού 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 παρκάτω 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 να πληροφορεί τον χρήστη ότι πρέπει κάτι να πληκτρολογήσει. Αυτό γίνεται με την 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 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) σην οθόνη χωρίς μετακίνηση σε άλλη γραμμή:

(format t “Hello!”) -> Hello! (αυτό είναι το αποτέλεσμα 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 αφήσουμε και κενές γραμμές τότε τοποθετούμε ένα ακέραιο αριθμό “n” πριν το “%” οπότε γίνεται αλλαγή γραμμής και αφήνονται (n-1) κενές γραμμές. Π.χ. η

(prong (format t “-%Hello-3%”
(format t “HELLO-%”))
Θα έχει σαν αποτέλεσμα:
HELLO
HELLO
NIL

Δηλαδή 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 “%” όπου 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 ότι η 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 εντυπωθεί η τιμή του ορίσματος που αντιστοιχεί στο “a” και μετά να αφεθούν 8 κενά πριν την επόμενη εκτύπωση κάποιου στοιχειού.

1.10 ΣΥΝΑΡΤΗΣΕΙΣ ΜΕ ΟΡΙΣΜΑΤΑ ΣΥΝΑΡΤΗΣΕΙΣ

Ένα κοινό χαρακτηριστικό των συναρτήσεων αυτών είναι ότι χρησιμοποιούν σαν όρισμα κάποια συνάρτηση-διαδικασία. Κάποιες από αυτές ονομάζονται και

συναρτήσεις αντιστοίχησης, διότι αναφέρονται ε αντιστοιχίες μεταξύ λιστών, που υλοποιούνται είτε ως κάποιος μετασχηματισμός είτε ως φιλτράρισμα μιας λίστας.

Συναρτήσεις Ελέγχου

Οι συναρτήσεις αυτές διαπιστώνουν αν κάποιο ή όλα τα στοιχεία μιας λίστας ικανοποιούν κάποιο κριτήριο. Αναφέρουμε δύο τέτοιες συναρτήσεις, τις “some”και “every”.

Some

Σύνταξη: (some #’<(function-name><list>)

Επιστρέφει: Τα αν υπάρχει κάποιο στοιχείο της λίστας που ικανοποιεί την συνάρτηση (δηλαδή επιστέφει Τα), αλλιώς NIL. Π.χ.

*(some #’oddp ‘(1 2 4)) -> T

*(some #’oddp ‘(2 2 4)) -> NIL

Every

Σύνταξη: (every #’<(function-name><list>)

Επιστρέφει: Τα αν όλα τα στοιχεία της λίστας που ικανοποιούν τη συνάρτηση (δηλαδή επιστρέφει T), αλλιώς NIL. Π.χ.

*(every #’oddp ‘(1 2 4)) -> NIL

*(every #’evenp ‘(2 2 4)) -> T

Συναρτήσεις φίλτρου

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 να έχει 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 μια συμμετρική της προηγούμενης. Έχει την ακόλουθη σύνταξη:

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

Το αποτέλεσμα είναι η λίστα χωρίς να 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)

Συναρτήσεις Αναζήτησης-Απαρίθμησης

Find-if

Συνάρτηση αναζήτησης. Σύνταξη:

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

Επιστρέφει το πρώτο στοιχεί 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

Συναρτήσεις Αντιστοίχησης

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 <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)

Στη 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.

Ανώνυμη Συνάρτηση

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 σε 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 <param1>…<paramn>)
<form1>
…
<formn>

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 χρησιμοποιούμε 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.

Συναρτήσεις Κλήσης Συναρτήσεων

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 #’ 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 tet-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 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 μεταβλητή και στη δημιουργία συνάρτησης με ορίσματα συναρτήσεις:

*(funcall x’(a b c)) -> (A B C)

(defun test-fun (argx funx)

(apply funx (list argx)))

1.11 ΛΙΣΤΕΣ ΙΔΙΟΤΗΤΩΝ-ΠΙΝΑΚΕΣ

1.11.1 Λίστα ιδιοτήτων

Η Lisp επιτρέπει σε ένα σύμβολο να έχει τιμές ιδιοτήτων. Δηλαδή, μπορούμε σ’ ένα σύμβολο να προσαρτήσουμε ιδιότητες και να αποδώσουμε σε αυτές τιμές. Το σύνολο των ιδιοτήτων και των τιμών τους που είναι προσαρτημένο σε ένα σύμβολο λέγεται λίστα ιδιοτήτων 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 είναι 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

1.11.2 Πίνακες

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 έναν πίνακα στην οθόνη).
Δημιουργεί έναν 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 έναν πίνακα, τότε δίνονται μηδενικά στοιχεία. Επίσης, πρέπει να σημειώσουμε ότι τα στοιχεία ενός πίνακα στη Lisp μπορεί να είναι οτιδήποτε, δηλαδή και ετερογενή. Δεν απαιτείται όπως σε άλλες γλώσσες να είναι κοινού τύπου.
Η αρχικοποίηση ενός πίνακα μπορεί να γίνει με διάφορους τρόπους:
α) Με την 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 stuents (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 stuents) ‘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 δυσδιάστατου 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 δυσδιάστατο πίνακα με όνομα marks που έχει 2χ3=6 στοιχεία.
Η αρχικοποίηση ενός δυσδιάστατου πίνακα μπορεί να γίνει με τους ίδιους ακριβώς τρόπους, όπως ακριβώς 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 στο στοιχείο (0 0))

*(setf (aref marks 0 1) 6) -> B (ανάθεση στο στοιχείο (0 1)) κλπ.

Η προσπέλαση των στοιχείων ενός δυσδιάστατου πίνακα γίνεται παρόμοια με αυτή του μονοδιάστατου.
(aref <array name> <element index>)
*(aref marks 1 2) -> (αναφερόμενοι την τελευταία αρχικοποίηση)
Δύο συναρτήσεις που συνδέονται με του πίνακες ου είναι οι array-dimension και array-dimensions, που επιστρέφουν την (τις) διάσταση (σεις) 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 όλα τα μεγέθη των διαστάσεων του πίνακα. Π.χ.
*(array-dimensions students) -> (4)
*(array-dimensions marks) -> (2 3)

1.12 ΔΟΜΕΣ ΣΤΗ LISP

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

```
(defstruct <structura name>
(<field1><value1>)
(<field2><value2>)
…
(<fieldn><valuen>)
```

Π.χ. η

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

Δημιουργεί τη δομή dtudent με τα πεδία ‘name’, ‘sex’ και ‘performance’ και αρχική τιμή γι αυτά nil.

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

```
*(setf maria (make-student)) ->
#(STUDENT:YEAR NIL:SEX MALE:PERFORMANCE NIL)
```

Δημιουργεί ένα στιγμιότυπο με όνομα MARIA με τιμές πεδίων τις αρχικές τιμές, ενώ η

```
*(setf giannis (make-student: year ‘b’: sex ‘male’: performance ‘good’)) ->
# S(STUDENT: YEAP 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 αναγνώρισης για την προσπέλαση των τιμών των πεδίων. Αυτέ έχουν σαν όνομα το <structure

name>-<field name>, δηλαδή στην περίπτωσή μας θα είναι οι student-year, student-sex και student-performance. Τώρα:
*(student-sex maria) ->NIL
*(student-year giannis) -> B

Επιπλέον, γενικεύεται η χρήση της setf να λειτουργήσει και για τα πεδία των στιγμιότυπων. 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 τιμής 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 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 name structure of type STUDENT.
It is included in the structures:
UNIV-STUDENT
Its slot names and values are:
YEAR- ASEX- FEMALE
PERFORMANCE- VERY-GOO

2ο ΚΕΦΑΛΑΙΟ

ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

2.1 ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

Ο όρος **Τεχνητή Νοημοσύνη** (ΤΝ, εκ του *Artificial Intelligence*) αναφέρεται στον κλάδο της επιστήμης υπολογιστών ο οποίος ασχολείται με τη σχεδίαση και την υλοποίηση υπολογιστικών συστημάτων που μιμούνται στοιχεία της ανθρώπινης συμπεριφοράς τα οποία υπονοούν ευφυΐα (έστω και στοιχειώδη): μάθηση, προσαρμοστικότητα, εξαγωγή συμπερασμάτων, κατανόηση από τα συμφραζόμενα, επίλυση προβλημάτων κλπ. Ο Τζον Μακάρθι όρισε τον τομέα αυτόν ως "επιστήμη και μεθοδολογία της δημιουργίας νοούντων μηχανών". Η ΤΝ αποτελεί σημείο τομής μεταξύ πολλών επιστημών όπως της επιστήμης υπολογιστών, της ψυχολογίας, της φιλοσοφίας, της νευρολογίας, της γλωσσολογίας και της μηχανικής, με στόχο τη σύνθεση ευφυούς συμπεριφοράς, μάθησης και προσαρμογής στο περιβάλλον, ενώ συνήθως εφαρμόζεται σε μηχανές ή υπολογιστές ειδικής κατασκευής. Διαιρείται στη **συμβολική τεχνητή νοημοσύνη**, η οποία επιχειρεί να εξομοιώσει την ανθρώπινη νοημοσύνη αλγοριθμικά χρησιμοποιώντας σύμβολα και λογικούς κανόνες, και στην **υπολογιστική νοημοσύνη**, η οποία προσπαθεί να αναπαράγει την ανθρώπινη νοημοσύνη χρησιμοποιώντας μαθηματικές μεθόδους οι οποίες προσομοιώνουν πραγματικές βιολογικές διαδικασίες (όπως η εξέλιξη των ειδών και η λειτουργία του εγκεφάλου). Επομένως η διάκριση αυτή αφορά τον χαρακτήρα των χρησιμοποιούμενων εργαλείων και όχι τον επιθυμητό επιστημονικό στόχο. Με βάση το τελευταίο κριτήριο η ΤΝ κατηγοριοποιείται σε άλλου τύπου ευρείς τομείς, όπως επίλυση προβλημάτων, μηχανική μάθηση, ανακάλυψη γνώσης, συστήματα γνώσης κλπ. Επίσης υπάρχει επικάλυψη με συναφή επιστημονικά πεδία όπως η μηχανική όραση, η επεξεργασία φυσικής γλώσσας, η ρομποτική (π.χ. γνωστική ρομποτική) κλπ. Αν και η λογοτεχνία και ο κινηματογράφος επιστημονικής φαντασίας από τη δεκαετία του 1920 μέχρι σήμερα έχουν δώσει στο ευρύ κοινό μία άλλη αντίληψη για την ΤΝ, δηλαδή την προσπάθεια κατασκευής μηχανικών

ανδροειδών ή αυτοσυνείδητων προγραμμάτων υπολογιστή (*Ισχυρή ΤΝ*), επηρεάζοντας μάλιστα ακόμα και τους πρώτους ερευνητές του τομέα, η πραγματικότητα είναι διαφορετική: η τεχνητή νοημοσύνη, ως επιστημονικό πεδίο, προσπαθεί να κατασκευάσει λογισμικό ή ολοκληρωμένα υπολογιστικά συστήματα τα οποία να επιλύουν με αποδεκτά αποτελέσματα ρεαλιστικά προβλήματα οποιουδήποτε τύπου (*Ασθενής ΤΝ*). Ωστόσο πολλοί ερευνητές πιστεύουν ότι η εξομοίωση ή η προσομοίωση της πραγματικής ευφυΐας, η Ισχυρή ΤΝ, πρέπει να είναι ο τελικός στόχος. Αν και από τη δεκαετία του 1940 είχαν προταθεί και υλοποιηθεί τα πρώτα τεχνητά νευρωνικά δίκτυα, με πολύ περιορισμένες δυνατότητες επίλυσης αριθμητικών προβλημάτων, η τεχνητή νοημοσύνη θεμελιώθηκε στη συνάντηση ορισμένων επιφανών Αμερικανών επιστημόνων του τομέα το 1956 (Μακάρθι, Μίνσκυ, Σάνον κλπ). Ήδη όμως ο μαθηματικός Άλαν Τούρινγκ, πατέρας της θεωρίας υπολογισμού και προπάτορας της τεχνητής νοημοσύνης, είχε προτείνει τη δοκιμή Τούρινγκ· μία απλή δοκιμασία που θα μπορούσε να εξακριβώσει αν μία μηχανή διαθέτει ευφυΐα. Επόμενοι σημαντικοί σταθμοί ήταν η ανάπτυξη της γλώσσας προγραμματισμού LISP το 1958 από τον Μακάρθι, δηλαδή της πρώτης γλώσσας συναρτησιακού προγραμματισμού η οποία έπαιξε πολύ σημαντικό ρόλο στη δημιουργία εφαρμογών ΤΝ κατά τις επόμενες δεκαετίες, η εμφάνιση των γενετικών αλγορίθμων την ίδια χρονιά από τον Φρίντμπεργκ και η παρουσίαση του βελτιωμένου νευρωνικού δικτύου *perceptron* το '62 από τον Ρόσενμπλατ. Κατά τα τέλη της δεκαετίας του '60 όμως άρχισε ο *χειμώνας της ΤΝ*, μία εποχή κριτικής, απογοήτευσης και υποχρηματοδότησης των ερευνητικών προγραμμάτων καθώς όλα τα μέχρι τότε εργαλεία του χώρου ήταν κατάλληλα μόνο για την επίλυση εξαιρετικά απλών προβλημάτων. Στα μέσα του '70 ωστόσο προέκυψε μία αναθέρμανση του ενδιαφέροντος για τον τομέα λόγω των εμπορικών εφαρμογών που απέκτησαν τα **έμπειρα συστήματα**, μηχανές ΤΝ με αποθηκευμένη γνώση για έναν εξειδικευμένο τομέα και δυνατότητα ταχείας εξαγωγής λογικών συμπερασμάτων, τα οποία συμπεριφέρονται όπως ένας άνθρωπος ειδικός στον αντίστοιχο τομέα. Παράλληλα έκανε την εμφάνιση της η γλώσσα λογικού προγραμματισμού Prolog η οποία έδωσε νέα ώθηση στη

συμβολική ΤΝ, ενώ στις αρχές της δεκαετίας του '80 άρχισαν να υλοποιούνται πολύ πιο ισχυρά και με περισσότερες εφαρμογές νευρωνικά δίκτυα, όπως τα πολυεπίπεδα perceptron και τα δίκτυα Hopfield. Κατά τη δεκαετία του '90, με την αυξανόμενη σημασία του Internet, ανάπτυξη γνώρισαν οι **ευφυείς πράκτορες**, αυτόνομο λογισμικό ΤΝ τοποθετημένο σε κάποιο περιβάλλον με το οποίο αλληλεπιδρά, οι οποίοι βρήκαν μεγάλο πεδίο εφαρμογών λόγω της εξάπλωσης του Διαδικτύου. Οι πράκτορες στοχεύουν συνήθως στην παροχή βοήθειας στους χρήστες τους, στη συλλογή ή ανάλυση γιγάντιων συνόλων δεδομένων ή στην αυτοματοποίηση επαναλαμβανόμενων εργασιών (π.χ. βλέπε διαδικτυακό ρομπότ), ενώ στους τρόπους κατασκευής και λειτουργίας τους συνοψίζουν όλες τις γνωστές μεθοδολογίες ΤΝ που αναπτύχθηκαν με το πέρασμα του χρόνου. Έτσι σήμερα, όχι σπάνια, η ΤΝ ορίζεται ως *η επιστήμη που μελετά τη σχεδίαση και υλοποίηση ευφυών πρακτόρων*. Επίσης τη δεκαετία του '90 η ΤΝ, ιδιαίτερα η μηχανική μάθηση και η ανακάλυψη γνώσης, άρχισε να επηρεάζεται πολύ από τη θεωρία πιθανοτήτων και τη στατιστική. Τα Μπεϋζιανά δίκτυα είναι η εστίαση αυτής της νέας μετακίνησης, που παρέχει τις συνδέσεις με τα πιο σχολαστικά θέματα της στατιστικής και της εφαρμοσμένης μηχανικής, όπως τα πρότυπα Markov και τα φίλτρα Kalman. Αυτή η νέα πιθανοκρατική προσέγγιση, λόγω του αυστηρού μαθηματικού χαρακτήρα της, συνήθως συγκαταλέγεται στην υπολογιστική νοημοσύνη παρ' όλο που δε βασίζεται σε κάποιο βιολογικό υπόδειγμα.

2.2 Ιστορική εξέλιξη της ΤΝ:

Χρόνος	Εξέλιξη
1950	Ο Άλαν Τούρινγκ περιγράφει τη δοκιμή Τούρινγκ, που επιδιώκει να εξετάσει την ικανότητα μιας μηχανής να συμμετάσχει απρόσκοπτα σε μια ανθρώπινη συνομιλία.

1951	Τα πρώτα προγράμματα ΤΝ γράφονται για τον υπολογιστή Ferranti Mark I στο Πανεπιστήμιο του Μάντσεστερ: ένα πρόγραμμα που παίζει ντάμα από τον Κρίστοφερ Στράκλι και ένα που παίζει σκάκι από τον Ντίτριχ Πρίνζ.
1956	Ο Τζον Μακάρθι πλάθει τον όρο «Τεχνητή Νοημοσύνη» ως κύριο θέμα της διάσκεψης του Ντάρτμουθ.
1958	Ο Τζον Μακάρθι εφευρίσκει την γλώσσα προγραμματισμού Lisp.
1965	Ο Έντουαρτ Φάιγκενμπαουμ ξεκινά το Dendral, μια δεκαετή προσπάθεια ανάπτυξης λογισμικού που θα συμπεράνει τη μοριακή δομή οργανικών ενώσεων χρησιμοποιώντας ενδείξεις επιστημονικών οργάνων. Ήταν το πρώτο Έμπειρο Σύστημα (Expert System).
1966	Ιδρύεται το Εργαστήριο Μηχανικής Νοημοσύνης στο Εδιμβούργο – το πρώτο από μια σημαντική σειρά που οργανώνεται από τον Ντόναλντ Μίτσι και άλλους.
1970	Αναπτύσσεται το Planner και χρησιμοποιείται στο SHRDLU, μια εντυπωσιακή επίδειξη αλληλεπίδρασης μεταξύ ανθρώπου και υπολογιστή.
1971	Ξεκινά εργασία πάνω στο σύστημα απόδειξης θεωρημάτων Boyer-Moore στο Εδιμβούργο.

1972	Η γλώσσα προγραμματισμού Prolog αναπτύσσεται από τον Αλάν Κολμεροέρ.
1973	Ρομπότ συναρμολόγησης «Φρέντι» στο Εδιμβούργο: ένα ευπροσάρμοστο σύστημα συναρμολόγησης που ελέγχεται από υπολογιστές.
1974	Ο Τέντ Σόρτλιφ γράφει τη διατριβή του για το πρόγραμμα MYCIN (Stanford), το οποίο κατέδειξε μια πολύ πρακτική προσέγγιση στην ιατρική διάγνωση που βασίζεται σε κανόνες, και δουλεύει ακόμα και στην παρουσία της αβεβαιότητας. Ενώ δανείστηκε από το DENDRAL, οι δικές του συνεισφορές επηρέασαν έντονα το μέλλον των έμπειρων συστημάτων, ένα μέλλον με κατ' εξοχήν εμπορικές εφαρμογές.
1991	Η εφαρμογή σχεδίασης ενεργειών DART χρησιμοποιείται αποτελεσματικά στον Α' Πόλεμο του Κόλπου και ανταμείβει 30 χρόνια έρευνας στην ΤΝ του Αμερικανικού Στρατού.
1994	Ντίκμαννς και Ντάιμλερ-Μπένζ οδηγούν περισσότερο από 1000 km σε μια εθνική οδό του Παρισιού υπό συνθήκες βαρείας κυκλοφορίας και σε ταχύτητες ως και 130 km/ώρα. Επιδεικνύουν αυτόνομη οδήγηση σε ελεύθερες παρόδους, οδήγηση σε συνοδεία, αλλαγή παρόδων και αυτόματη προσπέραση άλλων οχημάτων.
1997	Ο υπολογιστής Deep Blue της IBM νικά των παγκόσμιο πρωταθλητή σκακιού Γκάρι Κασπάροφ.

1998	Κυκλοφορεί ο Φέρμπι της Tiger Electronics και γίνεται η πρώτη επιτυχημένη εμφάνιση ΤΝ σε οικιακό περιβάλλον.
1999	Η Sony λανσάρει το AIBO, που είναι ένα από τα πρώτα κατοικίδια ΤΝ που είναι επίσης αυτόνομα.
2004	Η DARPA ξεκινά το DARPA Grand Challenge («Μεγάλη Πρόκληση DARPA»), που προκαλεί τους συμμετέχοντες να δημιουργήσουν αυτόνομα οχήματα για ένα χρηματικό βραβείο.

2.3 Συμβατική Τεχνητή Νοημοσύνη

Η συμβατική τεχνητή νοημοσύνη εμπλέκει μεθόδους μηχανικής μάθησης (machine learning), που χαρακτηρίζονται από αυστηρούς μαθηματικούς αλγόριθμους και στατιστικές μεθόδους ανάλυσης. Διακρίνεται σε:

- Έμπειρα ή Εξειδικευμένα συστήματα (Expert systems), που εφαρμόζουν προγραμματισμένες ρουτίνες λογικής, σχεδιασμένες αποκλειστικά για μία συγκεκριμένη εργασία, προκειμένου να εξαχθεί κάποιο συμπέρασμα. Για το σκοπό αυτό, διεξάγεται επεξεργασία μεγάλων ποσοτήτων γνωστών πληροφοριών.
- Λογική κατά περίπτωση (Case based reasoning). Η επίλυση ενός προβλήματος βασίζεται στην προηγούμενη επίλυση παρόμοιων προβλημάτων.
- Μπαϋεσιανά δίκτυα (Bayesian networks). Βασίζονται στη στατιστική ανάλυση για τη λήψη αποφάσεων.

- Συμπεριφορική τεχνητή νοημοσύνη (Behavior based AI). Μέθοδος τεμαχισμού της λογικής διαδικασίας και στη συνέχεια χειροκίνητης οικοδόμησης του αποτελέσματος.

2.4 Υπολογιστική Τεχνητή Νοημοσύνη

Η υπολογιστική τεχνητή νοημοσύνη βασίζεται στη μάθηση μέσω επαναληπτικών διαδικασιών (ρύθμιση παραμέτρων). Η μάθηση βασίζεται σε εμπειρικά δεδομένα και σε μη-συμβολικές μεθόδους. Διακρίνεται σε:

- Τεχνητά νευρωνικά δίκτυα (Artificial neural networks) με πολύ ισχυρές δυνατότητες αναγνώρισης προτύπων (pattern recognition). Προσομοιάζουν τη λειτουργία των νευρώνων των έμβιων όντων.
- Συστήματα Ασαφούς λογικής (Fuzzy logic systems). Αποτελούν τεχνικές λήψης απόφασης κάτω από αβεβαιότητα. Βασίζονται στην ύπαρξη μη-αυστηρά διαχωρισμένων καταστάσεων, των οποίων η βαρύτητα λαμβάνεται υπόψη κατά περίπτωση. Υπάρχουν ήδη πολλές εφαρμογές των τεχνικών αυτών.
- Εξελικτική υπολογιστική (Evolutionary computation). Η ανάπτυξή τους προέκυψε από τη μελέτη των έμβιων οργανισμών και αφορούν σε έννοιες όπως του πληθυσμού, της μετάλλαξης και της φυσικής επιλογής (επιβίωση του πιο προσαρμοσμένου) για την ακριβέστερη επίλυση ενός προβλήματος. Οι μέθοδοι αυτοί μπορούν να διακριθούν περαιτέρω σε εξελικτικούς αλγόριθμους (evolutionary algorithms) και σε νοημοσύνης σμήνους (swarm intelligence), όπως πχ οι αλγόριθμοι που προσομοιάζουν τη συμπεριφορά μίας κοινωνίας μυρμηγκιών.

3^Ο ΚΕΦΑΛΑΙΟ

ΠΡΟΒΛΗΜΑΤΑ ΤΝ ΚΑΙ LISP

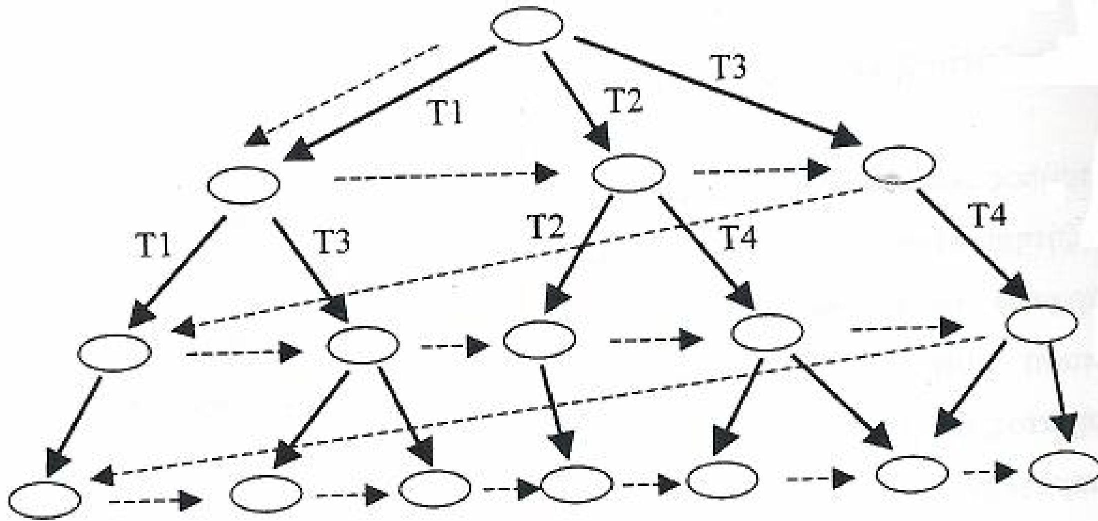
3.1 Αναζήτηση και Στρατηγικές Αναζήτησης

Ένας τρόπος επίλυσης προβλημάτων με μεθόδους Τεχνητής Νοημοσύνης (ΤΝ) είναι η 'αναζήτηση λύσης' (search). Σύμφωνα μ' αυτήν, ένα πρόβλημα παριστάνεται ως μια (αρχική) 'κατάσταση' και η επίλυσή του ως μια σειρά μεταβάσεων από την κατάσταση αυτή σε μια (τελική) κατάσταση, που αποτελεί τη λύση του προβλήματος, δια μέσου άλλων (ενδιάμεσων) καταστάσεων.

Η παραγωγή των ενδιάμεσων καταστάσεων γίνεται με τη χρήση των 'τελεστών μετάβασης'. Ένας τελεστής μετάβασης προσδιορίζει την επόμενη κατάσταση με βάση την προηγούμενη. Ένα πρόβλημα μπορεί να έχει ένα ή περισσότερους τελεστές μετάβασης, που ουσιαστικά αναπαριστούν τις ενέργειες που μπορούν να γίνουν για να φτάσουμε στη λύση του προβλήματος. Για να μπορεί να εφαρμοστεί ένας τελεστής πρέπει να ικανοποιούνται ορισμένες προϋποθέσεις, που αποτελούν μέρος του ορισμού του τελεστή. Δεδομένου ότι οι τελεστές μετάβασης είναι συνήθως περισσότεροι από ένας, τίθεται το ζήτημα του ποιόν θα εφαρμόσουμε κάθε φορά. Κατ' αρχή αποκλείονται αυτοί των οποίων οι προϋποθέσεις δεν ικανοποιούνται. Για τους υπόλοιπους καθορίζουμε μια σειρά με βάση κάποια 'στρατηγική αναζήτησης'. Υπάρχουν δύο κατηγορίες στρατηγικών αναζήτησης, οι 'τυφλές' και οι 'ευριστικές'.

Οι τυφλές στρατηγικές δεν λαμβάνουν υπ' όψιν το συγκεκριμένο πρόβλημα. Δύο τέτοιες στρατηγικές είναι οι **'αναζήτηση κατά πλάτος'** και '**αναζήτηση κατά βάθος' (με οπισθοδρόμηση)**. Η αναζήτηση κατά πλάτος (breadth-first search) εφαρμόζει κατ' αρχήν όλους τους εφαρμόσιμους τελεστές (με μια συγκεκριμένη σειρά) στην αρχική κατάσταση και παράγει τόσες νέες καταστάσεις όσοι και οι εφαρμόσιμοι τελεστές. Οι καταστάσεις αυτές, επειδή η διαδικασία της αναζήτησης μπορεί να παρασταθεί σαν ένα δένδρο, που λέγεται 'δένδρο αναζήτησης', λέγονται 'παιδιά' της αρχικής κατάστασης (που ονομάζεται 'ρίζα' του δένδρου). Στη συνέχεια, η αναζήτηση κατά πλάτος εφαρμόζει όλους τους

εφαρμόσιμους τελεστές (με την ίδια σειρά) στο πρώτο από τα ‘παιδιά’ που έχουν παραχθεί (και παράγει τα αντίστοιχα ‘παιδιά’ του), μετά στο δεύτερο κ.ο.κ. Όταν τελειώσουν τα ‘παιδιά’ της αρχικής κατάστασης, συνεχίζει με το πρώτο ‘παιδί’ του πρώτου ‘παιδιού’ της αρχικής κατάστασης κ.ο.κ. Όταν βρει τη λύση (τελική κατάσταση ή κατάσταση στόχου) σταματά.



Σχήμα 3.1 Αναζήτηση κατά πλάτος

Στο Σχήμα 3.1 απεικονίζεται ο τρόπος αναζήτησης κατά πλάτος, δηλ. η σειρά δημιουργίας των καταστάσεων (ως κόμβων ενός δέντρου). Τα Τ1, Τ2, Τ3 και Τ4 παριστάνουν τους τελεστές μετάβασης.

Ο αλγόριθμος της κατά πλάτος αναζήτησης σε ψευδοκώδικα παρουσιάζεται παρακάτω:

1. Δημιούργησε μια λίστα open (που αρχικά περιέχει τη ρίζα)
και μια κενή λίστα closed.
2. Ενόσω open ≠ [ ], έλεγχε αν το πρώτο στοιχείο, έστω Χ, είναι
ο στόχος
2.1 Αν είναι, τότε σταμάτα (επιτυχία)
2.2 Αν δεν είναι, τότε
2.2.1 Παράγαγε τα παιδιά του Χ και βάλε το Χ στην closed

2.2.2 Διάγραψε όσα παιδιά του X υπάρχουν στην closed

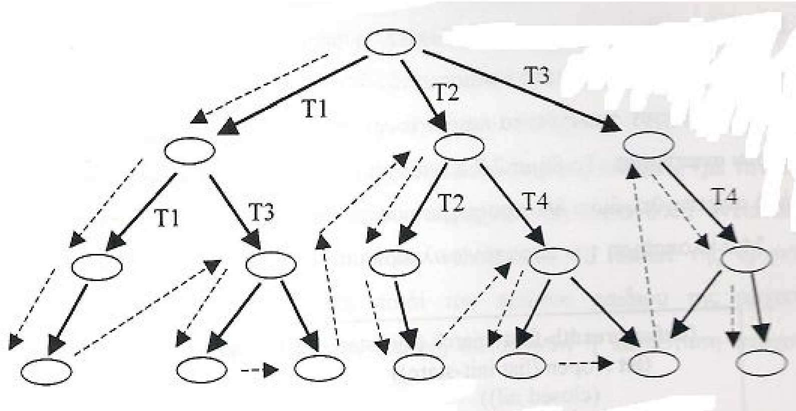
2.2.3 Εισάγαγε τα υπόλοιπα παιδιά του X στο τέλος της open

3. Σταμάτα (αποτυχία)

Βασικά, χρησιμοποιούνται δυο λίστες, η open και η closed, όπου αποθηκεύονται στην δε open οι ανοικτές καταστάσεις, δηλαδή αυτές που δεν έχουν ακόμη αναπτυχθεί (δηλαδή δεν έχουν παραχθεί τα ‘παιδιά’ τους), στη δε closed, οι κλειστές, δηλαδή αυτές που έχουν αναπτυχθεί. Το βήμα 2.2.2 υπάρχει για να διαγραφούν όσες καταστάσεις έχουν ήδη αναπτυχθεί, διότι δεν προσφέρουν κάτι καινούργιο.

Μια υλοποίηση του παραπάνω αλγορίθμου σε LISP είναι η παρακάτω.

```
(defun breadth-first-search (init-state final-states)
(let ((open (list init-state))
(closed nil))
(breadth-solve open closed final-states))
(defun breadth-solve (open closed final-states)
(if (null open) nil
(let ((cur-state (pop open))
(if (member cur-state final-states)
(show-solution-path cur-state closed)
(let* ((childs (make-childs cur-state))
(closed (push cur-state closed))
(childs (remove-closed-childs childs closed))
(open (append open childs)))
(breadth-solve open closed final-states))))))
```



Σχήμα 3.2 Αναζήτηση κατά βάθος και οπισθοδρόμηση

1. Δημιούργησε μια λίστα open (που αρχικά περιέχει τη ρίζα)
και μια κενή λίστα closed.
2. Ενόσω open ≠ [ ], έλεγχε αν το πρώτο στοιχείο, έστω X, είναι
ο στόχος
2.1 Αν είναι, τότε σταμάτα (επιτυχία)
2.2 Αν δεν είναι, τότε
2.2.1 <u>Παράγαγε</u> τα παιδιά του X και βάλε το X στην closed
2.2.2 Διάγραψε όσα παιδιά του X υπάρχουν στην closed
2.2.3 Εισάγαγε τα υπόλοιπα παιδιά του X <u>στην αρχή</u> της open
Η αντίστοιχη υλοποίηση σε LISP διαφέρει μόνο σε μια γραμμή:
(open (append childs open))
αντί (open (append open childs)), που καθορίζει το πού τοποθετούνται τα παραγόμενα παιδιά στη λίστα open.

Είτε με τη μία είτε με την άλλη στρατηγική ουσιαστικά αναζητούμε τη λύση δημιουργώντας όλες τις δυνατές καταστάσεις που μπορούν να δημιουργηθούν. Το ζητούμενο όμως στην Τεχνητή Νοημοσύνη είναι πώς θα μπορέσουμε να το αποφύγουμε αυτό κάνοντας την αναζήτηση 'έξυπνη'. Αυτό επιτυγχάνεται με τη χρήση 'ευριστικών' (heuristics), δηλ. έξυπνων τρικ ή κανόνων που συντομεύουν την αναζήτηση. Τα ευριστικά αυτά σχετίζονται με το συγκεκριμένο πρόβλημα κάθε φορά. Έτσι, δεν αναπτύσσουμε όλα τα παιδιά μιας κατάστασης, αλλά μόνο

τα ‘καλύτερα’, δηλ. αυτά που με βάση το ευριστικό μας «υπόσχονται» πιο σύντομη αναζήτηση. Μ’ αυτόν τον τρόπο δημιουργούνται ευριστικές στρατηγικές, που είναι παραλλαγές των παραπάνω τυφλών στρατηγικών. Π.χ. η στρατηγική που ονομάζεται ‘αναζήτηση δέσμης’ (ή ‘ακτινωτή αναζήτηση’) (beam search) λειτουργεί όπως η αναζήτηση κατά πλάτος, μόνο που κάθε φορά επιλέγει τα καλύτερα m από τα παιδιά κάθε κατάστασης για να αναπτύξει.

1. Δημιούργησε μια λίστα open (που αρχικά περιέχει τη ρίζα)
και μια κενή λίστα closed.
2. Ενόσω open ≠ [ ], έλεγχε αν το πρώτο στοιχείο, έστω X, είναι
ο στόχος
2.1 Αν είναι, τότε σταμάτα (επιτυχία)
2.2 Αν δεν είναι, τότε
2.2.1 Παράγαγε τα παιδιά του X και βάλε το X στην closed
2.2.2 Διάγραψε όσα παιδιά του X υπάρχουν στην closed
2.2.3 Διάταξε τα παιδιά του X με βάση το ευριστικό
2.2.4 Εισάγαγε τα παιδιά στην αρχή της open
3. Σταμάτα (αποτυχία)

Επίσης, η ‘αναρρίχηση λόφων’ (hill climbing) είναι μια παραλλαγή της αναζήτησης κατά βάθος, όπου δεν επιλέγεται κάθε φορά το πρώτο παιδί για ανάπτυξη, αλλά το καλύτερο, με βάση το ευριστικό (υπάρχουν διάφορες παραλλαγές του αλγορίθμου). Αυτές οι στρατηγικές βρίσκουν εν γένει τη λύση σε λιγότερα βήματα, αλλά χρειάζεται προσοχή στον ορισμό του ευριστικού και στην επιλογή του m (πόσο μικρό ή μεγάλο πρέπει να είναι).

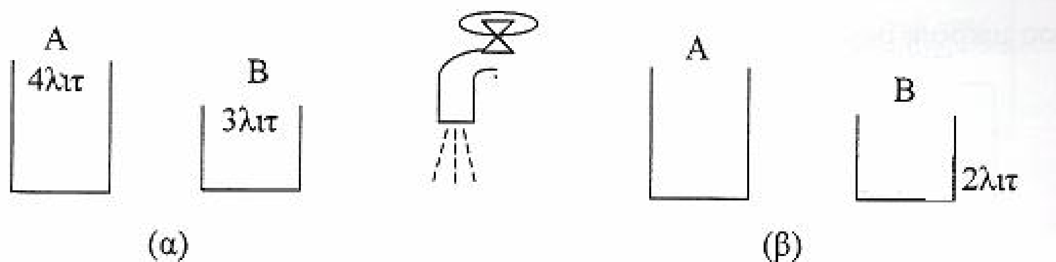
```
(defun hill-search (init-state final-states)
(let ((open (list init-state))
(closed nil))
(beam-solve open closed final-states))
(defun hill-solve (open closed final-states)
(if (null open) nil
(let ((cur-state (pop open))
```

```
(if (member cur-state final-states)
(show-solution-path cur-state closed)
(let* ((childs (make-childs cur-state))
(closed (push cur-state closed))
(childs (remove-closed-childs childs closed))
(childs (sort-childs childs))
(open (append childs open)))
```

3.2 Το Πρόβλημα των Δύο Δοχείων

Ας δούμε για παράδειγμα το γνωστό πρόβλημα των δύο δοχείων: «Υπάρχουν δύο δοχεία χωρητικότητας 4 και 3 λίτρων αντίστοιχα και μια βρύση. Κατ' αρχήν, τα δοχεία είναι άδεια. Θέλουμε να απομονώσουμε στο δεύτερο δοχείο ποσότητα 2 λίτρων. Οι δυνατές ενέργειες είναι: γέμισμα των δοχείων από τη βρύση, άδειασμα των δοχείων στο έδαφος, άδειασμα του ενός δοχείου στο άλλο, μερικώς ή ολικώς». Στο Σχήμα 3.3 φαίνεται η αρχική (α) και μια τελική κατάσταση (β) του προβλήματος.

Για να μπορέσουμε να λύσουμε το πρόβλημα με αναζήτηση θα πρέπει πρώτα να βρούμε ένα τρόπο αναπαράστασης μιας κατάστασης, ώστε να μπορέσουμε να αναπαραστήσουμε την αρχική και την τελική (ή τις τελικές) καταστάσεις. Μετά πρέπει να προσδιορίσουμε τους τελεστές μετάβασης, το ευριστικό και μετά να εφαρμόσουμε μια στρατηγική αναζήτησης.



Σχήμα 3.3 Το πρόβλημα των δύο δοχείων

όπου x η ποσότητα νερού στο δοχείο των 4 λίτρων (με δυνατές τιμές 0, 1, 2, 3, 4) και y στο δοχείο των 3 λίτρων (με δυνατές τιμές 0, 1, 2, 3). Οπότε η αρχική και οι τελικές καταστάσεις παριστάνονται:

Αρχική κατάσταση: (0 0)

Τελικές καταστάσεις : (2 2), (0 2), (1 2), (3 2), (4 2)

Παρατηρείστε ότι το κοινό στοιχείο των τελικών καταστάσεων είναι ότι το δεύτερο στοιχείο κάθε λίστας, που παριστάνει την ποσότητα νερού στο δοχείο B, είναι ‘2’. Σκοπός μα είναι να φτάσουμε σε μια από αυτές, ξεκινώντας από την αρχική.

Οι τελεστές μετάβασης έχουν ως εξής:

ΤΕΛΕΣΤΗΣ:ΠΕΡΙΓΡΑΦΗ	**ΠΡΟΫΠΟΘΕΣΕΙΣ**	**ΑΠΟΤΕΛΕΣΜΑ**
T1: Γέμισε το A	x<4	(4 y)
T2: Γέμισε το B	y<3	(x 3)
T3:Άδειασε το A	x>0	(0 y)
T4: Άδειασε το B	y>0	(x 0)
T5: Άδειασε το A στο B	x>0, y<3	Αν x≥3-y τότε ((x-(3-y)) 3), αλλιώς (0 (y+x))
T6: Άδειασε το B στο A	x<4, y>0	Αν y≥4-x τότε (4 (y-(4-x))), αλλιώς ((y+x) 0)

Στη συνέχεια δίνουμε τρόπους υλοποίησης των τελεστών μετάβασης. Π.χ. ο τελεστής T1 υλοποιείται ως εξής:

```
(defun t1 (state)
(let((x (car state))
(y(second state)))
(if(<x4)
(let ((newstate
(format t ‘’~%~3a ΓΕΜΙΣΕ ΤΟ ΔΟΧΕΙΟ A~ 3a’’ state newstate))))))
```

Παράδειγμα εφαρμογής: (t1 '(1 3)) -> (1 3) ΓΕΜΙΣΕ ΤΟ ΔΟΧΕΙΟ Α (4 3)
Δηλαδή εκτυπώνεται η αρχική κατάσταση, η περιγραφή του τελεστή και η κατάσταση που προκύπτει μετά την εφαρμογή του.
Ομοίως υλοποιείται και ο Τ2. Ο Τ5 μπορεί να υλοποιηθεί ως εξής:

```
(defun t5 (state)
(let ((x (car state))
(y (second state)))
(if (and (> x 0) (< y 3))
(let (( z (- 3 y)))
(if (or (> x z) (= x z))
(let ((newstate (list (-x (- 3 y))3)))
(print-state-t5 state newstate))
(let ((newstate (list 0 (+ y x))))
(print-state-t5 state newstate)))))))
(defun print-state-t5 (state newstate)
(prog1 newstate
(format t''~3a ΑΔΕΙΑΣΕ ΤΟ ΔΟΧΕΙΟ Α ΣΤΟ Β ~3a ''state newstate)))
```

Η sort-states, που είναι η συνάρτηση που πραγματοποιεί τη διάταξη, για να το κάνει αυτό χρησιμοποιεί (καλεί) δύο άλλες συναρτήσεις, τις find-minstate και remove-state, από τις οποίες η μεν πρώτη βρίσκει το ελάχιστο στοιχείο μιας λίστας καταστάσεων, ενώ η δεύτερη αφαιρεί μια κατάσταση από τη λίστα καταστάσεων. Ουσιαστικά η sort-states, με τη βοήθεια των δύο αυτών συναρτήσεων, υλοποιεί τον αλγόριθμο διάταξης με επιλογή (ένας άλλος τέτοιος αλγόριθμος θα μπορούσε να είναι ο αλγόριθμος bubble sort)

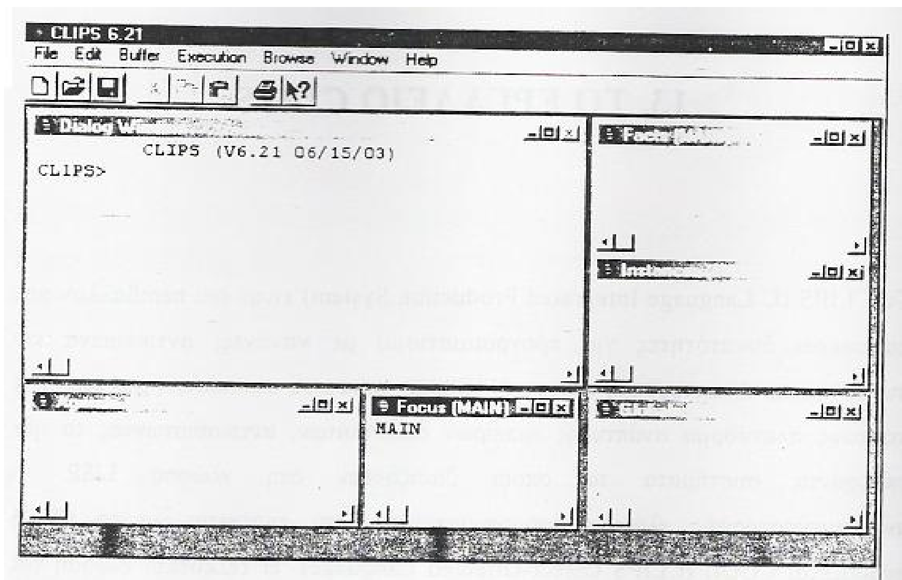
4^{ο} Κεφάλαιο
Το Εργαλείο Clips

Το Clips (C language intergrated production system) είναι ένα περιβάλλον που προσφέρει δυνατότητες για προγραμματισμό με κανόνες, αντικείμενα και συναρτήσεις. Αναπτύχθηκε από τη NASA με σκοπό να αποτελέσει μια χαμηλού κόστους πλατφόρμα ανάπτυξης εμπειρικών συστημάτων, αντικαθιστώντας τα ήδη υπάρχοντα συστήματα τα οποία βασίζονταν στη γλώσσα Lisp. Η αντικειμενικοστραφής γλώσσα προγραμματισμού που παρέχεται με το Clips ονομάζεται Cool (Clips Object-Oriented Language). Η τελευταία έκδοση του είναι η 6.21.

4.1 Περιβάλλον Clips

Το περιβάλλον του Clips εκκινεί σε περιβάλλον MS-WINDOWS με την εκτέλεση εμφανίζει περιβάλλον της γλώσσας όπως φαίνεται στο παρακάτω σχήμα. Για να φορτωθεί το πρόγραμμα Clips επιλέγεται από το μενού file την επιλογή load και μέσα από το σχετικό παράθυρο επιλέγεται το επιθυμητό αρχείο.
Από τα διάφορα παράθυρα που εμφανίζονται μέσω του μενού του Clips δύο είναι σημαντικά: το ένα facts εμφανίζει κάθε στιγμή το περιεχόμενο της λίστας γεγονότων και το άλλο το περιεχόμενο της ατζέντας (agenda). Στο παράθυρο της ατζέντας εμφανίζεται σαν λίστα το σύνολο των κανόνων που ενεργοποιούνται. Μαζί με το όνομα του κανόνα εμφανίζονται και τα fact-index των γεγονότων τα οποία τον ικανοποιούν. Αντίστοιχα στο παράθυρο των γεγονότων εμφανίζονται τα γεγονότα μαζί με το αντίστοιχο fact-index τους. Για να εκτελέσει κάποιο πρόγραμμα γραμμένο σε Clips πρέπει να πληκτρολογηθεί σε πρώτη φάση η εντολή reset. Η εντολή αυτή καταχωρεί στη μνήμη όλα τα γεγονότα τα οποία περιγράφονται στο αρχείο που φορτώθηκε και στη συνέχεια η εντολή run.

4.1 Το περιβάλλον του Clips



4.2 Η Δομή Του Clips

Το Clips είναι ένα διερμηνευόμενο τυπικό σύστημα παραγωγής (interpreted prudaction system), το οποίο την ορθή αλυσίδωση (forward chaining) σαν μέθοδο εξαγωγής συμπερασμάτων.

Τα κύρια μέρη του συστήματος είναι:

- **Η Λίστα γεγονότων (facts list),** η οποία αντιστοιχεί στη μνήμη εργασίας (working memory) των συστημάτων παραγωγής. Όπως δηλώνει και το όνομα της, είναι ο χώρος στον οποίο κατευθύνονται τα (facts), τόσο εκείνα που ορίζονται κατά την εκκίνηση του συστήματος, όσο και εκείνα που δημιουργούνται κατά την εκτέλεση του.
- **Η βάση κανόνων (rule base/knowledge base),** όπου περιέχονται οι κανόνες. Αν και οι κανόνες μπορούν να ορισθούν μέσα από το περιβάλλον του συστήματος, συνήθως είναι αποθηκευμένοι σε κάποιο αρχείο απλού κειμένου (text document), το οποίο φορτώνεται στο σύστημα.
- **Ο μηχανισμός εξαγωγής συμπερασμάτων (inference engine),** ο οποίος ελέγχει τη λειτουργία ολόκληρου του συστήματος. Ο μηχανισμός αυτός προσφέρει ένα πλήθος από *στρατηγικές επίλυσης συγκρούσεων* (confict

resolution strategies) για την *επιλογή* του κανόνα που θα πυροδοτηθεί. Το σύνολο των υποψήφιων κανόνων για πυροδότηση αποτελεί το σύνολο σύγκρουσης ή την ατζέντα (confict set or agenda) του συστήματος. Ένα πρόγραμμα στο Clips είναι ένα σύνολο από κανόνες και γεγονότα και η εκτέλεση του συνίσταται σε μια ακολουθία από πυροδοτήσεις κανόνων, των οποίων οι συνθήκες ικανοποιούνται. Η ικανοποίηση των συνθηκών γίνεται μέσω ταυτοποίησης τους με τα γεγονότα που υπάρχουν στη λίστα γεγονότων. Η εκτέλεση τερματίζεται όταν δεν υπάρχουν άλλοι κανόνες προς πυροδότηση ή όταν κληθεί συγκεκριμένη εντολή τερματισμού. Ο κύκλος λειτουργίας του συστήματος είναι ο τυπικός κύκλος λειτουργίας ενός συστήματος παραγωγής:

1. Εύρεση όλων των κανόνων των οποίων οι συνθήκες ικανοποιούνται και προσθήκη τους στην ατζέντα (agenda-confict set).
2. Αν η ατζέντα είναι κενή τότε η εκτέλεση τερματίζεται.
3. Επιλογή ενός κανόνα με βάση τη στρατηγική επίλυσης ανταγωνισμού (confict resolution) και εκτέλεση του.
4. Επιστροφή στο βήμα 1, εκτός αν υπάρχει εντολή τερματισμού (halt).

4.3 Η σύνταξη του Clips

Η σύνταξη του Clips είναι απλή και θυμίζει εκείνη την γλώσσα προγραμματισμού Lips. Στην ενότητα που ακολουθεί παρουσιάζεται η σύνταξη της γλώσσας που προσφέρει το σύστημα, ξεκινώντας από τα βασικά δομικά στοιχεία της και ακολουθεί η παρουσίαση των μεταβλητών, των γεγονότων και των κανόνων. Θα πρέπει Να σημειωθεί πως στο Clips υπάρχει διαχωρισμός κεφαλαίων και πεζών χαρακτήρων (case-sensitive)

4.4 Τα Δομικά Στοιχεία

Τα βασικά δομικά στοιχεία του Clips είναι τα ακόλουθα:
Σύμβολα (symbols): Σύμβολο μπορεί να είναι οποιαδήποτε ακολουθία χαρακτήρων η οποία ξεκινά με οποιονδήποτε χαρακτήρα εκτός από τους ακόλουθους:
I & ( ) $ + _ και δεν περιέχει τους χαρακτήρες < I / & ( ) ; Για παράδειγμα: cat-and-dog, airway-32, memory-relocation-fail.

Αλφαριθμητικά (strings): Όπως συνηθίζεται, ένα αλφαριθμητικό ξεκινά και τελειώνει με διπλά εισαγωγικά. Για παράδειγμα “This is a program”, “123 Lotus” κλπ.

Αριθμοί (numbers): Το Clips υποστηρίζει τις ακόλουθες αναπαραστάσεις αριθμών (η σημειολογία θεωρείται γνωστή, καθώς είναι κοινή σε πολλές γλώσσες προγραμματισμού): 23, 43, 90, -23, 4e10, 4E10.

Σχόλια: Το Clips θεωρεί σχόλιο ότι ακολουθεί το χαρακτήρα ‘’,’’ μέχρι το τέλος μιας γραμμής. Επίσης, ορισμένες εντολές περιέχουν στη δομή της σύνταξης τους ειδικό όρισμα για εισαγωγή σχολίων, τα οποία συνήθως περικλείονται σε διπλά εισαγωγικά.

4.5 Οι Μεταβλητές

Οι μεταβλητές στο Clips είναι σύμβολα, τα οποία ξεκινούν *απαραίτητα* με τους χαρακτήρες ? ή $? Και με τον περιορισμό ο πρώτος να χαρακτήρας που ακολουθεί να μην είναι αριθμός. Υπάρχουν δυο είδη μεταβλητών, οι *μονότιμες και οι πολλαπλών τιμών.*

- Οι μονότιμες (singlevalue) μεταβλητές μπορούν να πάρουν σαν τιμή μόνο ένα σύμβολο, αριθμό ή αλφαριθμητικό, όπως άλλωστε δηλώνει και το όνομά τους.

Αυτές ξεκινούν με τον χαρακτήρα ?. Για παράδειγμα, το σύμβολο ?varl είναι μια μονό τιμή μεταβλητή με παραδείγματα επιτρεπτών τιμών τα symbol, flight326, monday-meeting, 32, 456, κλπ.

- Οι μεταβλητές πολλαπλών τιμών (multivalue) παίρνουν σαν τιμές ένα ή περισσότερα σύμβολα. Οι μεταβλητές αυτές ξεκινούν με τους χαρακτήρες $?. Παράδειγμα τέτοιας μεταβλητής είναι η $?days_of_week που μπορεί να πάρει σαν τιμή (sun mon tue thu fri sat) ή (name alekos alexandrou), κλπ.

Οι μεταβλητές εμφανίζονται τόσο στις συνθήκες όσο και στις ενέργειες ενός κανόνα, αλλά παίρνουν τιμές κυρίως στις συνθήκες των κανόνων μέσω της διαδικασίας ταυτοποίησης. Ανάθεση τιμής σε μεταβλητή στις ενέργειες ενός κανόνα είναι βέβαια δυνατή με την χρήση κατάλληλης συνάρτησης, αλλά καλό είναι να αποφεύγεται. Τέλος θα πρέπει να τονισθεί πως η εμβέλεια των μεταβλητών θα πρέπει να περιορίζεται στον κανόνα που αυτές εμφανίζονται.

4.6 Τα γεγονότα

Τα γεγονότα αποτελούν την πληροφορία την οποία το σύστημα ‘’γνωρίζει’’ και στην οποία βασίζεται ώστε να εξάγει συμπεράσματα. Τα γεγονότα είναι λίστες από σύμβολα τα οποία περικλείονται σε παρενθέσεις, όπως για παράδειγμα:

(person john smith) (day monday)

(flight_time_arrival 18:45)

(list of days (mon tue wed thu fri sat) list of months

(jan feb mar)).

Όπως προαναφέρθηκε τα γεγονότα αποθηκεύονται στη λίστα γεγονότων και είναι δυνατό να εμφανιστούν στον χρήστη με δύο τρόπους:

- Με την εντολή facts από τη γραμμή εντολών του Clips.

- Μέσω του αντίστοιχου παραθύρου facts window (το περιβάλλον του Clips περιγράφεται στην αντίστοιχη ενότητα στο τέλος του παραρτήματος).

Τα γεγονότα που υπάρχουν στην μνήμη της εργασίας δεν είναι στατικά και μπορούν να εισαχθούν ή να διαγραφούν από αυτή κατά την εκτέλεση του προγράμματος.

Τέλος, κάθε γεγονός το οποίο είναι αποθηκευμένο στην λίστα των γεγονότων έχει ένα χαρακτηριστικό αριθμό(fact index), ο οποίος ορίζεται αυτόματα από το σύστημα κατά την καταχώρηση του γεγονότος από την λίστα. Ο αριθμός αυτός χαρακτηρίζει μοναδικά το γεγονός και χρησιμεύει τόσο στη διαγραφή των γεγονότων όσο και στο να γνωρίζει το σύστημα και ο χρήστης ποια γεγονότα ενεργοποίησαν έναν κανόνα.

Εισαγωγή και διαγραφή γεγονότων

Δύο είναι οι βασικοί τρόποι εισαγωγής γεγονότων στη λίστα γεγονότων: με τη εντολή assert και την εντολή deffacts. Η διαγραφή ενός γεγονότος από την λίστα γίνεται με την εντολή retract.

Assert

Σύνταξη: (assert<fact>)

Η εντολή assert εισάγει ένα γεγονός στην λίστα γεγονότων. Χρησιμοποιείται συνήθως στο δεξιό μέρος των κανόνων.

Παράδειγμα:

Σύνταξη:

```
(deffacts<fact_set_name>        ;όνομα του συνόλου των γεγονότων
 ‘’comment’’                    ;σχόλιο που αφορά το σύνολο των γεγονότων
 (fact1)                        ;τα γεγονότα τα οποία
```

(fact2) ;θα εισαχθούν στην λίστα γεγονότων

(factn)

Η εντολή χρησιμοποιείται για την μαζική εισαγωγή των γεγονότων κατά την εκκίνηση ενός προγράμματος (π.χ ενός έμπειρου συστήματος) στο Clips. Για να εισαχθούν τα γεγονότα που ορίζονται με την παραπάνω εντολή στη λίστα γεγονότων θα πρέπει όχι μόνο να φορτωθεί το αντίστοιχο αρχείο αλλά και να εκτελεστεί η εντολή reset, η οποία παρουσιάζεται στην παρακάτω παράγραφο.

Παράδειγμα :

(deffacts type_a_extinguisher

(extinguisher dry_chemicals type a)

(extinguisher water type a)

(extinguisher water-based type a))

Retract

Σύνταξη:(retract<fact_index>)

Η διαγραφή ενός γεγονός από τη λίστα γίνεται με την εντολή retract. Το όρισμα fact-index είναι ο μοναδικός αριθμός που αντιστοιχεί το Clips στο συγκεκριμένο γεγονός. Η ίδια εντολή μπορεί να χρησιμοποιηθεί τόσο από την γραμμή εντολών του Clips όσο και στις ενέργειες των κανόνων. Όταν όμως χρησιμοποιείται στις ενέργειες των κανόνων πρέπει να χρησιμοποιηθεί ο ειδικός τελεστής ‘’<_’’ για να ανατεθεί το fact-index του συγκεκριμένου γεγονότος σαν τιμή σε μια μεταβλητή.

Παράδειγμα:

1) Αν η λίστα γεγονότων περιέχει τα ακόλουθα:
   f-1 (day mon) f-2 (month jan)

όπου f-1, f-2 είναι οι μοναδικοί αριθμοί τους οποίους δίνει το σύστημα αυτόματα στα γεγονότα (fact index), τότε αν δοθεί στην γραμμή εντολών η εντολή.

(retract 2)

θα αφαιρεθεί το γεγονός (month jan) από την λίστα.

2) Παρακάτω δίνεται ένα παράδειγμα χρήσης της εντολής μέσα σε έναν κανόνα.

Η σύνταξη των κανόνων δίνεται στα επόμενα.

(defrule example

''this is an example rule''

(day monday)

(time noon)

(had lunch)

?x<- (should go to coffee shop); χρήση του ιδικού τελεστή

=>

(assert (went to coffee shop))

(retract ?x) ; διαγραφή του γεγονότος από τη λίστα

Ο ειδικός τελεστής ''<_'' αναθέτει σαν τιμή στην μεταβλητή ?x το fact index του γεγονότος ''(should go to the coffee shop)''. Η εκτέλεση του παραπάνω κανόνα θα έχει ως αποτέλεσμα τη διαγραφή του συγκεκριμένου γεγονότος.

4.7 Οι Κανόνες

Οι κανόνες στο Clips, είναι της μορφής:

If (συνθήκες) then (ενέργειες)

Οι συνθήκες είναι τις περισσότερες φορές ένα σύνολο από γεγονότα τα γεγονότα τα οποί θα πρέπει να υπάρχουν στη λίστα γεγονότων για να ικανοποιείται ο κανόνας. Οι συνθήκες ενός κανόνα μπορούν να περιέχουν μεταβλητές και έτσι

να ταυτοποιούνται με περισσότερα του ενός γεγονότα της λίστας γεγονότων. Αυτό επιτρέπει να ικανοποιείται ο κανόνας με περισσότερους από έναν τρόπους, έχοντας κάθε φορά διαφορετικές αναθέσεις τιμών στις μεταβλητές του. Στην περίπτωση αυτή στην ατζέντα εισάγεται ο κανόνας τόσες φορές, όσοι είναι οι διαφορετικοί τρόποι με τους οποίους ικανοποιείται και φυσικά με διαφορετικές αναθέσεις στις αντίστοιχες μεταβλητές.

Στις *ενέργειες* του κανόνα περιγράφεται το τι θα λάβει χώρα κατά την πυροδότηση του (firing). Οι ενέργειες μπορούν να περιλαμβάνουν οποιαδήποτε συνάρτηση του Clips, όπως για παράδειγμα την εισαγωγή ή την διαγραφή ενός γεγονότος από την λίστα γεγονότων, την εκτύπωση αποτελεσμάτων, υπολογισμό τιμών κλπ. Οι κανόνες ορίζονται μέσω της συνάρτησης *defrule* του Clips.

Defrule

Σύνταξη:

```
(defrule <rule-name>;    ; Όνομα κανόνα (μοναδικό)
‘’<comments>’’            ; Επεξηγηματικά σχόλια για τον κανόνα
(condition 1)             ; Συνθήκη 1
(condition 2)             ; Συνθήκη 2
(condition n)             ; Συνθήκη n
=>
(command 1)               ; Εντολή 1
(command 2)               ; Εντολή 2
(commsnd n)               ; Εντολή n
)
```

Το σύμβολο ‘’=>’’ διαχωρίζει τις συνθήκες από τις ενέργειες του κανόνα. Το όνομα του κανόνα θα πρέπει να είναι μοναδικό. Αν και είναι δυνατό, εντολές της παραπάνω μορφής να εισαχθούν από την γραμμή εντολών του συστήματος,

συνήθως αποθηκεύονται σε κάποιο αρχείο κειμένου (text document) και φορτώνονται στο Clips.

Παράδειγμα:

(defrule fire-type-rule

‘’finding the type of fire’’

(burning ?material)

(material ?material is of type ?type)

=>

(assert (fire-type ?type)))

Ο κανόνας του παραδείγματος συνάγει το τύπο φωτιάς βάσει του υλικού που καίγεται (burning ?material) και του γεγονότος που χαρακτηρίζει το είδος του υλικού (material ?material is of type ?type)

4.8 Ταυτοποίηση

Η διαδικασία ταυτοποίησης έχει σαν στόχο να κάνει ‘’όμοια’’ τη συγκεκριμένη συνθήκη με το αντίστοιχο γεγονός, μέσω κατάλληλων αναθέσεων τιμών στις μεταβλητές. Η διαφορά της από την ενοποίηση που υπάρχει στο λογικό προγραμματισμό είναι ότι στην ταυτοποίηση η ανάθεση τιμών γίνεται μόνο στο ένα μέρος (συνθήκες κανόνα) ενώ στην ενοποίηση η ανάθεση τιμών γίνεται και στις δύο εκφράσεις. Παρακάτω δίνονται κάποια παραδείγματα (πίνακας 2 και πίνακας 3) που αποσαφηνίζουν τη διαδικασία ταυτοποίησης του Clips.

Ιδιαίτερη προσοχή πρέπει να δίνεται όταν στις συνθήκες ενός κανόνα εμφανίζονται μεταβλητές πολλαπλών τιμών. Για παράδειγμα η συνθήκη (List $?a ?b $?c) μπορεί να ταυτοποιηθεί με το γεγονός (list a b c d) με τέσσερις διαφορετικούς τρόπους, όπως δείχνει ο πίνακας 4. Θα πρέπει να τονισθεί ιδιαίτερα ότι ο κανόνας στον οποίο εμφανίζεται η παραπάνω συνθήκη θα μπει

στην agenda τέσσερις φορές, με διαφορετικές κάθε φορά τιμές στις αντίστοιχες μεταβλητές

Πίνακας 2. Παραδείγματα ταυτοποίησης γεγονότων.

Συνθήκη	Γεγονός το οποίο ταυτοποιείται	Αναθέσεις τιμών στις μεταβλητές
(day ?d ?t)	(day fri 12)	?d=fri, ?t=12
(list ?list)	(list a b c d e f)	$?1^{st}$=(a b c d e f)
(car ?c model $?m license ?1)	(car 1 model BMW FIAT license wqw45)	?c=1$ /m=(BMW FIAT ?1=wqw45

Πίνακας 3. Παραδείγματα μη ταυτοποίησης γεγονότων.

Συνθήκη	Γεγονός	Γιατί δεν ταυτοποιείται
(day ?b ?t)	(days fri 12)	Το πρώτο σύμβολο των δύο εκφράσεων είναι διαφορετικό
(list a b $? 1^{st}$)	(list 1 2 c d e f)	Το 2° και 3° σύμβολο είναι διαφορετικά
(car ?c type $?m licence ?1	(car 1 2 type BMW FIAT license wqw45)	Η μονότιμη μεταβλητή ?c δεν μπορεί να πάρει σαν τιμές τα σύμβολα 1 και 2

Πίνακας 4. Παράδειγμα ταυτοποίησης μεταβλητών πολλαπλών τιμών.

$?A	**$?B**	**$?C**
0	A	(B C D)
(A)	B	(C D)
(A B)	C	(D)
(A B C)	D	0

4.9 Οι Βασικές Εντολές Στο Περιβάλλον

Στα επόμενα δίνονται μερικές βασικές εντολές του περιβάλλοντος του Clips, οι οποίες είναι απαραίτητες για την εκτέλεση ενός προγράμματος.

Load

Σύνταξη(load’’<file_name>’’)

Η εντολή αυτή φορτώνει στο σύστημα ένα αρχείο με ορισμούς κανόνων, γεγονότων και συναρτήσεων. Στο παραθυρικό περιβάλλον η ίδια λειτουργία γίνεται μέσω του μενού **file_>loadconstructs**.

Παράδειγμα:

(Load ‘’rules_example.clp’’)

Η παραπάνω εντολή θα φορτώσει τις εντολές που είναι αποθηκευμένες στο αρχείο rules_example_clp. Συνήθως οι εντολές που περιέχονται σε αρχεία είναι εντολές ορισμού κανόνων, εισαγωγής γεγονότων και ορισμού συναρτήσεων.

Reset

Σύνταξη:(reset)

Η εντολή αυτή διαγράφει από την λίστα γεγονότων τα γεγονότα που έχουν πιθανόν δημιουργηθεί κατά την εκτέλεση του προγράμματος και επαναφέρει σε αυτή τα αρχικά γεγονότα που ορίζονται στο αρχείο που έχουμε φορτώσει στο σύστημα. Επίσης εισάγει στη λίστα γεγονότων και το γεγονός (initial-fact). Θα πρέπει να σημειωθεί ότι ακόμη και αν φορτωθούν γεγονότα από αρχείο, αυτά δεν εισάγονται αυτόματα στην λίστα γεγονότων εάν δεν προηγηθεί η εντολή reset.

Run

Σύνταξη:(run)

Εκκινεί την εκτέλεση των κανόνων που έχουν φορτωθεί στην μνήμη. Η εντολή μπορεί να συνταχθεί και σαν (run N), όπου N είναι ο αριθμός των κύκλων λειτουργίας που θα εκτελεστούν. Μετά την εκτέλεση των N κύκλων λειτουργίας το σύστημα θα σταματήσει. Η τελευταία αυτή δυνατότητα αυτή χρησιμοποιείται κυρίως κατά την διαδικασία αποσφαλμάτωσης. Η εκτέλεση μπορεί να συνεχιστεί από το σημείο όπου σταμάτησε από μια νέα εντολή Run ή Run N.

Clear

Σύνταξη:(clear)

Η εντολή αυτή ‘’καθορίζει’’ τελείως το περιβάλλον της γλώσσας διαγράφοντας όλα τα γεγονότα από τη λίστα γεγονότων και όλους τους κανόνες που τυχόν έχουν φορτωθεί στο σύστημα.

4.10 Οι Συναρτήσεις

Η συνάρτηση είναι συντακτικά μια δομή που περικλείεται μέσα σε παρενθέσεις όπου το πρώτο στοιχείο της παρένθεσης είναι το όνομα της συνάρτησης και τα επόμενα στοιχεία, τα ορίσματα αυτής. Το διαχωριστικό σύμβολο μεταξύ των ορισμάτων είναι το κενό. Η κλήση μιας συνάρτησης γίνεται με την δήλωση:

(<όνομα συνάρτησης>όρισμα1 όρισμα2 … όρισμα N)

Για παράδειγμα με την έκφραση (+ 2 3) θα κληθεί η συνάρτηση + με ορίσματα 2 και 3. Το αποτέλεσμα της συνάρτησης θα επιστραφεί στο σημείο που αυτή εμφανίζεται. Για παράδειγμα αν μέσα σε ένα πρόγραμμα Clips υπάρχει η εντολή (assert (the number is (+ 2 3))) τότε το γεγονός το οποίο θα αποθηκευτεί στη μνήμη θα είναι το (the number is 5). Στα επόμενα δίνονται βασικές συναρτήσεις της γλώσσας, όπως οι αριθμητικές συναρτήσεις, οι συναρτήσεις ελέγχου και οι συναρτήσεις για την σύγκριση αριθμών.

Πίνακας 5. Βασικές αριθμητικές συναρτήσεις.

Συνάρτηση	**Σύνταξη**
Πρόσθεση αριθμών	(+<ορίσματα>)
Αφαίρεση αριθμών	(-<ορίσματα>)
Πολλαπλασιασμός	(*<ορίσματα>)
Διαίρεση αριθμών	(/<ορίσματα>)

<u>Βασικές αριθμητικές συναρτήσεις</u>

Το Clips υποστηρίζει όλες τις βασικές αριθμητικές συναρτήσεις (πίνακας 5). Θα πρέπει να σημειωθεί ότι οι συναρτήσεις δέχονται περισσότερα από δύο ορίσματα.

<u>Παραδείγματα:</u>

(+ 2 3 10)   (- 5 2)   (* 2 3 10)

15   3   60

Πίνακας 6. Συναρτήσεις Σύγκρισης Αριθμών.

Συνάρτηση	Σύνταξη	Επιστρέφει True
Αριθμητική ισότητα	(=<αριθμητικά ορίσματα>)	…αν όλα τα ορίσματα είναι ίσα.
Μεγαλύτερο	(><αριθμητικά ορίσματα>)	…αν τα ορίσματα είναι κατά φθίνουσα σειρά.
Μικρότερο	(<<αριθμητικά ορίσματα>)	…αν τα ορίσματα είναι κατά αύξουσα σειρά.
Μεγαλύτερο ή ίσο	(>=<αριθμητικά ορίσματα>)	…αν τα ορίσματα είναι κατά φθίνουσα σειρά (όχι απόλυτη).
Μικρότερο ή ίσο	(<=<αριθμητικά ορίσματα>)	…αν τα ορίσματα είναι κατά αύξουσα σειρά (όχι απόλυτη).
Διάφορο	(0<αριθμητικά ορίσματα>)	…αν τα ορίσματα δεν είναι ίσα.

Σύγκριση αριθμών

Οι συναρτήσεις σύγκρισης αριθμών στο Clips ακολουθούν και αυτές τη σύνταξη των αντίστοιχων εντολών στη Lips. Και στην περίπτωση αυτοί οι συναρτήσεις δέχονται περισσότερα από δυο ορίσματα. Ο πίνακας 6 παραθέτει τις διαθέσιμες συναρτήσεις της κατηγορίας.

Παράδειγμα :

Οι ακόλουθες συναρτήσεις επιστρέφουν TRUE

(>=5 5 4 2 2 1) (<**=** 2 3 3 4 6)(<> 3 5)

Λογικές Συναρτήσεις

Το Clips δίνει την δυνατότητα να εκφραστούν πολύπλοκες συνθήκες κανόνων με την χρήση κλασσικών λογικών συναρτήσεων (πίνακας 7). Οι λογικές συναρτήσεις επιτρέπουν τον συνδυασμό γεγονότων και κατά συνέπεια την πιο συμπαγή διατύπωση κανόνων.

Πίνακας 7. Λογικές συναρτήσεις

Συνάρτηση	Σύνταξη	Επιστρέφει True
Λογικό και	(and<ορίσματα.)	…αν όλα τα ορίσματα της είναι True.
Λογικό ή	(or<ορίσματα>)	…αν ένα τουλάχιστον όρισμα είναι True.
Λογική άρνηση	(not<ορίσματα>)	…αν το όρισμα είναι False.
Ισότητα	(eq<ορίσματα>)	…αν τα ορίσματα είναι ίσα κατά τύπο και τιμή. Τα ορίσματα είναι οποιεσδήποτε συμβολοσειρές.
Ανισότητα	(neq<ορίσματα>)	…αν τα ορίσματα δεν

		είναι ίσα κατά τύπο και τιμή. Τα ορίσματα είναι οποιοσδήποτε συμβολοσειρές.

Παραδείγματα:

Οι ακόλουθες συναρτήσεις επιστρέφουν TRUE

(and (>= 5 5) (> 3 2 1) (= 10 10)

(and (not = 10 7) (= 9 9))

Στο παρακάτω παράδειγμα που ακολουθεί, φαίνεται η χρήση της λογικής συνάρτησης ή στις συνθήκες. Χωρίς τη χρήση της συνάρτησης θα έπρεπε να γραφούν δύο κανόνες, ένας για κάθε περίπτωση.

```
 (defrule days
   (month jan feb)
    (or (hour 12)    (hour 13))
=>
(assert  (day is monday noon time))
```

Συναρτήσεις Ελέγχου Τύπου

Σε πολλές περιπτώσεις είναι δυνατό να πρέπει να εξακριβωθεί ο τύπος της τιμής μιας μεταβλητής, ώστε να γίνει η κατάλληλη επεξεργασία. Ο πίνακας 8 παρουσιάζει τις διαθέσιμες συναρτήσεις για έλεγχο τύπων.

Πίνακας 8. Συναρτήσεις ελέγχου τύπου

Συναρτήσεις	**Σύνταξη**	**Επιστρέφει True**
Αριθμών	(numberp<όρισμα>)	…αν το όρισμα είναι αριθμός.
Κινητικής υποδιαστολής	(floatp<όρισμα>)	…αν το όρισμα είναι αριθμός κινητής.
Ακεραίου	(integerp<όρισμα>)	..αν το όρισμα είναι ακέραιος αριθμός.
Αλφαριθμητικό	(stringp<όρισμα>)	…αν το όρισμα είναι αλφαριθμητικό.
Σύμβολο	(symbol<όρισμα>)	…αν το όρισμα είναι σύμβολο.

Παράδειγμα:

Οι ακόλουθες συναρτήσεις επιστρέφουν TRUE:

(numberp 4)   (symbolp day)

Συναρτήσεις χειρισμού πολλαπλών τιμών.

Create$

Σύνταξη:(create$<ορίσματα>)

Επιστρέφει μια τιμή που μπορεί να ανατεθεί σε μεταβλητή πολλαπλών τιμών (πολλαπλή τιμή), την οποία δημιουργεί από τα ορίσματα.

Παράδειγμα:

Η εντολή

(create$ the day is (grey as it was))

επιστρέφει:

(the day is grey as it was)

Explode$

Σύνταξη (explode$<string>)

Επιστρέφει μια πολλαπλή τιμή την οποία δημιουργεί από το αλφαριθμητικό. Η εντολή είναι όμοια με την προηγούμενη, διαφέροντας μόνο στο είδος των ορισμάτων που δέχεται.

Παράδειγμα:

Η εντολή

(explode$ ''the night was blue'')

θα επιστρέψει:

(the night was blue).

Implode$

Σύνταξη:(implode$<multivalue>)

Επιστρέφει το αντίστοιχο αλφαριθμητικό από μια πολλαπλή τιμή.

Παράδειγμα:

Η εντολή

(implode$ (explode$ ''the night was blue''))

θα επιστρέψει ''the night was blue''.

Nth$

Σύνταξη:(nth$ N <multivalue>)

Επιστρέφει το N-οστό πεδίο μιας πολλαπλής τιμής.

Παράδειγμα:

Η εντολή

(nth$ 2 (create$ 3 4 5))

θα επιστρέψει την τιμή 4..

Members$

Σύνταξη: (member$ <symbol><multivalue>)

Επιστρέφει τη θέση του πεδίου <symbol> μέσα στην τιμή <multivalue> εφόσον αυτό υπάρχει. **Εάν αυτό δεν υπάρχει επιστρέφει False.**

Παράδειγμα:

Η εντολή

(member$ c (create$ a b c))

θα επιστρέψει την τιμή 3.

First$

Σύνταξη: (first$ <multivalue>).

Επιστρέφει το πρώτο στοιχείο μιας πολλαπλής τιμής, αλλά σε μορφή λίστας.

Παράδειγμα:

Η εντολή

(first$ (create$ 3 1 5))

θα επιστρέψει (3).

Rest$

Σύνταξη: (rest$<multivalue>).

Επιστρέφει τα υπόλοιπα στοιχεία εκτός από το πρώτο στοιχείο μιας πολλαπλής τιμής.

Παράδειγμα:

Η εντολή

(rest$ (create$ 3 4 5))

θα επιστρέψει (4 5).

Συναρτήσεις εισόδου-εξόδου

Printout

Σύνταξη: (printout<device><expression>).

Αποστέλλει την έκφραση <expression> στην συγκεκριμένη συσκευή <device>. Η συσκευή μπορεί να είναι οποιαδήποτε συσκευή I/O, για παράδειγμα ένα αρχείο ή η οθόνη. Στην περίπτωση που η συσκευή είναι η οθόνη, η εντολή συντάσσεται με τιμή t (terminal) στην παράμετρο <device>.

Παράδειγμα:

(printout t ''the day was'' ?type crlf)

όταν εκτελεστεί η παραπάνω εντολή και η τιμή της μεταβλητής ?type είναι για παράδειγμα sunny, τότε θα τυπωθεί στην οθόνη το μήνυμα: The day was sunny. Το σύμβολο crlf δηλώνει ότι μετά την εκτύπωση του μηνύματος στην οθόνη ο δρομέας θα αλλάξει γραμμή

Read

Σύνταξη: (read)

Η εντολή εισάγει από την προκαθορισμένη συσκευή εισόδου (standard input) το επόμενο σύμβολο. Συνήθως χρησιμοποιείται σε συνδυασμό με την εντολή bind, η οποία όπως θα δούμε στα επόμενα αναθέτει τιμή σε μια μεταβλητή στις ενέργειες ενός κανόνα.

Παράδειγμα:

```
(defrule get-user-answer
''get the users answer''
=>
(printout  t  ''what is your first name: '')
(bind  ?name (read))
(assert  (user-name  ?name))
)
```

Όπως φαίνεται στο παραπάνω παράδειγμα η εντολή read θα ''διαβάσει'' μεταβλητή ?name.

Συνάρτηση ανάθεσης τιμής σε μεταβλητή

Bind

Σύνταξη:(bind <variable><value>)

Η εντολή χρησιμοποιείται για να ανατεθεί η τιμή <value>σε μια μεταβλητή <variable> στις ενέργειες των κανόνων.

Παράδειγμα:

```
(defrule rule
''examle rule''
(oldcost  ?cost)
(newcost  ?newcost)
=>
(bind  ?total_cost  (+  ?new cost  ?oldcost))
(assert  (cost  ?total_cost))
```

(printout t ‘’the total cost is ‘’ ?total_cost crlf)

Με την χρήση της εντολής bind στο παραπάνω παράδειγμα, υπολογίζεται το συνολικό κόστος (άθροισμα των μεταβλητών ?oldcost ?newcost) μόνο μια φορά, το αποτέλεσμα αποθηκεύεται στη συνέχεις σαν όρισμα σε δύο διαφορετικές συναρτήσεις.

Συναρτήσεις Ελέγχου Ροής Προγράμματος

While

Σύνταξη:

```
(While  (condition) do        ;συνθήκη
(command 1)                   ;εντολή 1
(command 2)                   ;εντολή 2
(command n)                   ;εντολή n)
```

Η συνάρτηση έχει την ίδια ακριβώς λειτουργία με τις αντίστοιχες συναρτήσεις των άλλων γλωσσών προγραμματισμού, δηλαδή όσο ικανοποιείται μια συνθήκη, εκτελείται ένα σύνολο συναρτήσεων. Πολλές φορές στην περίπτωση είναι απαραίτητη η χρήση της εντολής bind για να αλλάξει τις τιμές μεταβλητών που συμμετέχουν στη συνθήκη.

Παράδειγμα:

Έστω ότι υπάρχει ένα γεγονός στην λίστα γεγονότων της μορφής (num 1). Ο επόμενος κανόνας θα τυπώσει διαδοχικά όλους τους αριθμούς από το 1 έως το 9, με την φράση ‘’the num is:’’ σαν πρόθεμα.

```
(defrule test
(num ?n)
=>
```

```
(while  (<  ?n  10)
(printout  t  ''the num is:  ''  ?n  crlf)
(bind  ?n   (+  1  ?)
```

if.....then.....else

Σύνταξη:

```
(if  (condition)  ;συνθήκη
then
(command 1)    ;εντολή 1
(command 2)    ;εντολή 2
(command m)   ;εντολή m
else
(command a)    ;εντολή a
(command n)    ;εντολή n
)
```

Η κλασσική συνάρτηση ελέγχου η οποία είναι κοινή σχεδόν σε όλες τις γλώσσες προγραμματισμού. Εκτελεί, υπό συνθήκη, κάποια σύνολα εντολών.

Παράδειγμα:

Ο επόμενος κανόνας θα τυπώσει positive,negative ή zero, αν ο αριθμός ?n είναι αντίστοιχα μεγαλύτερος, μικρότερος ή ίσος με το μηδέν.

```
(defrule sign
(num ?n)
=>
(if (> ?n 0)
```

```
then (printout  t  ‘’positive’’  crlf))
else   (if  (<  ?n  0)
then  (printout  t  ‘’negative’’  crlf)
else   (printout  t  ‘’zero’’         crlf)
```

4.10.1 Ο Ορισμός συναρτήσεων στο CLIPS

Εκτός από τις προκαθορισμένες συναρτήσεις, μέρος των οποίων παρουσιάστηκε στις προηγούμενες παραγράφους, η γλώσσα δίνει τη δυνατότητα να ορίσει ο χρήστης τις δικές του συναρτήσεις. Ο ορισμός αυτών των συναρτήσεων γίνεται μέσω μιας ειδικής συνάρτησης, της deffunction.

Deffunction

Σύνταξη:

```
(deffunction <function name>   ;το όνομα της νέας μεταβλητής
 (<variables>)                  ;μια λίστα με μεταβλητές που είναι
                                ;ορίσματα της συνάρτηση
 (command 1)                    ;η πρώτη εντολή
 (command n)                    ;η τελευταία εντολή)
```

Θα πρέπει εδώ να σημειωθεί ότι η συνάρτηση επιστρέφει την τιμή της τελευταίας εντολής που εκτελείται. Η εντολή αυτή μπορεί να είναι μια συνάρτηση, ένα σύμβολο, ή μια μεταβλητή της οποίας η τιμή επιστρέφεται. Οι ορισμοί των συναρτήσεων, όπως και
οι ορισμοί γεγονότων και κανόνων, μπορούν να παρέχονται σε ένα αρχείο κειμένου
και να φορτώνονται μαζικά στο σύστημα.

Π.χ :

Η ακόλουθη συνάρτηση υπολογίζει τη μέση τιμή τεσσάρων αριθμών:

```
(deffuction  mean-value
(?v1   ?v2   ?v3   ?v4
 ( /  (+    ?v1   ?v2   ?v3   ?v4)   4)
```

και η εκτέλεση της θα δοθεί:

```
(mean-value  4  5  6  7 )
```

4.11 Οι περιορισμοί στις συνθήκες των κανόνων

Εκτός από απλή ταυτοποίηση γεγονότων οι συνθήκες των κανόνων μπορούν να περιέχουν και περιορισμούς, καταλήγοντας έτσι σε πιο σύνθετες και εκφραστικές μορφές. Οι περιορισμοί εκφράζονται είτε με τη χρήση συγκεκριμένων συνδετικών ή με την εισαγωγή συνθηκών υπό μορφή συναρτήσεων (test conditions) και αποτελούν ένα ακόμη επίπεδο ελέγχου ικανοποίησης των κανόνων. Ο πίνακας 8 περιέχει τα διαθέσιμα συνδετικά της πρώτης κατηγορίας.

Πίνακας 9. Συνδετικά συνθηκών κανόνων

Λογική Πράξη	**Συνδετικό**
Λογική άρνηση	-
Λογική διάζευξη (ή)	I
Λογική σύζευξη (και)	&

Για παράδειγμα η συνθήκη (month Jan day ~mon) ικανοποιείται εάν υπάρχει ένα ή περισσότερα γεγονότα στη λίστα γεγονότων με τη μορφή (month Jan day <symbol>) όπου το τελευταίο σύμβολο της λίστας δεν είναι mon. Με παρόμοιο τρόπο χρησιμοποιείται το συνδετικό ¨I¨ (or): η συνθήκη (hour 12 I 13) , μπορεί να ταυτοποιηθεί είτε με το γεγονός
(hour 12) ή με το (hour 13).
Για παράδειγμα, ο κανόνας:

```
(defrule days
(month Jan  day-mon)
 (hour 12I13)
  =>
 (printout t ¨ Day is not Monday, but  it is noon time ! ! ! ¨
crlf)
```

Θα ενεργοποιηθεί εάν υπάρχει ένα γεγονός στη λίστα όπου το τελευταίο του πεδίο να μην είναι mon και ένα γεγονός (hour 12) ή (hour13).

Το συνθετικό ¨&¨ (and) χρησιμοποιείται συνήθως σε συνδυασμό με άλλους περιορισμούς. Σύμφωνα με τα παραπάνω, ο κανόνας του προηγούμενου παραδείγματος μπορεί να γραφτεί με την ακόλουθη μορφή:

```
(defrule  days
(month Jan day   ?day4-mon)
(hour 12I13)
 =>
(printout t ´´ Day is ´´ ?day ´´ , but it is noon time!  !  !´´
 crlf ) )
```

Η συνθήκη ?day&-mon δηλώνει ουσιαστικά ¨στη θέση αυτή πρέπει να υπάρχει κάποιο σύμβολο και το σύμβολο αυτό να μην είναι το mon¨. Η ουσιαστική διαφορά μεταξύ των συνθηκών των παραπάνω κανόνων είναι ότι στη δεύτερη περίπτωση υπάρχει και η συγκεκριμένη τιμή του πεδίου η οποία ενεργοποίησε τον κανόνα σαν τιμή στη μεταβλητή ?day, όπως φαίνεται στο παράδειγμα.

Έστω ότι υπάρχουν τα παρακάτω γεγονότα στη λίστα γεγονότων του CLIPS.

```
(deffacts elements
(element a)
(element b)
 (element c)
  )
```

και απαιτείται να τυπωθούν στην οθόνη οι διαφορετικοί συνδυασμοί τους ανά δύο. Ο κανόνας:

```
 (defrule  Cartesian
 (element  ?a)
 (element  ?b)
  =>
 (printout t ´´ Elements :´´  ?a ´´  ´´  ?b  crlf)
   )
```

θα τυπώσει και τους συνδυασμούς (Elements: a a) (Elements: b b) (Elements c c) οι οποίοι δεν είναι επιθυμητοί. Για να αποφευχθεί η παραπάνω συμπεριφορά εισάγονται περιορισμοί στη συνθήκη του κανόνα:

```
  (defrule  Cartesian
  (element  ?a)
  (element  ?bS-?a)               ;η μεταβλητή ?b παίρνει τιμές #από την ?a
    =>
  (printout t “Elements: “ ?a “  “  ?b crlf)             )
```

Ο συνδυασμός των συνδετικών επιβάλλει στην τιμή της μεταβλητής ?b να είναι διαφορετική από εκείνη της ?a.

Εκτός από τη χρήση συνδετικών επιτρέπεται και η χρήση λογικών συναρτήσεων στις συνθήκες των κανόνων. Οι λογικές συναρτήσεις παρουσιάστηκαν στα προηγούμενα και είναι οι (and …), (or …) και (not …) . Προφανώς όλοι οι συνδυασμοί τους είναι επιτρεπτοί. Για παράδειγμα οι δυο επόμενοι κανόνες :

```
 (defrule emergl
 (emerg type fire)
   =>
  (assert  (evacuate building) )
    )
  (defrule emerg2
```

```
 (fire drill)
   =>     (assert   (evacuate building) )
```

Μπορούν να γραφούν σαν ένας :

```
(defrule emerg1
(or   (emerg type fire) (fire drill) )
 =>
(assert (evacuate building) )   )
```

Θα πρέπει να σημειωθεί ότι, παρόμοια με τη γλώσσα PROLOG , αν εμφανίζονται μεταβλητές μέσα σε κάποιο (not) τότε δεν γίνεται ανάθεση τιμών σε αυτές.

Για παράδειγμα, ο επόμενος κανόνας προκαλεί λάθος εκτέλεσης εφόσον η μεταβλητή ?b δεν θα πάρει τιμή στις συνθήκες του κανόνα και κατά συνέπεια η εντολή εκτύπωσης δε θα λειτουργήσει σωστά :

```
(defrule wrong-rule
 (not  (element  ?b) )
  =>
 (printout t " not element " ?b crlf) )
```

Τέλος θα πρέπει να αναφερθεί ότι είναι δυνατό να γίνουν και συγκρίσεις στις συνθήκες ενός κανόνα, χρησιμοποιώντας τη συνάρτηση (test). Για παράδειγμα κανόνας cartesian μπορεί να γραφτεί :

```
(defrule cartesi
(element ?a)
(element ?b)
(test  (neq ?a ?b) )    ; η μεταβλητή ?b παίρνει τιμές # από την ?a
 =>
 (printout t  "Elements: " ?a "  "  ?b crlf)
  )
```

Όπου ο έλεγχος για τις διαφορετικές τιμές των μεταβλητών γίνεται στη τελευταία γραμμή των συνθηκών με τη χρήση της σχετικής εντολής.

Παράδειγμα:

Ένα ενδιαφέρον παράδειγμα που συνδυάζει όλα τα παραπάνω είναι το ακόλουθο.

Έστω ότι από μια λίστα βιβλίων που παριστάνεται με γεγονότα της μορφής:

```
(deffacts prices
 (book A price 34)
  (book B price 20)
  (book C price 68)
```

Απαιτείται να επιλεγεί το πιο ακριβό. Ο κανόνας που θα επιστρέψει το ακριβότερο βιβλίο είναι ο ακόλουθος:

```
(defrule select-exp2
 (book  ?Book  price  ?price)
 (not   (and   (book  ?Book2&-?Book price2)
 (test  (>  ?price2  ?price) )  )  )
  =>
 (printout t  “The Book with the highest price is: “
  ?Book crlf)
  )
```

Ο παραπάνω κανόνας να ερμηνευτεί ως ΄΄ Εάν υπάρχει βιβλίο?Book με τιμή?price? και δεν υπάρχει άλλο βιβλίο ?Book2 διαφορετικό από το ?Book το οποίο έχει μεγαλύτερη τιμή από την ?price τότε τύπωσε το βιβλίο ?Book ΄΄.

<u>4.12 Πρότυπα Γεγονότων</u>

Ορισμοί

Μέχρι τώρα στο σύνολο των παραδειγμάτων που αναφέρθηκαν δεν υπήρχε η ανάγκη για ορισμό μεγάλων σε μέγεθος γεγονότων. Ωστόσο σε μεγαλύτερα

προγράμματα εμφανίζεται σχεδόν πάντα η ανάγκη αναπαράστασης της διαθέσιμης πληροφορίας με τέτοια γεγονότα. Για παράδειγμα, έστω ότι υπάρχει μια βάση δεδομένων μαθητών όπου καταχωρούνται διαφορά στοιχεία τους, κάθε εγγραφή της οποίας αναπαρίσταται με ένα γεγονός της μορφής:

(student name <name> surname <surname> sex <sex> age <age> classes <classes>)

Όπως για παράδειγμα:

(student name John surname Ref sex male age 28 classes math physics chem)

Η χρήση του παραπάνω γεγονότος ως συνθήκη σε κάποιο κανόνα, ή ακόμη και οι πιθανές αλλαγές που θα έπρεπε να γίνουν στα στοιχεία της βάσης, απαιτούν να γραφούν κανόνες με όλες τις παραμέτρους του. Για παράδειγμα ένας κανόνας που απλώς τυπώνει τα ονόματα των μαθητών σε ένα τέτοιο πρόγραμμα γράφεται:

```
 (defrule print-students
 (student name  ?n sex  ?s age  ?a classes  $?cl)
 =>
(printout t “Student:  “  ?n  crlf)
    )
```

Το CLIPS προσφέρει ένα εναλλακτικό τρόπο δημιουργίας και διαχείρισης τέτοιων γεγονότων με την οποία μπορεί να οριστεί η μορφή που θα έχουν τα γεγονότα σε ένα πρόγραμμα. Κάθε πρότυπο έχει σύνολο από ιδιοκτήτες (slots), στις οποίες μπορούν να ανατεθούν τιμές αυτόνομα, ενώ επιπλέον μπορεί να ορισθούν και οι τύποι των τιμών αυτών ώστε να γίνονται οι απαραίτητοι έλεγχοι. Ο ορισμός προτύπων γεγονότων γίνεται με τη συνάρτηση deftemplate.

Deftemplate

Σύνταξη:

```
(deftemplate <template name>
(slot <slotnamel> (type <typel>))   ;ιδιότητα 1 τύπου 1
 (multislot<slotname2>  (type<type2>)) ;ιδιότητα 2 τύπου 2
  (slot<slotnameN> (type <typeN>))       ;ιδιότητα N τύπου N
```

όπου slotnameN είναι το όνομα της ιδιότητας και το όρισμα (type <typeN>) καθορίζει τον τύπο της τιμής της συγκεκριμένης ιδιότητας. Ο καθορισμός τύπου είναι προαιρετικός και αν δεν ορισθεί για κάποια ιδιότητα αυτή μπορεί να δεχθεί τιμή οποιουδήποτε τύπου. Υπάρχουν δύο είδη ιδιοτήτων: οι ιδιότητες slot που μπορούν να πάρουν τιμή μόνο ένα σύμβολο ή αριθμό και οι ιδιότητες multislot οι οποίες δέχονται πολλαπλές τιμές. Μπορεί να θεωρηθεί ότι οι πρώτες αντιστοιχούν στις μονότιμες μεταβλητές, ενώ οι δεύτερες στις μεταβλητές πολλαπλών τιμών.

Π.χ:

Χρησιμοποιώντας τα παραπάνω, ο ορισμός του προτύπου για τη βάση δεδομένων των μαθητών θα είναι:

```
 (deftemplate student
 (slot name)
 (slot surname)
 (slot age)
         (multislot classes)
          )
```

Και ο κανόνας ο οποίος τυπώνει τα ονόματα όλων των αρένων μαθητών

```
 (defrule print-students
 (student (name  (name  ?name)  (sex male))
  =>
  (printout t  “Student:     “   ?name crlf)
   )
```

Όπως παρατηρείται στον παραπάνω κανόνα, δεν χρειάζεται να γραφεί στις συνθήκες ολόκληρο το γεγονός, αλλά μόνο το όνομα του προτύπου και τα

ονόματα των ιδιοτήτων που ενδιαφέρουν. Ο **Πίνακας 10** περιέχει τούς διαθέσιμους τύπους τιμών για τις ιδιότητες προτύπων.

Πίνακας 10. Τύποι τιμών για τις ιδιότητες των προτύπων

ΤΥΠΟΣ	Η ιδιότητα μπορεί να περιέχει
SYMBOL	Σύμβολα
STRING	Αλφαριθμητικά
LEXEME	Σύμβολα ή αλφαριθμητικά
INTEGER	Ακέραιες τιμές
FLOAT	Πραγματικές τιμές
NUMBER	Ακέραιες τιμές ή πραγματικές
?VARIABLE	Τιμές οποιουδήποτε τύπου

Σύμφωνα με τα παραπάνω, ένα ολοκληρωμένο πρότυπο που αφορά τους μαθητές μπορεί να γραφεί:

```
(deftemplate student
(slot name  (type  SYMBOL) )
(slot surname  (type SYMBOL) )
(slot sex  (type SYMBOL) )
(slot age  (type INTEGER) )
 (multislot classes  (type SYMBOL)
  )
```

Στην περίπτωση που επιχειρηθεί μη επιτρεπτή τιμή σε κάποια ιδιότητα, το σύστημα θα επιστρέψει μήνυμα λάθους.

Εκτός από τον επιτρεπτό τύπο της τιμής της ιδιότητας , το CIIPS δίνει τη δυνατότητα να απαριθμηθούν οι επιτρεπτές τιμες που μπορεί να πάρει αυτή. Η απαρίθμηση γίνεται χρησιμοποιώντας τη δήλωση (allowed-prefix <values>), όπου το prefix μπορεί να πάρει μία από τις τιμές symbols, strings, lexemes, integers, floats, numbers, values όπως φαίνεται παρακάτω:

(allowed-symbols <symbols>)

(allowed-strings <stings>)

(allowed-lexemes <symbols or strings>)

(allowed-integers <integers>)

(allowed-floats <floats>)

(allowed-numbers <numbers or floats>)

(allowed-values <values>)

Για παράδειγμα στο πρότυπο student η παρακάτω δήλωση περιορίζει τις τιμές της ιδιότητας sex σε male ή female:

```
(deftemplate student
(slot sex  (type SYMBOL)  (allowed-symbols male  female) )
```

Ειδικά για τις αριθμητικές τιμές είναι δυνατό να ορισθεί το επιτρεπτό εύρος τιμών που επιδέχεται η ιδιότητα, με μια δήλωση του τύπου (range <min value> <max value>). Για παράδειγμα, μπορεί να οριστεί ότι οι επιτρεπτές ηλικίες (ιδιότητα age) των μαθητών της βάσης είναι μεταξύ 18 και 60, σε μια δήλωση της μορφής:

```
(deftemplate student
(slot age  (type INTEGER)  (range 18 60) )
 )
```

Εάν η ιδιότητα δεν πρέπει να έχει άνω ή κάτω όριο τότε εισάγεται το σύμβολο ?VARIABLE στην αντίστοιχη θέση. Η παρακάτω δήλωση θέτει μόνο κάτω όριο στο εύρος τιμών της ιδιότητας:

```
(deftemplate student
(slot age  (type INTEGER)  (range 18    ?VARIABLE) )
 )
```

Στις ιδιότητες πολλαπλών τιμών (multislots) μπορεί να περιοριστεί το πλήθος των συμβόλων που μπορούν να δοθούν σαν τιμή, σε μια δήλωση της μορφής

(cardinality <min> <max>).

Για παράδειγμα, αν πρέπει να δηλωθεί ότι τα επιτρεπτά μαθήματα ανα μαθητή είναι το πολύ τέσσερα, ο ορισμός του προτύπου student γίνεται:

```
(deftemplate student
(multislot classes  (type SYMBOL)  (cardinality 1  4) )
 . )
```

Τέλος είναι δυνατή η δήλωση προκαθορισμένων τιμών (default) σε όλες τις ιδιότητες. Οι τιμές αυτές εισάγονται με τη δήλωση (default <value>). Για παράδειγμα:

```
(deftemplate student
 (slot age  (type INTEGER)  (default  18))
 )
```

Μέσα σε μια δήλωση default μπορεί να δοθούν δύο σύμβολα με ειδική σημασία:

- (default ?DERIVE): Στην περίπτωση αυτή δίνεται σαν τιμή στην ιδιότητα μια από τις επιτρεπόμενες τιμές, όπως αυτές προκύπτουν από τους περιορισμούς που υπάρχουν για την ιδιότητα. Αυτή είναι και η εξ ορισμού δήλωση προκαθορισμένης τιμής.
- (default ?NONE): Δηλώνει ότι δεν υπάρχει προκαθορισμένη τιμή για την ιδιότητα, που σημαίνει ότι πρέπει οπωσδήποτε να δοθεί τιμή στη συγκεκριμένη ιδιότητα κατά την δημιουργία του αντίστοιχου γεγονότος. Σε αντίθετη περίπτωση το γεγονός δεν θα εισαχθεί στη λίστα γεγονότων.

Παράδειγμα:

```
(deftemplate student
 (slot name   (type SYMBOL)         (default   ?NONE) )
 )
```

Στο παραπάνω παράδειγμα δε θα εισαχθεί γεγονός που να βασίζεται στο πρότυπο student, αν δεν υπάρχει τιμή για την ιδιότητα name.

4.13 Ενεργοποίηση / απενεργοποίηση Ελέγχου τιμών

Το CLIPS διαθέτει δύο επίπεδα ελέγχου τιμών που εισάγονται στα πρότυπα γεγονότων, το στατικό και το δυναμικό. Ο στατικός έλεγχος τιμών αφορά τα γεγονότα τα οποία εισάγονται στο σύστημα από αρχείο , μέσα από τις δηλώσεις deffacts , defrule , κλπ. Ενεργοποιείτε από την εντολή:

(set-static-constraint-checking TRUE/FALSE)

Το FALSE απενεργοποιεί το στατικό έλεγχο των τιμών, ενώ το TRUE έχει το αντίθετο αποτέλεσμα. Η τρέχουσα τιμή επιστρέφεται με την εντολή:

(get-static-constraint-checking)

Στο δυναμικό έλεγχο κάθε γεγονός που εισάγεται στη λίστα γεγονότων ελέγχεται για την εγκυρότητα των τιμών του, ακόμη και αν η εισαγωγη αυτή γίνεται κατά την εκτέλεση του προγράμματος. Η ενεργοποίηση / απενεργοποίηση αυτού του επιπέδου ελέγχου γίνεται με εντολές παρόμοιες με τις προηγούμενες, δηλαδή:

(set-dynamic-constraint-checking- TRUE/FALSE)
ενώ η τρέχουσα τιμή δίνεται με την εντολή:
(get-dynamic-constraint-checking)

Σε περίπτωση ύπαρξης σφάλματος στις τιμές που δίνονται σε κάποια ιδιότητα το CLIPS επιστρέφει μήνυμα λάθους και σταματά την εκτέλεση των κανόνων.

4.14 Αλλαγή τιμής μιας ιδιότητας

Η αλλαγή της τιμής μιας ιδιότητας προτύπου γίνεται με την εντολή modify.

Modify

Σύνταξη: (modify <fact-index> (<slot> <νέα τιμή slot>))

Η εντολή μεταβάλλει την τιμή της ιδιότητας <slot> σε <νέα τιμή slot> στο γεγονός με αριθμό fact-index. Θα πρέπει να σημειωθεί ότι η εντολή δίνει τη δυνατότητα αλλαγής της τιμής κάθε ιδιότητας ενός γεγονότος που ακολουθεί κάποιο πρότυπο, ανεξάρτητα από τις άλλες ιδιότητες. Η χρήση του χαρακτηριστικού αριθμού για την εφαρμογή της εντολής επιβάλλει τη χρήση του τελεστή <- στις συνθήκες του κανόνα.

Πχ:

Ο ακόλουθος κανόνας αλλάζει ταυτόχρονα τις τιμές των ιδιοτήτων name και classes στα γεγονότα student που βρίσκονται στη μνήμη εργασίας.

```
(defrule change-information
?x <- (student (name ?name) )
 =>
(modify ?x (name noname) (classes (create$ math physics chem) ) )
 )
```

Πρέπει να τονιστεί ότι η εντολή modify αφαιρεί το παλαιό γεγονός από τη λίστα και προσθέτει σε αυτή ένα καινούργιο με τις απαραίτητες αλλαγές. Αυτό σημαίνει ότι ο παραπάνω κανόνας θα εκτελείται επ' άπειρων, καθώς θα υπάρχει πάντα σε κάθε κύκλο ένα ΄΄νέο΄΄ γεγονός (με νέο fact index) το οποίο θα ικανοποιεί τις συνθήκες του. Μια ορθότερη εκδοχή του κανόνα θα ήταν:

```
(defrule change-information
?x <- (student (name ?name) )
 =>
 (modify ?x (name noname) (classes (create$ math physics chem) ) )
  )
```

4.15 Στρατηγικές Επίλυσης Συγκρούσεων

Όπως έχει αναφερθεί στην εισαγωγή, όλοι οι κανόνες των οποίων οι συνθήκες ικανοποιούνται εισάγονται στην ατζέντα (agenda), η οποία αντιστοιχεί στο σύνολο συγκρούσεων (conflict set) των κλασσικών συστημάτων παραγωγής.

Από το σύνολο των κανόνων αυτών επιλέγεται κάθε φορά ένας κανόνας, ο οποίος και πυροδοτείτε με βάση δύο κριτήρια: την προτεραιότητα των κανόνων και τη στρατηγική επίλυσης συγκρούσεων. Στην πράξη, η ατζέντα συμπεριφέρεται σαν μια στοίβα (stack) όπου όσο μεγαλύτερη προτεραιότητα έχει ένας κανόνας τόσο πιο ψηλά βρίσκεται σε αυτή. Ο κανόνας ο οποίος εκτελείται είναι εκείνος ο οποίος βρίσκεται στην κορυφή της στοίβας. Ένας νέος κανόνας τοποθετείται στην ατζέντα σύμφωνα με τα ακόλουθα κριτήρια:

1. Οι νέοι μπαίνουν ¨πάνω¨ από τους όλους τους κανόνες με μικρότερη ή ίση προτεραιότητα (salience) και ¨κάτω¨ από όλους τους κανόνες με μεγαλύτερη προτεραιότητα.
2. Στους κανόνες με ίδια προτεραιότητα χρησιμοποιείται η τρέχουσα στρατηγική επίλυσης συγκρούσεων για να καθοριστεί η σειρά τους.
3. Εάν κάποιοι κανόνες ενεργοποιήθηκαν από το ίδιο σύνολο γεγονότων και τα προηγούμενα βήματα δεν μπόρεσαν να ορίσουν μια σειρά, τότε δίνεται σε αυτούς μια αυθαίρετη σειρά (όχι τυχαία), η οποία εξαρτάται από την υλοποίηση του συστήματος.

4.16 Προτεραιότητα Κανόνων

Η προτεραιότητα ενός κανόνα μέσω της δήλωσης

```
(declare (salience<number>))
```

στη συνάρτηση ορισμού του κανόνα. Π.χ.

```
(defrule cartesian
(declare (salience 30) )   ;ορισμός προτεραιότητας κανόνα
(element  ?a)
(element  ?b)
  =>
(printout  t  “Elements: “  ?a “  “  ?b crlf)
```

Η αριθμητική τιμή καθορίζει την προτεραιότητα του κανόνα. Όσο μεγαλύτερη είναι η τιμή αυτή τόσο μεγαλύτερη είναι και η προτεραιότητα του συγκεκριμένου

κανόνα. Οι επιτρεπτές τιμές προτεραιότητας είναι από -10000 έως 10000. Εάν δεν υπάρχει δήλωση, ότι ο κανόνας θεωρείται ότι έχει την προκαθορισμένη τιμή μηδέν.

4.17 Στρατηγικές Επίλυσης Συγκρούσεων

Το CLIPS διαθέτει επτά στρατηγικές επίλυσης συγκρούσεων. Σε κάθε χρονική στιγμή μόνο μια από αυτές είναι ενεργή και χρησιμοποιείται για την επιλογή του κανόνα από την ατζέντα. Η τρέχουσα στρατηγική δίνεται από τη συνάρτηση:

(get-strategy)

η οποία επιστρέφει το όνομα της στρατηγικής.

Ο ορισμός της επιθυμητής στρατηγικής επίλυσης συγκρούσεων γίνεται με τη χρήση της συνάρτησης:

(set-strategy <strategy>)

όπου <strategy> είναι μια από τις τιμές : depth , breadth , simplicity , complexity , lex , mea , random.

Η σημασία των τιμών αυτών είναι η εξής:

- **Depth :** Σύμφωνα με αυτή τη στρατηγική οι νέοι κανόνες μπαίνουν ¨πάνω¨ από τους ¨παλαιούς¨. Η σύγκρουση γίνεται με βάση το πότε εισήχθησαν τα γεγονότα που ικανοποιούν τον κανόνα στη λίστα γεγονότων.
- **Breadth :** Οι κανόνες με απλούστερες συνθήκες κατατάσσονται ¨πάνω¨ από τους κανόνες με τις περισσότερο πολύπλοκες. Η πολυπλοκότητα ενός κανόνα εξαρτάται από τον αριθμό των συνθηκών και από τις συγκρίσεις (περιορισμούς) που λαμβάνουν χώρα στις συνθήκες του κανόνα.
- **Complexity :** Αντίθετα με την προηγούμενη στρατηγική, οι πιο πολύπλοκοι κανόνες μπαίνουν ¨πάνω¨ από τους ¨απλούστερους¨.
- **Lex :** Η στρατιγηκή αυτή πρωτοεμφανίστηκε στο σύστημα OPSS. Οι κανόνες οι οποίοι ενεργοποιούνται από ¨νεότερα¨ γεγονότα κατατάσσονται υψηλότερα στην agenda. Για τους κανόνες οι οποίοι κατατάσσονται στην ¨ίδια ομάδα¨ με βάση το προηγούμενο κριτήριο, υψηλότερα κατατάσσονται οι κανόνες οι οποίοι έχουν

περισσότερες συνθήκες. Ουσιαστικά αποτελεί συνδυασμό των στρατηγών depth και complexity.

- **Mea :** Και αυτή η στρατηγική εμφανίστηκε στο σύστημα OPSS. Εξετάζεται το γεγονός το οποίο αντιστοιχεί στην πρώτη συνθήκη και οι κανόνες διατάσσονται με βάση το πότε εισάχθηκε αυτό στη λίστα. Όσο νεότερο είναι το γεγονός τόσο ¨ψηλότερα¨ μπαίνει ο κανόνας. Για κανόνες οι οποίοι έχουν την ίδια σειρά με βάση αυτό το κριτήριο χρησιμοποιείται η στρατηγική lex.
- **Random :** Οι κανόνες μπαίνουν στην ατζέντα με τυχαίο τρόπο. Υπενθυμίζεται ότι η στρατηγική επίλυσης συγκρούσεων εφαρμόζεται σε κανόνες ίδιας προτεραιότητας

5° ΚΕΦΑΛΑΙΟ

ΕΜΠΕΙΡΑ ΣΥΣΤΗΜΑΤΑ

5.1 Ορισμός Δομή και Λειτουργία ΕΣ

Τα έμπειρα συστήματα είναι υπολογιστικά συστήματα που εμφανίστηκαν σαν πρακτική εφαρμογή της έρευνας και των πορισμάτων της ΤΝ στα μέσα της δεκαετίας του '70. Λειτουργούν σαν προσομοιωτές της ανθρώπινης σκέψης για τη λύση προβλημάτων, οι οποίοι προσφέρουν ορισμένα πλεονεκτήματα έναντι του ανθρώπινου-εμπειρογνώμονα. Τα ΕΣ χρησιμοποιούνται κυρίως σαν σύμβουλοι των ειδικών/εμπειρογνωμόνων για τη λήψη αποφάσεων. Ένα έμπειρο σύστημα είναι ένα πρόγραμμα Η/Υ που μιμείται, σε κάποιο βαθμό, τις διαδικασίες λήψης αποφάσεων ενός ανθρώπινου/εμπειρογνώμονα. Αυτό που επιτυγχάνει στηριζόμενο σε εκτεταμένη γνώση γύρω από ένα στενό πεδίο προβλημάτων, που το σύστημα καλείται να αντιμετωπίσει.

Τα βασικά φυσικά χαρακτηριστικά που ξεχωρίζουν ένα ΕΣ από ένα συμβατικό πρόγραμμα είναι:

α) μπορεί να χειρίζεται όχι μόνο αριθμούς, αλλά και σύμβολα

β) υπάρχει διαχωρισμός μεταξύ γνώσης και χρήσης της γνώσης

γ) χρήση ευριστικής γνώσης σχετικής με το πεδίο της εφαρμογής

Η ακόλουθη εξίσωση συχνά χρησιμοποιείται για την περιγραφή ενός ΕΣ:

Έμπειρο Σύστημα = Γνώση + Συλλογισμός

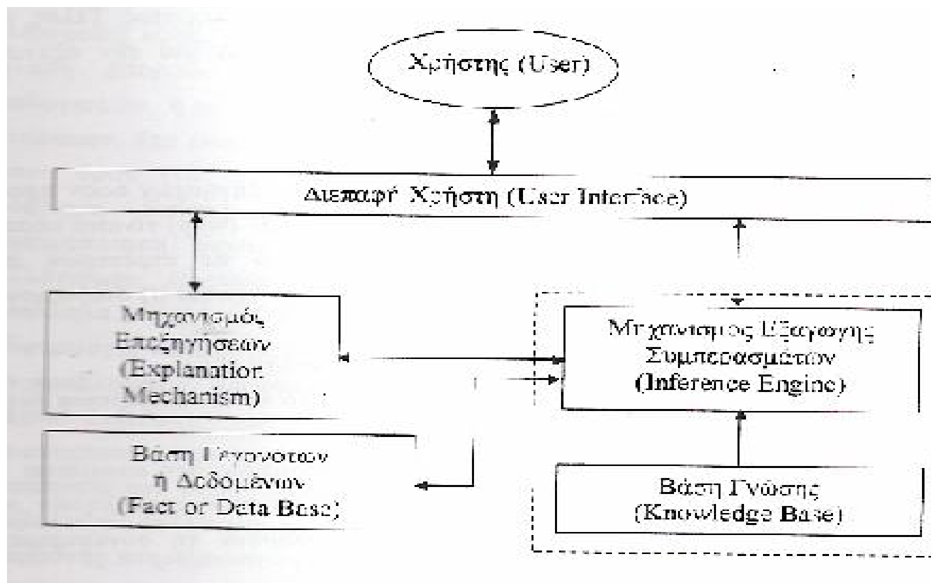
Έτσι ο πυρήνας ενός ΕΣ αποτελείται από δύο βασικές συνιστώσες, τη βάση γνώσης και τον μηχανισμό ή μηχανή εξαγωγής συμπερασμάτων. Πρακτικά όμως, ένα ΕΣ συνήθως περιλαμβάνει επιπλέον μια συνιστώσα επικοινωνίας χρήστη, μια βάση εργασίας και, ίσως, ένα μηχανισμό επεξηγήσεων. Η δομή ενός πλήρους ΕΣ απεικονίζεται στο σχήμα 5.1, όπου ο βασικός πυρήνας σημειώνεται μέσα σε ορθογώνιο με διακεκομμένη γραμμογράφηση.

5.2 Βάση Γνώσης

Περιέχει τη γνώση γύρω από το πεδίο εφαρμογής για το οποίο έχει σχεδιαστεί το ΕΣ, και η οποία συνήθως αποκτάται από τη σχετική βιβλιογραφία και από εμπειρογνώμονες στο αντικείμενο της εφαρμογής. Η γνώση αυτή κωδικοποιείται σαν γεγονότα και κανόνες, μέσω κάποιας γλώσσας αναπαράστασης της γνώσης (ΓΑΓ), και αποτελεί τη μόνιμη γνώση του συστήματος. Η πιο γνωστή ΓΑΓ στα ΕΣ είναι οι κανόνες παραγωγής. Γι' αυτό και συνήθως ταυτίζονται οι όροι ''προσέγγιση/τεχνολογία εμπείρων συστημάτων'' και ''προσέγγιση/τεχνολογία βασισμένη σε κανόνες''. Έτσι Η ΒΓ αποτελείται συνήθως από κανόνες και καλείται βάση κανόνων. Όμως, σε κάποιες περιπτώσεις, χρησιμοποιούνται και άλλες ΓΑΓ, όπως τα πλαίσια, τα σημαντικά δίκτυα και η λογική πρώτης τάξεως. Αυτές οι ΓΑΓ χρησιμοποιούνται συνήθως μαζί με τους κανόνες παραγωγής, σαν δεύτερο συστατικό υβριδικής ΓΑΓ. Είναι γεγονός ότι τα τελευταία χρόνια άρχισαν να χρησιμοποιούνται υβριδικές ΓΑΓ, δηλαδή γλώσσες που συνδυάζουν στοιχεία από δύο ή περισσότερες από τις αναφερθείσες βασικές ΓΑΓ. Μια συνήθης τέτοια περίπτωση είναι ο συνδυασμός πλαισίων και κανόνων παραγωγής.

5.3 Βάση Εργασίας

Η ΒΕ περιέχει κάθε φορά γνώση σχετική με γεγονότα που αφορούν το υπό εξέταση συγκεκριμένο πρόβλημα/ερώτημα. Αυτή η γνώση αλλάζει όταν αλλά το υπό εξέταση πρόβλημα, σε αντίθεση με τη γνώση στη ΒΓ, που παραμένει αμετάβλητη. Στη ΒΕ δηλαδή καταχωρούνται πληροφορίες για την αρχική κατάσταση του προβλήματος, απαντήσεις του χρήστη σε ερωτήσεις του ΕΣ, ενδιάμεσα συμπεράσματα και τελικά συμπεράσματα. Πολλές φορές η ΒΕ δεν αναφέρεται σαν ξεχωριστή μονάδα αλλά θεωρείται ενσωματωμένη στη ΒΓ.



Σχήμα 5.1. Δομή έμπειρου Συστήματος

5.4 Μηχανισμός Εξαγωγής Συμπερασμάτων (ΜΕΣ)

Ο ΜΕΣ εξάγει συμπεράσματα χρησιμοποιώντας τα δεδομένα της ΒΔ και τη γνώση της ΒΓ. Καθορίζει το ''πως'' θα χρησιμοποιηθεί η γνώση που βρίσκεται στη ΒΓ (και στη ΒΕ) για τη λύση του εκάστοτε προβλήματος. Για κάθε συγκεκριμένο πρόβλημα καθορίζει ποια δεδομένα θα χρησιμοποιηθούν σε ποια δεδομένη στιγμή της διαδικασίας, ποιοι κανόνες θα ενεργοποιηθούν και με ποια σειρά, για ποια δεδομένα θα ερωτηθεί ο χρήστης, κλπ.

Ο ΜΕΣ εργάζεται ανεξάρτητα από τη ΒΓ. Δηλαδή, υπάρχει διαχωρισμός της γνώσης (ΒΓ) από τον τρόπο χρήσης της (ΜΕΣ). Αυτό είναι ένα από τα κύρια χαρακτηριστικά των ΕΣ, όπως προαναφέραμε. Λόγω αυτού, ο ίδιος ο ΜΕΣ μπορεί να χρησιμοποιηθεί για περισσότερες από μια ΒΓ. Αυτό αποτελεί και τη βάση στην οποία στηρίζονται τα κελύφη ΕΣ.

5.5 Συνιστώσα Επικοινωνίας Χρήστη

Είναι το μέσον δια του οποίου ο χρήστης επικοινωνεί με τις διάφορες μονάδες του ΕΣ. Μια βασική μορφή επικοινωνίας είναι η δυνατότητα διερεύνησης ή/και μεταβολής του περιεχομένου της ΒΓ. Μια άλλη συνήθης μορφή επικοινωνίας είναι αυτή που γίνεται μέσω ερωτήσεων-απαντήσεων κατά τη διάρκεια μιας διαδικασίας εξαγωγής κάποιου συμπεράσματος, για τη συλλογή δεδομένων προς λύση του υπό εξέταση προβλήματος. Τέλος, μια άλλη μορφή είναι η δυνατότητα παροχής επεξηγήσεων για την εξαγωγή κάποιου συμπεράσματος.

5.6 Συνιστώσα Επεξηγήσεων (ΣΕΠ)

Η συνιστώσα αυτή είναι υπεύθυνη για την παροχή επεξηγήσεων όσον αφορά το ‘’πως’’ εξήχθη κάποιο συμπέρασμα ή το ‘’γιατί’’ γίνεται κάποια ερώτηση ή και το ποιο θα ήταν το συμπέρασμα σε περίπτωση που διαφορετικά δεδομένα είχαν δοθεί, γνωστή σαν ‘’τι εάν’’ διαδικασία.

Τα βασικά λειτουργικά χαρακτηριστικά ενός ΕΣ είναι τα εξής:

* *Εξαγωγή συμπερασμάτων, χωρίς να είναι απαραίτητη όλη η διαθέσιμη πληροφορία-γνώση*. Δηλαδή για την εξαγωγή κάποιου συμπεράσματος δεν απαιτείται να έχουμε γνώση/πληροφορία για τις τιμές όλων των μεταβλητών του συστήματος, αλλά μόνο γι’ αυτές που αφορούν τη συγκεκριμένη εξαγωγή συμπεράσματος.

* *Διαδραστική καθοδήγηση της εισόδου των δεδομένων στο σύστημα.* Αυτό συνδέεται με το προηγούμενο, καθώς το σύστημα ζητά από τον χρήστη κατά τη διάρκεια της λειτουργία του, δηλαδή της εξαγωγής συμπερασμάτων, την απαιτούμενη γνώση/πληροφορία.

**Επεξήγηση των συμπερασμάτων.* Ένα ΕΣ πρέπει να είναι ικανό να δίνει εξηγήσεις για τα συμπεράσματα που εξάγει με απλό και κατανοητό τρόπο.

**Τμηματικότητα.* Η τμηματικότητα αναφέρεται σε δύο επίπεδα. Πρώτον, η βάση γνώσης έχει τμηματικότητα αναπαράστασης, δηλαδή η γνώση αποτελείται από ανεξάρτητες μονάδες γνώσης (π.χ. κανόνες) που συνδέονται μεν μεταξύ τους

εννοιολογικά, αλλά όχι συντακτικά. Αυτό δημιουργεί μια ευελιξία στην αναπαράσταση γνώσης και στη δημιουργία της ΒΓ. Δεύτερον, υπάρχει τμηματικότητα στη δομή ενός ΕΣ. Αυτό συνδέεται με τον διαχωρισμό της γνώσης από τη χρήση της, που αναφέραμε πιο πάνω.

5.7 Κριτήρια Καταλληλότητας

Τα ΕΣ δεν είναι κατάλληλα για όλες τις εφαρμογές. Εφαρμογές στις οποίες οι διαδικασίες λύσης του προβλήματος είναι από τη φύση τους αλγοριθμικές, δηλαδή μπορούν να εκφραστούν σαν μια ακολουθία αριθμητικών υπολογισμών, ή οι λύσεις μπορούν να δοθούν μέσω επίλυσης μαθηματικών εξισώσεων, δεν είναι κατάλληλες για χρησιμοποίηση ΕΣ. Π.χ. η εύρεση του μέσου όρου ενός συνόλου τιμών ενός μεγέθους είναι ένα αλγοριθμικό πρόβλημα. Επίσης, προβλήματα που απαιτούν κατά μεγάλο μέρος χρησιμοποίηση γενικών γνώσεων ή κοινότυπου συλλογισμού για τη λύση τους δεν είναι κατάλληλα για χρήση ΕΣ.
Εφαρμογές κατάλληλες για χρήση ΕΣ είναι εκείνες στις οποίες για τη λύση των προβλημάτων απαιτείται ανθρώπινη εμπειρία, δηλαδή δεν υπάρχει σαφής θεωρία επίλυσης, και οι διαδικασίες επίλυσης μπορούν καλύτερα να παρασταθούν μέσω του συλλογικού συμβολισμού. Π.χ. η ερμηνεία του μέσου όρου ενός συνόλου τιμών μπορεί να είναι πρόβλημα συμβολικού συλλογισμού. Επίσης, όταν οι λύσεις των προβλημάτων μιας εφαρμογής στηρίζονται σε σχετικά στενό, καλά καθορισμένο πεδίο γνώσεων. Ακόμη, προβλήματα των οποίων η λύση στηρίζεται σε αβέβαια ή ασαφή δεδομένα, είναι κατάλληλα για προσέγγιση μέση ΕΣ.
Το παραπάνω συνιστούν τεχνολογικούς παράγοντες καταλληλότητας της χρήσης ΕΣ για μια εφαρμογή. Όμως, παράλληλα υπάρχουν και ανθρώπινοι παράγοντες. Έτσι, παρ’ όλο που είναι πρόβλημα μπορεί να είναι τεχνολογικά κατάλληλο για λύση μέσω ΕΣ, αν οι εμπειρογνώμονες του είδους είναι φθηνοί ή δεν είναι διαθέσιμοι (δεν προτίθενται) να συνεργαστούν, τότε δεν συνιστάται η κατασκευή ΕΣ. Το ίδιο μπορεί να ισχύσει και στην περίπτωση που σημαντικό έγγραφο υλικό (π.χ. εγχειρίδια, μελέτες παραδειγματικών περιπτώσεων κλπ.) δεν υπάρχει.

Στον παρακάτω πίνακα συνοψίζονται τα βασικά κριτήρια καταλληλότητας μιας εφαρμογής για χρήση ΕΣ.

Μερικά παραδείγματα εφαρμογών κατάλληλων για ΕΣ είναι:

* διάγνωση ασθενειών
* διάγνωση τεχνολογικών συστημάτων
* σχεδίαση ηλεκτρονικών κυκλωμάτων
* εκτίμηση γεωλογικών ευρημάτων
* απόδειξη θεωρημάτων
* ανάλυση χημικών ενώσεων

Είδος Χαρακτηριστικών	**Χαρακτηριστικά Προβλημάτων Κατάλληλων για ΕΣ**	**Χαρακτηριστικά Προβλημάτων Ακατάλληλων για ΕΣ**
Τεχνολογικά Χαρακτηριστικά	*Συμβολική φύση διαδικασίας επίλυσης *Καλά καθορισμένο και σχετικά στενό πεδίο γνώσης *Απαιτείται ανθρώπινη εμπειρία *Απαιτούνται λύσεις από ελλειπή, ασαφή ή αβέβαια δεδομένα	*Αλγοριθμική φύση διαδικασίας επίλυσης *Απαιτούνται γενικές γνώσεις ή κοινότυπος συλλογισμός *Υπάρχει μαθηματική περιγραφή *Απαιτούνται λύσεις από απόλυτα γνωστά δεδομένα
Ανθρώπινοι Παράγοντες	*Εμπειρογνώμονες σπάνιοι και ακριβοί *Εμπειρογνώμονες διαθέσιμοι	*Εμπειρογνώμονες πολλοί και φθηνοί *Εμπειρογνώμονες μη διαθέσιμοι

5.8 Τύποι Έμπειρων Συστημάτων

Τα ΕΣ εξυπηρετούν διαφορετικούς ρόλους σε διαφορετικές εφαρμογές. Έτσι, έχουμε διάφορους τύπους ΕΣ, ανάλογα με το ρόλο που εξυπηρετεί ο καθένας από αυτούς. Οι σπουδαιότεροι τύποι παρουσιάζονται στη συνέχεια εν συντομία.

ΕΣ Βοηθοί

Καλούνται να εκτελέσουν μια συγκεκριμένη εργασία, που είναι μέρος ενός μεγαλυτέρου έργου που εκτελείται από ένα ή περισσότερους ανθρώπους ή άλλα συστήματα

Κριτικά ΕΣ

Επανεξετάζουν ένα έργο που ήδη έχει εκτελεστεί και κάνουν σχόλια για την ακρίβεια, την συνέπεια, την πληρότητα κλπ. Του έργου. ΕΣ αυτού του τύπου είναι από τα πιο σπάνια

ΕΣ Ειδικοί Σύμβουλοι

Προσφέρουν συμβουλές ή γνώμες για κάποιο θέμα, βασισμένα σε πληροφορίες που παρέχει ο χρήστης. Είναι ο πιο διαδεδομένος τύπος ΕΣ.

ΕΣ Δάσκαλοι

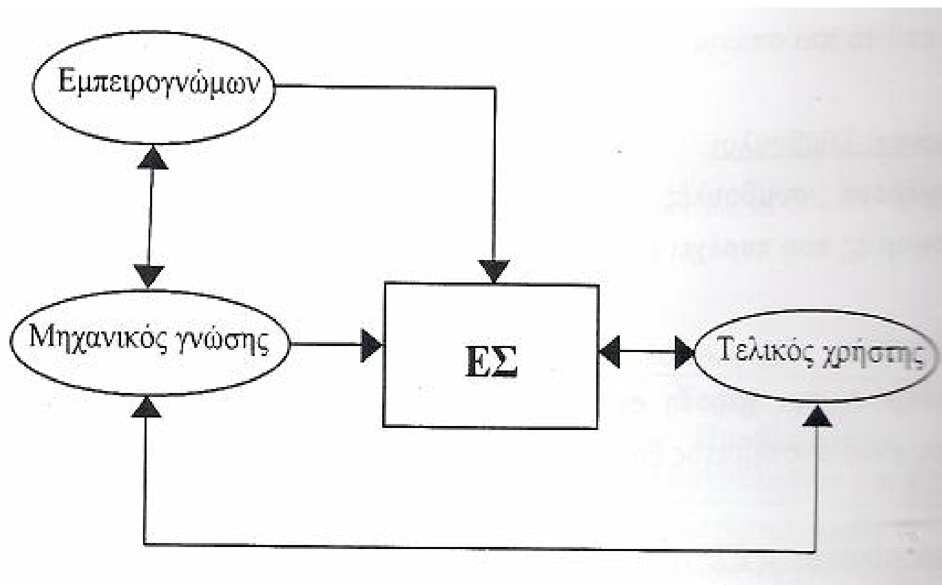
Εκπαιδεύουν τον χρήστη στην εκτέλεση ενός ειδικού έργου, όπως π.χ. ο έλεγχος ενός συστήματος βαλβίδων πίεσης σ’ ένα εργοστάσιο.

Αυτοματοποιημένα ΕΣ

Εκτελούν ένα έργο αυτοματοποιημένα, ανεξάρτητα από τον χρήστη, και αναφέρουν αποτέλεσμα. Είναι ο δεύτερος πιο διαδεδομένος τύπος ΕΣ μετά από τα ΕΣ συμβούλους.

5.9 Μηχανολογία της Γνώσης

Στην ανάπτυξη/κατασκευή ενός ΕΣ εμπλέκονται τρεις κατηγορίες ανθρώπων, ο μηχανικός γνώσης, ο εμπειρογνώμων και ο τελικός χρήστης. Ο μηχανικός γνώσης είναι αυτός που έχει τη διαχείριση της κατασκευής και κατευθύνει τις διάφορες φάσεις της. Είναι αυτός που επιλέγει τα εργαλεία ανάπτυξης (είτε αφορούν το υλικό είτε το λογισμικό), εκμαιεύει την απαραίτητη γνώση από τον εμπειρογνώμονα και την κωδικοποιεί (αναπαριστά) με αποδοτικό τρόπο στη ΒΓ του συστήματος. Ο μηχανικός γνώσης μπορεί να μην έχει καμία απολύτως γνώση του γνωστικού πεδίου της εφαρμογής εκ των προτέρων. Ο εμπειρογνώμων ή ειδικός είναι αυτός που παρέχει τη γνώση για το γνωστικό πεδίο της εφαρμογής. Είναι άτομο που έχει εργασθεί στο πεδίο της εφαρμογής για ικανό χρόνο, ώστε ξέρει καλά τα (πιθανά) προβλήματα και τον τρόπο αντιμετώπισής τους, καθώς και διάφορα τεχνάσματα που αποκτήθηκαν μέσω της εμπειρίας του. Δηλαδή έχει όλα εκείνα τα στοιχεία που τον χαρακτηρίζουν σαν ειδικό. Τέλος, ο τελικός χρήστης είναι εκείνος που θέτει περιορισμούς από πλευράς χρήστη κατά την ανάπτυξη του συστήματος, ιδιαίτερα όσον αφορά την συνιστώσα επικοινωνίας χρήστη.



Σχήμα 5.2 Κατηγορίες εμπλεκομένων ανθρώπων στην κατασκευή ΕΣ

Στο σχήμα 5.2 απεικονίζεται η αλληλεπίδραση των τριών κατηγοριών ανθρώπων στην κατασκευή ενός ΕΣ.

Σχήμα 5.3 Στάδια μηχανολογίας της γνώσης

Η διαδικασία απόκτησης γνώσης από τον εμπειρογνώμονα και η μεταφορά της σε μορφή εκτελέσιμη από τον Η/Υ είναι γνωστή σαν μηχανολογία της γνώσης. Η διαδικασία της ΜΓ. Περιλαμβάνει τρία στάδια: απόκτηση γνώσης, αναπαράσταση γνώσης και μηχανιστική υλοποίηση. Τα περιεχόμενα των τριών αυτών σταδίων απεικονίζονται στη σχήμα 1.3.
Όσον αφορά στην αναπαράσταση της γνώσης ασχοληθήκαμε στα προηγούμενα κεφάλαια. Στη συνέχεια θα ασχοληθούμε με την απόκτηση γνώσης. Η υλοποίηση στον Η/Υ δεν είναι από τα θέματα αυτών των σημειώσεων.

5.10 Απόκτηση Γνώσης

Η απόκτηση γνώσης αποτελεί το δυσκολότερο στάδιο ΜΓ. Η απόκτηση γνώσης γίνεται από διάφορες πηγές γνώσης, όπως εμπειρογνώμονες, βιβλιογραφία και βάσεις δεδομένων. Η δυσκολία έγκειται κυρίως στην απόκτηση γνώσης από εμπειρογνώμονες, γνωστή στην εκμαίευση της γνώσης, διότι υπάρχει μια εγγενής αδυναμία των εμπειρογνωμόνων στο να εκφράσουν τη γνώση που κατέχουν. Γι' αυτό και η απόκτηση γνώσης έχει χ χαρακτηριστεί σαν το ταμείο συμφόρησης στην εν γένει ανάπτυξη ενός ΕΣ.

Υπάρχουν αρκετές τεχνικές για την απόκτηση γνώσης από ένα ειδικό, μερικές δανεισμένες από την ανάλυσης των συμβατικών συστημάτων επεξεργασίας δεδομένων. Σκοπός των μεθόδων αυτών είναι να γίνουν φανερά τα δεδομένα, οι κανόνες και οι διαδικασίες συλλογισμού (λύσης) που χρησιμοποιεί ο ειδικός για τη λύση προβλημάτων σχετικών με την υπ' όψιν εφαρμογή. Οι τεχνικές αυτές περιγράφονται εν συντομία στη συνέχεια.

Δομημένες συνεντεύξεις

Στην πιο απλή μορφή τους, σ' ένα προκαταρκτικό στάδιο, οι συνεντεύξεις είναι μια σειρά από αδόμητες συνομιλίες με τον ειδικό, ώστε να σκιαγραφηθεί το πρόβλημα. Στα επόμενα στάδια απόκτησης γνώσης μπορεί να είναι ημιδομημένες συζητήσεις ή συζητήσεις εντοπισμένες σε ορισμένα θέματα, χωρίς όμως συγκεκριμένες ερωτήσεις. Καλό είναι οι συνεντεύξεις να μαγνητοφωνούνται, με την άδεια του ειδικού.

Ανάλυση πρωτοκόλλων

Στην τεχνική αυτή, η διαδικασία δεν καθοδηγείται από τον μηχανικό γνώσης, όπως στην προηγούμενη, αλλά από τον ειδικό. Ο ειδικός υποχρεούται να σκέφτεται ''φωναχτά'', ενώ επεξεργάζεται κάποιες περιπτώσεις προβλημάτων, και προτίμηση πραγματικές. Η μαγνητοφώνηση συνίσταται και εδώ. Η ανάλυση πρωτοκόλλου είναι πολύ χρήσιμη για την εξαγωγή της γενικής δομής της γνώσης που χρησιμοποιείται ο ειδικός και του τρόπου με τον οποίο την εφαρμόζει. Πιθανόν να χρειάζεται να συμπληρωθεί και από άλλη τεχνική για την εξαγωγή λεπτομερειών της γνώσης.

Παρατήρηση

Στην τεχνική αυτή ο μηχανικός γνώσης είναι απλός παρατηρητής του τρόπου με τον οποίο ο ειδικός επιτελεί κάποια νοητική εργασία που πρέπει να αποτυπωθεί στη ΒΓ. Συνήθως ο μηχανικός γνώσης δεν διακόπτει καθόλου, αλλά αυτό

εξαρτάται από τη φύση της εφαρμογής. Εδώ συνίσταται η χρήση μαγνητοφώνησης ή και βιντεοσκόπησης.

Αυτοεξέταση

Συχνά αποκαλείται τεχνική τελευταίας καταφυγής. Στην τεχνική αυτή, κατά τη διάρκεια επεξεργασίας μιας περίπτωσης, ο μηχανικός διακόπτει τον ειδικό για να κάνει ερωτήσεις του τύπου: ‘’πες μου τι σκέπτεσαι τώρα’’ ή ‘’πώς θα μπορούσα να κάνω αυτό’’, όπου ο μηχανικός διακόπτει μόνο για να θυμίσει στον εμπειρογνώμονα να συνεχίσει ομιλών.

5.11 Μεθοδολογία Ανάπτυξης ΕΣ

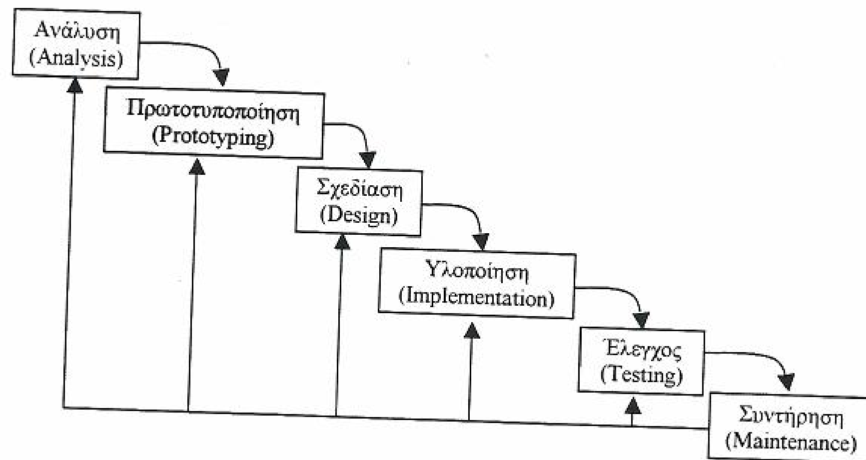
Ενώ για την ανάπτυξη ενός συμβατικού συστήματος επεξεργασίας δεδομένων μπορεί να καθοριστεί μια συγκεκριμένη μεθοδολογία ανάπτυξης, σαν ένα σύνολο διαδοχικών διακεκριμένων μεταξύ τους φάσεων, γνωστό σαν κύκλος ζωής του συστήματος, δεν είναι εύκολο να γίνει το ίδιο με ένα ΕΣ. Οι βασικοί λόγοι γι’ αυτό είναι οι εξής:

*ο μη αλγοριθμικός χαρακτήρας της λύσης των προβλημάτων

*η μη ντετερμινιστική συμπεριφορά της διαδικασίας συλλογισμού

*η μη ύπαρξη ακριβούς σχέσεως εισόδου-εξόδου

Για τους λόγους αυτούς, η ανάπτυξη ενός ΕΣ βασίζεται κυρίως στην ανάπτυξη ενός προτύπου συστήματος και την επαναληπτική του βελτίωση. Μπορεί όμως, παρ’ όλα αυτά να δοθεί μια κατά προσέγγιση μεθοδολογία ανάπτυξης ενός ΕΣ κατ’ αντιστοιχία με αυτή ενός συμβατικού συστήματος, όπως φαίνεται στο σχήμα 5.4. Όπως είναι φανερό, σε σχέση με τη συμβατική μεθοδολογία υπάρχει μεγαλύτερη αλληλοκάλυψη των φάσεων και εντονότερα επαναληπτική προσέγγιση.



Σχήμα 5.4 Κύκλος ζωής έμπειρων συστημάτων

Στη συνέχεια παρουσιάζουμε εν συντομία τις διαδικασίες που συμπεριλαμβάνει κάθε φάση.

Ανάλυση

Η φάση της ανάλυσης περιλαμβάνει δύο υποφάσεις, την ανάλυση προβλήματος και τον προσδιορισμό απαιτήσεων, που περιλαμβάνουν τις παρακάτω διαδικασίες η κάθε μια.

Ανάλυση προβλήματος

- Εκτίμηση εφαρμοσιμότητας ΕΣ
- Εκτίμηση διαθεσιμότητας πόρων
- Εκτίμηση κόστους-ωφέλειας

* *Προσδιορισμός απαιτήσεων*

- Προσδιορισμός στόχων και μέσων επίτευξης
- Προσδιορισμός απαιτήσεων χρήστη (είσοδοι-έξοδοι)
- Προσδιορισμός απαιτήσεων συστήματος (περιορισμοί)

Προτυποποίηση

Επίσης, η φάση της προτυποποίησης περιλαμβάνει κι αυτή δύο υποφάσεις, την προκαταρκτική σχεδίαση και την δημιουργία και αξιολόγηση προτύπου. Στη συνέχεια αναλύονται επιγραμματικά οι δύο αυτές υποφάσεις.

*_Προκαταρκτική Σχεδίαση_

- Σχεδίαση αρχιτεκτονικής πρωτοτύπου

- Μικρή κλίμακας απόκτησης γνώσης

- Επιλογή γλώσσας Αναπαράστασης

- Επιλογή εργαλείου ανάπτυξης

* _Δημιουργία και Αξιολόγηση Πρωτοτύπου_

- Αναπαράσταση γνώσης

- Υλοποίηση στον Η/Υ

- Εγκυροποίηση πρωτοτύπου

ΚΕΦΑΛΑΙΟ 6^{Ο}

ΕΠΙΛΟΓΟΣ

Στην Πτυχιακή Εργασία αυτή προσπαθήσαμε να αναπτύξουμε και να παρουσιάσουμε τις γλώσσες προγραμματισμού οι οποίες εντάσσονται σε μια ευρύτερη «οικογένεια» τον Ευφυή Προγραμματισμό. Η συνεχής αναζήτηση των επιστημών για νέες μεθόδους και αναπαράστασης-αποκωδικοποίησης των ανθρωπίνων συμπεριφορών σε εφαρμογές Η/Υ καθιστά την αναζήτηση-έρευνα καθημερινή και επίμονη. Μέσα από την δημιουργία της Lisp από το τέλος της δεκαετίας του ’50 και η διαφοροποίηση αυτής με το πέρασμα των χρόνων καθώς επίσης και ο ανταγωνισμός μέσα από τη δημιουργία άλλων Γλωσσών Προγραμματισμού, Clips, διαφαίνεται η ανάγκη αναζήτησης. Τεχνητή Νοημοσύνη και Έμπειρα Συστήματα συμπληρώνουν το Puzzle της Πτυχιακής Εργασίας.

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Ι. Χατζηλυγερούδης και Κ. Κουτσογιάννης "Ευφυής Προγραμματισμός", Πανεπιστημιακές Παραδόσεις, 2006 (Σημειώσεις - Διαφάνειες)

2. Τεχνητή νοημοσύνη : Σχεδιάζοντας τη νόηση : Από την υπολογιστική θεωρία στις σύγχρονες ευφυείς μηχανές / John Haugeland. Μετάφραση Στέλιος Ζαχαρίου (1992)

3. Τεχνητή νοημοσύνη & έμπειρα συστήματα / Αθανάσιος Τσακαλίδης, Ιωάννης Χατζηλυγερούδης (1998)

4. Έμπειρα συστήματα - τεχνητή νοημοσύνη και LISP / Γεώργιος Ι. Δουκίδης, Μάριος Κ. Αγγελίδης (1998)

WEBSITES

1. http://el.wikipedia.org/wiki/Lisp

2. http://www.focusmag.gr/articles/view-article.rx?oid=141788

3.http://el.wikipedia.org/wiki/%CE%A4%CE%B5%CF%87%CE%BD%CE%B7%CF%84%CE%AE_%CE%BD%CE%BF%CE%B7%CE%BC%CE%BF%CF%83%CF%8D%CE%BD%CE%B7

4. http://pileas.csd.auth.gr/login/index.php

5. http://mmlab.ceid.upatras.gr/aigroup/undergrad/aiprog/