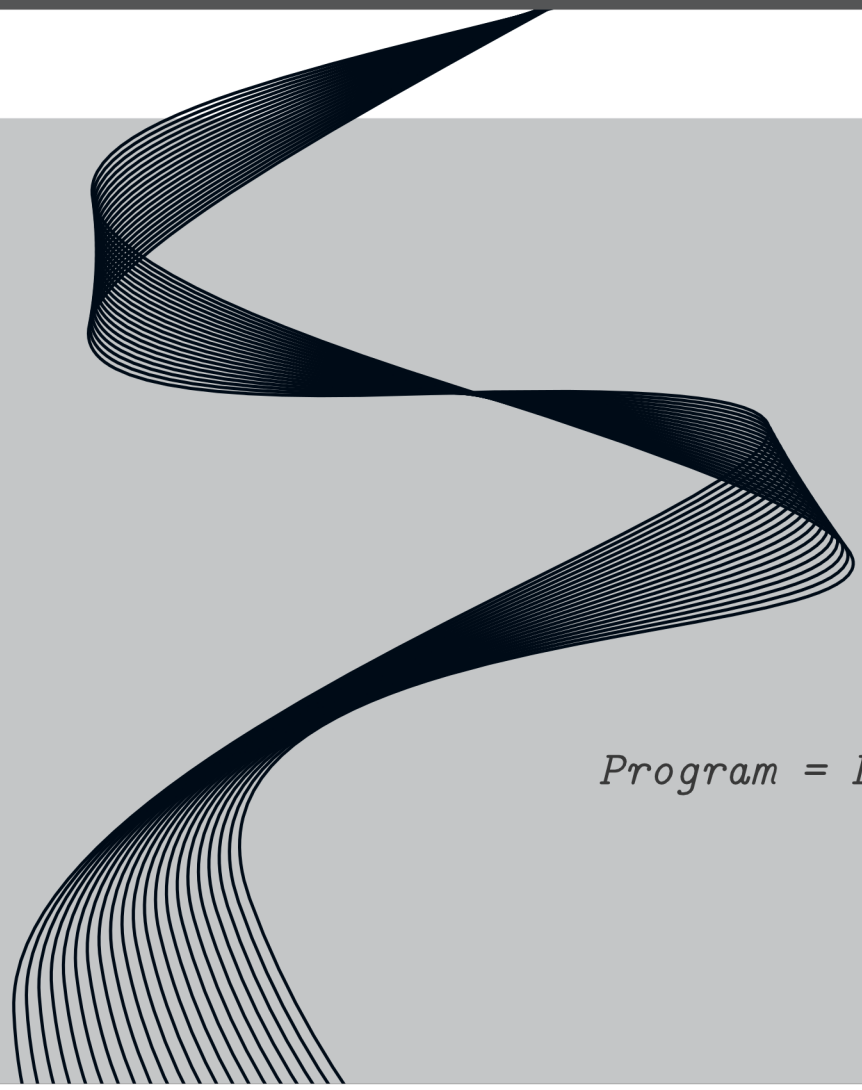
Ηλίας Σακελλαρίου Νικόλαος Βασιλειάδης
Πέτρος Κεφαλάς Δημοσθένης Σταμάτης

ΤΕΧΝΙΚΕΣ ΛΟΓΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Η γλώσσα Prolog



Program = Logic + Control
(R.Kowalski)

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά Συγγράμματα και Βοηθήματα
www.kallipos.gr

HEALLINK
Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

ΕΣΠΑ 2007-2013
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

ΗΛΙΑΣ ΣΑΚΕΛΛΑΡΙΟΥ
Λέκτορας, Πανεπιστήμιο Μακεδονίας

ΝΙΚΟΛΑΟΣ ΒΑΣΙΛΕΙΑΔΗΣ
Αναπληρωτής Καθηγητής, Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

ΠΕΤΡΟΣ ΚΕΦΑΛΑΣ
Honorary Professor, University Of Sheffield

ΔΗΜΟΣΘΕΝΗΣ ΣΤΑΜΑΤΗΣ
Καθηγητής, Αλεξάνδρειο ΤΕΙ Θεσσαλονίκης

Τεχνικές Λογικού Προγραμματισμού

Η γλώσσα Prolog

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά
Συγγράμματα και Βοηθήματα
www.kallipos.gr

Τεχνικές Λογικού Προγραμματισμού

Συγγραφή

Ηλίας Σακελλαρίου

Νικόλαος Βασιλειάδης

Πέτρος Κεφαλάς

Δημοσθένης Σταμάτης

Κριτικός αναγνώστης

Αθανάσιος Τσαδήρας

Συντελεστές έκδοσης

Γλωσσική Επιμέλεια: Η συγγραφική Ομάδα

Γραφιστική Επιμέλεια: Η συγγραφική Ομάδα

Τεχνική Επεξεργασία: Η συγγραφική Ομάδα

ISBN: 978-960-603-246-2

ΣΥΝΔΕΣΜΟΣ ΕΛΛΗΝΙΚΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΒΙΒΛΙΟΘΗΚΩΝ
Εθνικό Μετσόβιο Πολυτεχνείο
Ηρώων Πολυτεχνείου 9, 15780 Ζωγράφου
www.kallipos.gr

Στην Κατερίνα, στην Ελένη και στον Χριστόφορο
Η.Σακελλαρίου

Στην Δήμητρα, στην Άννα και στην Ευδοκία
Ν. Βασιλειάδης

Στη Σόφη, στην Αφροδίτη και στο Γρηγόρη.
Π.Κεφαλάς

Στη Δήμητρα, στη Βέρα και στη Βάσω
Δ. Σταμάτης

Πίνακας περιεχομένων

Πίνακας συντομεύσεων-ακρωνύμια

ΛΠ	Λογικός Προγραμματισμός
ALP	Association for Logic Programming
ASCII	American Standard Code for Information Interchange
CLP	Constraint Logic Programming
DCG	Definite Clause Grammar
LP	Logic Programming
OWL	Web Ontology Language
RDF	Resource Description Framework
SWRL	Semantic Web Rule Language

Πρόλογος

Ο Λογικός Προγραμματισμός ανήκει στις πλέον ενδιαφέρουσες σχολές προγραμματισμού, σημαντικά διαφορετικές από τις "κλασικές" σχολές του προστακτικού - αντικειμενοστραφούς προγραμματισμού. Η χρήση της Μαθηματικής Λογικής ως εργαλείου αφαίρεσης της περιγραφής υπολογισμών και η εκμετάλλευση των αποδεικτικών διαδικασιών της πρώτης οδηγεί σε συμπαγή προγράμματα τα οποία βρίσκουν εφαρμογές σε πολύπλοκα και ενδιαφέροντα πεδία, όπως είναι η Τεχνητή Νοημοσύνη και ο Σημασιολογικός Ιστός. Όμως, αυτή η υψηλού επιπέδου προσέγγιση στον προγραμματισμό δημιουργεί προβλήματα σε εκείνους που για πρώτη φορά έρχονται σε επαφή με τον Λογικό Προγραμματισμό, αφενός γιατί απαιτεί να υιοθετήσουν μια δηλωτική προσέγγιση στην ανάπτυξη προγραμμάτων "ξένη" προς αυτή των κυρίαρχων γλωσσών, και αφετέρου, γιατί απαιτείται η εκμάθηση τεχνικών αναπαράστασης και επίλυσης προβλημάτων που αν και είναι γενικά εφαρμόσιμες στον προγραμματισμό, δεν χρησιμοποιούνται συχνά στις υπόλοιπες σχολές, με κλασικό παράδειγμα την αναδρομή.

Το παρόν βιβλίο φιλοδοξεί να καλύψει τις παραπάνω δύο απαιτήσεις και να αποτελέσει το βασικό σύγγραμμα για οποιονδήποτε επιθυμεί να εντρυφήσει στην τέχνη και τεχνικές του Λογικού Προγραμματισμού. Έχοντας ως όχημα τον κυριότερο εκπρόσωπο της σχολής Λογικού Προγραμματισμού, την γλώσσα Prolog, το βιβλίο έχει ως στόχο:

- να αναφέρει σύντομα τα θεωρητικά θεμέλια του Λογικού Προγραμματισμού, δηλαδή της Κατηγορηματικής Λογικής Πρώτης Τάξης και της αρχής της ανάλυσης,
- να παρουσιάσει σε βάθος την γλώσσα προγραμματισμού Prolog, τα διαθέσιμα κατηγορήματα και πώς αυτά εντασσόμενα σε τεχνικές Λογικού Προγραμματισμού αποτελούν ισχυρά εργαλεία για την επίλυση προβλημάτων,
- να παρουσιάσει εφαρμογές στις οποίες ο Λογικός Προγραμματισμός προσφέρει σημαντικά πλεονεκτήματα, και τέλος,
- να αναπτύξει προγραμματιστικές δεξιότητες του αναγνώστη, όπως είναι η αναδρομή, εφαρμόσιμες σε όλες τις σχολές προγραμματισμού.

Το βιβλίο προέκυψε από την ανάγκη των συγγραφέων να συγκεντρώσουν με συστηματικό τρόπο όλο το υλικό που έχουν δημιουργήσει είτε ατομικά είτε σε συνεργασία μεταξύ τους και στο οποίο έχει ενσωματωθεί η πολύχρονη διδακτική τους εμπειρία, στον τομέα του Λογικού Προγραμματισμού και της γλώσσας Prolog.

Ο Alan Perlis στο άρθρο του “Epigrams on Programming” έγραφε: “*Μία γλώσσα που δεν επηρεάζει τον τρόπο που σκέφτεται κανείς για τον προγραμματισμό δεν αξίζει να τη γνωρίσει*”. Οι συγγραφείς αυτού του βιβλίου πιστεύουν ακράδαντα ότι οι αναγνώστες μελετώντας τα κεφάλαια του και δοκιμάζοντας τα πολλά προγραμματιστικά παραδείγματα που παρουσιάζονται θα κατακτήσουν έναν εναλλακτικό δημιουργικό τρόπο σκέψης για την ανάπτυξη εφαρμογών λογισμικού. Θα είναι γνώστες νέων τεχνικών και μεθοδολογιών και θα έχουν τη διάθεση να τις εφαρμόσουν για να αντιμετωπίσουν τις σύγχρονες προκλήσεις που θέτει σήμερα η επιστήμη των υπολογιστών και τα σύνθετα πεδία εφαρμογής της.

Ο κώδικας που αντιστοιχεί στα παραδείγματα των κεφαλαίων του βιβλίου μπορεί να ανακτηθεί από το σύνδεσμο http://users.uom.gr/~iliass/LPTechniques, ο οποίος το συνοδεύει και θα ανανεώνεται συνεχώς.

Η Συγγραφική Ομάδα

Οκτώβριος 2015

Οι Συγγραφείς

Ο **Ηλίας Σακελλαρίου** είναι λέκτορας στο Τμήμα Εφαρμοσμένης Πληροφορικής στο Πανεπιστήμιο Μακεδονίας. Έχει πτυχίο Φυσικής από το ΑΠΘ (1995), μεταπτυχιακό τίτλο σπουδών από το τμήμα Τεχνητής Νοημοσύνης του Πανεπιστημίου του Εδιμβούργου (1996) και διδακτορικό από το Τμήμα Πληροφορικής του ΑΠΘ (2006), με θέμα "Παράλληλα Συστήματα Λογικού Προγραμματισμού με Περιορισμούς". Έχει διδάξει μαθήματα Λογικού Προγραμματισμού, Τεχνητής Νοημοσύνης και Ευφυών Πρακτόρων, τόσο σε προπτυχιακό όσο και σε μεταπτυχιακό επίπεδο, για περισσότερο από 10 χρόνια. Είναι συν-συγγραφέας του βιβλίου Τεχνητή Νοημοσύνη, Εκδόσεις Παν. Μακεδονίας και του βιβλίου Parallel and Constraint Logic Programming (Kluwer). Τα ερευνητικά του ενδιαφέροντα περιλαμβάνουν, λογικό προγραμματισμό με περιορισμούς, μοντελοποίηση μέσω περιορισμών, συστήματα πρακτόρων με έμφαση στις γλώσσες προγραμματισμού μοντελοποίησης και προσόμοίωσής τους, και σχεδιασμό ενεργειών. Είναι μέλος της Ελληνικής Εταιρείας Τεχνητής Νοημοσύνης (ΕΕΤΝ), της IEEE και της ACM.

Home Page: http://users.uom.gr/~iliass

Ο **Νικόλαος Βασιλειάδης** αποφοίτησε από το ΑΠΘ με πτυχίο Φυσικής το 1991, έλαβε Μάστερ στην Τεχνητή Νοημοσύνη (MSc Applied Artificial Intelligence) από το Πανεπιστήμιο του Aberdeen, Σκοτία, το 1992, και Διδακτορικό από το Τμήμα Πληροφορικής του ΑΠΘ το 1998, με αντικείμενο τις Παράλληλες Βάσεις Γνώσης. Διατελεί μέλος ΔΕΠ στο Τμήμα Πληροφορικής ΑΠΘ από το 2003, αρχικά ως Επίκουρος Καθηγητής, και από το 2010 ως Αναπληρωτής Καθηγητής. Παράλληλα, από το 2010 διατελεί Ακαδημαϊκός Συντονιστής της Σχολής Επιστημών- Τεχνολογίας του Διεθνούς Πανεπιστημίου Ελλάδος. Έχει διδάξει μαθήματα σχετιζόμενα με τα Συστήματα Γνώσης, τον Λογικό Προγραμματισμό, τη Διαχείριση Γνώσης, τον Σημασιολογικό Ιστό, τα Ευφυή Συστήματα και τους Ευφυείς Πράκτορες, σε προπτυχιακό και μεταπτυχιακό επίπεδο, σε πέντε διαφορετικά ιδρύματα. Στα επιστημονικά του ενδιαφέροντα συμπεριλαμβάνονται τα συστήματα γνώσης και τα συστήματα κανόνων, τα πολυπρακτορικά συστήματα, ο Σημασιολογικός Ιστός, οι Οντολογίες και τα ανοιχτά συνδεδεμένα δεδομένα. Έχει δημοσιεύσει πάνω από 150 επιστημονικά άρθρα σε διεθνή περιοδικά, συνέδρια και κεφάλαια σε βιβλία και έχει συγγράψει 2 βιβλία. Το δημοσιευμένο του έργο έχει λάβει περισσότερες από 880 αναφορές (h-index 16). Συμμετείχε στην επιστημονική επιτροπή πάνω από 70 διεθνών συνεδρίων και στην οργανωτική επιτροπή 5 συνεδρίων. Ήταν πρόεδρος της επιτροπής προγράμματος των συνεδρίων RuleML-2008, RuleML-2011@IJCAI και WIMS-2014. Έχει συμμετάσχει σε 29 ερευνητικά και αναπτυξιακά έργα, όντας επιστημονικός υπεύθυνος σε 9 από αυτά. Διετέλεσε γενικός γραμματέας της Ελληνικής Εταιρείας Τεχνητής Νοημοσύνης, αποτελεί μέλος του διευθυντικού συμβουλίου της διεθνούς επιστημονικής οργάνωσης RuleML, Inc., και είναι μέλος της ΕΠΥ, της IEEE και της ACM.

Home Page: http://lpis.csd.auth.gr/people/nbassili/

Ο **Πέτρος Κεφαλάς** είναι επίτιμος Καθηγητής του Πανεπιστημίου του Sheffield, UK. Είναι διευθυντής Μάθησης και Διδασκαλίας (Learning & Teaching) στο Διεθνές Τμήμα του Πανεπιστημίου (The University of Sheffield International Faculty) στη Θεσσαλονίκη. Την περίοδο 1993-2007 διετέλεσε Επιστημονικός Συνεργάτης του Τμήματος Πληροφορικής του ΤΕΙ Θεσσαλονίκης. Αποφοίτησε από το Τμήμα Φυσικής του ΑΠΘ το 1986 και απέκτησε μεταπτυχιακό τίτλο MSc στην Τεχνητή Νοημοσύνη το 1987 από το Πανεπιστήμιο του Essex. Στο ίδιο πανεπιστήμιο ολοκλήρωσε τη διδακτορική του διατριβή το 1991 σε "Παράλληλη Εκτέλεση Λογικών Προγραμμάτων με Εφαρμογές σε Αλγόριθμους Αναζήτησης". Στα ερευνητικά του ενδιαφέροντα περιλαμβάνονται η Τεχνητή Νοημοσύνη, τα Προγράμματα Πράκτορες, ο Παράλληλος Λογικός Προγραμματισμός και οι Τυποποιημένες Μέθοδοι στη Μηχανική Λογισμικού. Είναι ένας από τους συγγραφείς του πιο δημοφιλούς Ελληνικού βιβλίου Τεχ\χητής Νοημοσύνης. Έχει διατελέσει αξιολογητής σε Ευρωπαϊκά προγράμματα, σε πολλά συνέδρια και επιστημονικά περιοδικά. Έχει δημοσιεύσει πάνω από 100 επιστημονικά άρθρα σε διεθνή περιοδικά, συνέδρια και κεφάλαια σε βιβλία. Είναι μέλος των

επιστημονικών και επαγγελματικών ενώσεων και συλλόγων ΕΠΥ, ΕΕΤΝ, BCS, IEEE, ALP και ACM. Διετέλεσε μέλος της διοικούσας επιτροπής της Εταιρείας Επιστημόνων και Επαγγελματιών Πληροφορικής και Τηλεπικοινωνιών (ΕΠΥ) και της Ελληνικής Εταιρείας Τεχνητής Νοημοσύνης (ΕΕΤΝ).

Home Page: http://citycollege.sheffield.eu/frontend/members_profile.php?m=23

Ο **Δημοσθένης Σταμάτης** είναι Καθηγητής του Τμήματος Πληροφορικής του Αλεξάνδρειου Τεχνολογικού Εκπαιδευτικού Ιδρύματος Θεσσαλονίκης (ΑΤΕΙΘ). Έχει Πτυχίο Μαθηματικών από το Πανεπιστήμιο Ιωαννίνων (1981) και Διδακτορικό Δίπλωμα στην Επιστήμη των Υπολογιστών-Τεχνητή Νοημοσύνη, από τον Τομέα Ηλεκτρονικής και Υπολογιστών, του Φυσικού Τμήματος, του Αριστοτελείου Πανεπιστημίου Θεσσαλονίκης (1987). Στα ερευνητικά του ενδιαφέροντα περιλαμβάνονται οι Τεχνικές της Τεχνητής Νοημοσύνης, ο Λογικός Προγραμματισμός, οι Ευφυείς Πράκτορες και η Χρήση των Τεχνολογιών της Πληροφορικής και των Επικοινωνιών στην Εκπαίδευση - Ηλεκτρονική Μάθηση. Έχει μακροχρόνια διδακτική εμπειρία, σε προπτυχιακό και μεταπτυχιακό επίπεδο, στα αντικείμενα: Μεθοδολογίες Προγραμματισμού, Δομές Δεδομένων, Τεχνητή Νοημοσύνη, Λογικός Προγραμματισμός και Ηλεκτρονική Μάθηση. Έχει διδάξει στο Τμήμα Πληροφορικής του ΑΤΕΙΘ, στο ΑΠΘ (ως Επιστημονικός Συνεργάτης και Λέκτορας μέχρι το 1991) και στο Πανεπιστήμιο Μακεδονίας. Εχει διατελέσει Προϊστάμενος του Τμ. Πληροφορικής, Διευθυντής της Σχολής Τεχνολογικών Εφαρμογών και Αντιπρόεδρος της Επιτροπής Ερευνών του ΑΤΕΙΘ. Είναι μέλος του Συμβουλίου Ιδρύματος. Διαθέτει μεγάλη εμπειρία στους τομείς της ηλεκτρονικής μάθησης, της εξ αποστάσεως εκπαίδευσης και της ανάπτυξης προγραμμάτων σπουδών. Υπήρξε Ιδρυματικός Υπεύθυνος εκ μέρους του ΑΤΕΙΘ σε ευρωπαϊκά έργα με αντικείμενο την Ηλεκτρονική Μάθηση και τη Διαδικτυακή Εξ Αποστάσεως Εκπαίδευση, όπως τα DoODL, EuroCompetence, CIOC, MENU και dCCDFLITE. Επίσης ήταν μέλος της 3-μελους επιστημονικής επιτροπής υλοποίησης του Ψηφιακού Εκδοτικού Κέντρου του ΑΤΕΙΘ, που αναπτύχθηκε στο πλαίσιο του προγράμματος ΕΠΕΑΕΚ Ι και Επιστημονικά Υπεύθυνος του ΑΤΕΙ Θεσσαλονίκης, στο πλαίσιο του προγράμματος ΕΠΕΑΕΚ ΙΙ, για την Αναμόρφωση των Προπτυχιακών Προγραμμάτων Σπουδών.

Home Page: http://www.it.teithe.gr/~demos

ΚΕΦΑΛΑΙΟ 1: Εισαγωγή – Ιστορικά στοιχεία – Σχέση Λογικού Προγραμματισμού με την Τεχνητή Νοημοσύνη

Λέξεις Κλειδιά
Τεχνητή Νοημοσύνη, Συμβολικός Υπολογισμός, Λογικός Προγραμματισμός, Prolog

Περίληψη
Στο κεφάλαιο αυτό πραγματευόμαστε την εξέλιξη του Λογικού Προγραμματισμού (ΛΠ) και της γλώσσας Prolog ως μίας ξεχωριστής μεθοδολογίας προγραμματισμού. Κάνουμε μία σύντομη αναφορά στην ιστορική εξέλιξη τους, τη σχέση τους με την έννοια του Συμβολικού Προγραμματισμού και με τον τομέα της Τεχνητής Νοημοσύνης (ΤΝ) και των εφαρμογών της.

Μαθησιακοί Στόχοι

Με την ανάγνωση αυτού του κεφαλαίου, ο αναγνώστης θα έχει μία πρώτη εικόνα του τι είναι ο Λογικός Προγραμματισμός και πως προκύπτει η γλώσσα Prolog ως μία συγκεκριμένη υλοποίηση του. Επίσης η σύντομη ξενάγηση στην ιστορία της εξέλιξης του ΛΠ θα του δώσει τη δυνατότητα να κατανοήσει πως εξελίχθηκε ο ΛΠ στο πλαίσιο της Τεχνητής Νοημοσύνης, πως σχετίζεται με τις τεχνικές της και ποια είναι η σημαντική διαφοροποίησή του από τις κλασικές γλώσσες προγραμματισμού. Τέλος ο αναγνώστης θα βρει τις κατάλληλες αναφορές για να εμβαθύνει, εφόσον το επιθυμεί, σε όλες τις παραπάνω έννοιες, η αναλυτική παρουσίαση των οποίων ξεφεύγει από τους στόχους αυτού του βιβλίου.

1.1 Εισαγωγή

Ενώ εκατοντάδες γλώσσες προγραμματισμού έχουν αναπτυχθεί και υλοποιηθεί από διάφορες ερευνητικές ομάδες, εταιρείες ή και διεθνείς επιτροπές, πολλές από αυτές δεν χρησιμοποιήθηκαν ποτέ έξω από τα στενά πλαίσια των ομάδων που τις σχεδίασαν. Άλλες πάλι, ενώ χρησιμοποιήθηκαν ιδιαίτερα για κάποιο χρονικό διάστημα, αντικαταστάθηκαν από πιο σύγχρονες. Παρόλα αυτά υπάρχει σήμερα ένας μεγάλος αριθμός γλωσσών προγραμματισμού εν ενεργεία. Γιατί λοιπόν ακόμα μία γλώσσα προγραμματισμού; Στο βιβλίο αυτό αναφερόμαστε στην αναγκαιότητα του λογικού προγραμματισμού και της γλώσσας Prolog και εξηγούμε τη σχέση τους με την Τεχνητή Νοημοσύνη, αλλά και γενικότερα με τις τεχνικές και τις μεθοδολογίες ανάπτυξης προγραμματιστικών συστημάτων.

Οι γλώσσες προγραμματισμού της Τεχνητής Νοημοσύνης (ΤΝ) αποτελούν τα βασικά εργαλεία για την ανάπτυξη προγραμματιστικών εφαρμογών που μπορούν να χρησιμοποιηθούν για την προσομοίωση ευφυών διεργασιών, όπως είναι η δυνατότητα συλλογιστικής και επίλυσης προβλημάτων, η μάθηση και η κατανόηση συμβολικής πληροφορίας.

Η ανάπτυξη των πρώτων γλωσσών προγραμματισμού, όπως για παράδειγμα η FORTRAN, βασίστηκε στην ανάγκη υλοποίησης αριθμητικών υπολογισμών. Πολύ γρήγορα, στη συνέχεια, διαπιστώθηκε η ανάγκη αναπαράστασης και επεξεργασίας, όχι μόνο αριθμητικών δεδομένων, αλλά και αντικειμένων και οντοτήτων οποιασδήποτε μορφής. Η ανάγκη αυτή οδήγησε στην έννοια του συμβολικού υπολογισμού (symbolic computation). Με τη βοήθεια του συμβολικού υπολογισμού γίνεται δυνατή η ανάπτυξη αλγορίθμων επεξεργασίας πληροφορίας οποιοδήποτε τύπου, μέσω λειτουργιών που δίνουν τη δυνατότητα αναπαράστασης κανόνων για τη δημιουργία, τη συσχέτιση και τον χειρισμό συμβολικών δεδομένων. Ο συμβολικός υπολογισμός οδήγησε στην ανάπτυξη γλωσσών προγραμματισμού υψηλότερου επιπέδου αφαίρεσης (abstraction) που χρησιμοποιήθηκαν ευρύτατα για την υλοποίηση ευφυών συστημάτων.

Υπάρχει ένας αριθμός Γλωσσών Προγραμματισμού που αναπτύχθηκαν και έγιναν γνωστές ως Γλώσσες της Τεχνητής Νοημοσύνης, καθώς χρησιμοποιήθηκαν σχεδόν αποκλειστικά για την ανάπτυξη εφαρμογών

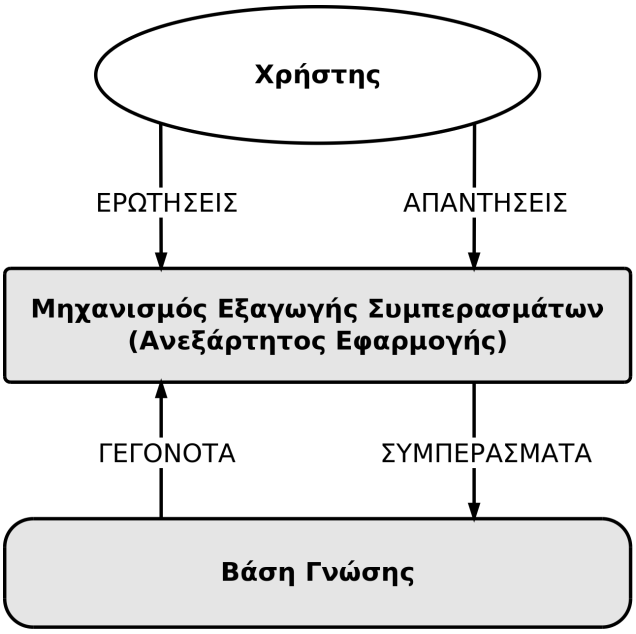
ευφυών συστημάτων. Οι δύο ποιο γνωστές από αυτές είναι η LISP και η Prolog. Η πρώτη βασίζεται στη Συναρτησιακή Λογική και η δεύτερη στη Μαθηματική Λογική.

Η LISP (LISt Processor) αποτελεί την πρώτη πρακτική γλώσσα προγραμματισμού, που χρησιμοποιήθηκε από την κοινότητα της ΤΝ και αναπτύχθηκε από τον John McCarthy στα τέλη της δεκαετίας του 1950. Στη LISP έχουν γραφεί πολλές σημαντικές εφαρμογές ευφυών συστημάτων (όπως στους τομείς της αναγνώρισης φυσικής γλώσσας, της ανάκτησης πληροφοριών, της μηχανικής μάθησης και της κατάστρωσης πλάνων), οι οποίες είχαν μεγάλη επιρροή στην εξέλιξη της ΤΝ. Η LISP αποτέλεσε και τον πρόγονο αρκετών σύγχρονων συναρτησιακών γλωσσών προγραμματισμού, όπως είναι η SML, η Haskell και η Erlang.

Η Prolog εμφανίστηκε αργότερα, τη δεκαετία του 1970 ως μία εναλλακτική λύση για τον προγραμματισμό συστημάτων συμβολικού υπολογισμού και αρχικά προέκυψε από την επιτυχημένη προσπάθεια ανάπτυξης συστημάτων αυτόματης απόδειξης θεωρημάτων (automatic theorem proving). Ο προγραμματισμός στην Prolog συνίσταται στον ορισμό γεγονότων που ορίζουν τη σχέση ανάμεσα σε δεδομένα/αντικείμενα και κανόνων μέσω των οποίων επάγονται νέες σχέσεις ανάμεσα σε δεδομένα/αντικείμενα με τη βοήθεια άλλων σχέσεων που έχουν ήδη οριστεί. Ένα πρόγραμμα στην Prolog είναι μία συλλογή από φράσεις (γεγονότα και κανόνες) που δηλώνουν **ποιο** είναι το προς επίλυση πρόβλημα, χωρίς να περιγράφουν **πως** πρέπει να υπολογιστεί το αποτέλεσμα που προκύπτει από τη λύση του προβλήματος. Οι φράσεις του προγράμματος της Prolog μπορεί να θεωρηθεί ότι ορίζουν μία Βάση Γνώσης (Knowledge Base) από την οποία μπορεί να συνεπάγεται η εξαγωγή νέας γνώσης, μέσω ενός περιβάλλοντος εκτέλεσης (συνήθως ενός διερμηνέα), το οποίο είναι ανεξάρτητο του προβλήματος. Με τη βοήθεια ερωτήσεων που υποβάλει ο χρήστης, διαπιστώνεται εάν κάποιες σχέσεις ισχύουν ανάμεσα σε κάποιες οντότητες (σχήμα 1.1).

Με τον τρόπο αυτό, το πρόγραμμα περιγράφει το πρόβλημα σε ένα πολύ υψηλό επίπεδο αφαίρεσης, το οποίο είναι αντίστοιχο με εκείνο το οποίο στη μηχανική λογισμικού αποτελεί την τυπική προδιαγραφή του (formal specification). Αυτή η μορφή του προγραμματισμού είναι τελείως διαφορετική τόσο από τον προγραμματισμό των διατακτικών (imperative) γλωσσών προγραμματισμού, όπως η η C/C++ και η Java, όσο και από τον προγραμματισμό των συναρτησιακών γλωσσών, όπως η LISP, SML, Haskell, Erlang, όπου η εστίαση είναι στο ποια είναι τα βήματα εκτέλεσης για την επίλυση του προβλήματος, ή στον τρόπο υπολογισμού της λύσης.

Η Prolog χρησιμοποιήθηκε με επιτυχία για την ανάπτυξη ευφυών συστημάτων σε πεδία της ΤΝ όπως είναι η επεξεργασία φυσικής γλώσσας, τα έμπειρα συστήματα και τα συστήματα γνώσης, όπως και σε άλλους τομείς όπως τα συστήματα σχεσιακών βάσεων δεδομένων και η εκπαίδευση.



Σχήμα 1.1: *Το περιβάλλον εκτέλεσης της Prolog*

1.2 Σύντομη Ιστορία του Λογικού Προγραμματισμού και της Prolog

Όπως αναφέρθηκε εισαγωγικά, η Prolog εξελίχθηκε σε μία ξεχωριστή "σχολή" προγραμματισμού που βασίζεται στη Μαθηματική Λογική. Η Μαθηματική Λογική είναι ένα σημαντικό εργαλείο για την ανάλυση και την παρουσίαση επιχειρημάτων. Διερευνά τη σχέση ανάμεσα σε υποθέσεις και συμπεράσματα, αν δηλαδή κάποιες υποθέσεις μπορούν να συνεπάγουν ένα ή περισσότερα συμπεράσματα. Για παράδειγμα οι υποθέσεις:

```
Η Νίκη συμπαθεί τους άνδρες με χιούμορ
Ο Κώστας έχει χιούμορ
```

συνεπάγουν το συμπέρασμα ότι:

```
Η Νίκη συμπαθεί τον Κώστα
```

Μία ειδική μορφή μαθηματικής λογικής, οι φράσεις Horn (Horn clauses), που με τη σειρά τους αποτελούν ένα υποσύνολο του κατηγορηματικού λογισμού 1ης τάξης (1st order predicate calculus) αποτέλεσαν τη βάση ανάπτυξης μιας γλώσσας λογικού προγραμματισμού για την επίλυση προβλημάτων.

Ο λογικός προγραμματισμός με τη μορφή που απέκτησε σήμερα έχει τις ρίζες του στα τέλη της δεκαετίας του 1970, όταν στα πλαίσια της ερευνητικής κοινότητας της Τεχνητής Νοημοσύνης είχε ανοίξει μία μεγάλη συζήτηση (πολλές φορές με έντονες διαφωνίες) σε σχέση με το δηλωτικό (declarative) τρόπο αναπαράστασης της γνώσης σε αντίθεση με το διαδικαστικό (procedural) τρόπο.

Οι υποστηρικτές της δηλωτικής αναπαράστασης εργάζονταν κυρίως στο Πανεπιστήμιο του Στάνφορντ όπως οι John McCarthy, Bertman Raphael και Cordell Green και στο πανεπιστήμιο του Εδιμβούργου, όπως οι Robert Kowalski, Pat Hayes και ο John Alan Robinson (ο οποίος προερχόμενος από το πανεπιστήμιο Syracuse βρισκόταν εκεί ως ακαδημαϊκός επισκέπτης). Στους υπέρμαχους της διαδικαστικής αναπαράστασης ήταν κυρίως ο Marvin Minsky και ο Seymour Papert και οι ερευνητικές τους ομάδες στο MIT. Αξίζει να σημειωθεί ότι εξαιτίας της δουλειάς όλων των παραπάνω ερευνητών το πεδίο της ΤΝ γνώρισε μία αλματώδη ανάπτυξη. Τα πρόσωπα αυτά αποτελούν σήμερα μερικούς από τους θρύλους τόσο της ΤΝ όσο και της επιστήμης των υπολογιστών γενικότερα.

Αν θα έπρεπε να διαλέξουμε 3 πρόσωπα "κλειδιά" που είχαν καταλυτική συνεισφορά στην εξέλιξη του Λογικού Προγραμματισμού πρέπει να αναφέρουμε τους John Alan Robinson, Robert Kowalski, και Alain Colmerauer. Ο πρώτος με την ερευνητική του δουλειά για την αρχή της ανάλυσης (resolution principle) και ιδιαίτερα με την εργασία του "A machine oriented Logic based on the Resolution Principle" (Robinson, 1965), ο δεύτερος, ο οποίος απέδειξε ότι η λογική μπορεί να χρησιμοποιηθεί ως γλώσσα προγραμματισμού και ο τρίτος γιατί υλοποίησε την πρώτη πρακτική γλώσσα που βασίζεται στο λογικό προγραμματισμό.

Ο Alan Robinson ανέπτυξε μία τεχνική αυτόματης απόδειξης θεωρημάτων με το στόχο οι μαθηματικές αποδείξεις να γίνονται μηχανικά μέσω μιας υπολογιστικής μηχανής. Η τεχνική που εισήγαγε αποτελείται από έναν μοναδικό κανόνα συμπερασμού (inference rule) μέσω του οποίου αποδεικνύεται ότι από ένα σύνολο υποθέσεων (assumptions) Α συνεπάγεται ένα θεώρημα G. Η διαδικασία απόδειξης με τη βοήθεια του κανόνα της ανάλυσης, έδειξε ότι η μαθηματική λογική και ειδικότερα κατηγορηματικός λογισμός μπορεί να χρησιμοποιηθεί ως τυπικός συμβολισμός για τον ορισμό αλγορίθμων και συνεπώς για την πραγματοποίηση συμβολικών υπολογισμών.

Ο Robert Kowalski, βασιζόμενος στη δουλειά του Alan Robinson, ανέπτυξε μία παραλλαγή του κανόνα της ανάλυσης με την επωνυμία Selection Linear Definite clause (SLD) resolution (Kowalski and Kuehner, 1971) και διατύπωσε την άποψη ότι ο υπολογισμός στον λογικό προγραμματισμό μπορεί να εκφραστεί ως μία ελεγχόμενη διαδικασία επαγωγής συμπερασμάτων (controlled deduction) από δηλωτικές φράσεις. Επίσης όρισε τη διαδικαστική ερμηνεία της λογικής που βασίζεται στις φράσεις Horn και απέδειξε ότι φράσεις της μορφής G if A μπορεί να ερμηνευτούν ως διαδικασίες μιας αναδρομικής γλώσσας προγραμματισμού, όπου το G θεωρείται η επικεφαλίδα της διαδικασίας και Α είναι το σώμα της. Η διπλή δυνατότητα ερμηνείας των φράσεων Horn και ως προς τη δηλωτική αλλά και ως προς τη διαδικαστική σημασιολογία τους οδήγησε στο συμπέρασμα ότι ο Λογικός Προγραμματισμός μπορεί να αποτελέσει τη βάση μιας υλοποιήσιμης γλώσσας προγραμματισμού.

Η ιστορία της γλώσσας προγραμματισμού Prolog, ως μία ειδική περίπτωση υλοποίησης ενός συστήματος Λογικού Προγραμματισμού, ξεκινάει το 1971-1972 με τη δημιουργία του πρώτου διερμηνέα (interpreter) στο πανεπιστήμιο της Μασσαλίας, στη Γαλλία από τους Alain Colmerauer και Philippe Roussel (Colmerauer et. al, 1973; Battani and Meloni, 1973). Ο αρχικός στόχος της ερευνητικής ομάδας ήταν να αναπτύξει ένα σύστημα επεξεργασίας φυσικής γλώσσας (της Γαλλικής), στη συνέχεια όμως το σύστημα αυτό εξελίχθηκε σε γλώσσα προγραμματισμού (Colmerauer and Roussel, 1993). Το όνομα Prolog αναφέρεται ότι προτάθηκε από τη Jacqueline Roussel, γυναίκα του Philippe Roussel ως συγκοπτόμενη λέξη της φράσης "PROgrammation en LOGique". Η ομάδα της Μασσαλίας συνεργάστηκε στενά με τον Kowalski και με τον τρόπο αυτό συνδυάστηκε η προσπάθεια της ανάπτυξης του συστήματος επεξεργασίας φυσικής γλώσσας των Γάλλων με την εξειδικευμένη εφαρμογή του κανόνα της ανάλυσης .

Μια προηγούμενη αντίστοιχη προσέγγιση για την υλοποίηση ενός συστήματος Λογικού Προγραμματισμού μπορεί να θεωρηθεί αυτή που οδήγησε στην υλοποίηση των συστημάτων Planner και MicroPlanner στο MIT της Αμερικής (Hewitt, 1969; Sussman et al., 1970), χωρίς όμως τα συστήματα αυτά να μετεξελιχθούν σε γλώσσα προγραμματισμού που χρησιμοποιήθηκε σε ευρεία κλίμακα.

Ο πρώτος μεταγλωττιστής (compiler) της γλώσσας υλοποιήθηκε το 1977 από την ομάδα του David Warren, στο πανεπιστήμιο του Εδιμβούργου (Pereira et. al., 1978). Ο μεταγλωττιστής αυτός βασίστηκε στις τεχνικές υλοποίησης της γλώσσας που αναπτύχθηκαν από την ομάδα της Μασσαλίας επεκτείνοντάς τες με τεχνικές βελτιστοποίησης της μνήμης και μετέφραζε την Prolog σε γλώσσα μηχανής του υπολογιστή DEC-10. Για το λόγο αυτό έγινε γνωστή ως DEC-10 Prolog. Στη συνέχεια ο Warren βασιζόμενος στις τεχνικές υλοποίησης της DEC-10 Prolog, σχεδίασε μία αρχιτεκτονική αφηρημένης μηχανής για την εκτέλεση κώδικα Prolog, που είχε ως βάση μία αρχιτεκτονική διαχείρισης μνήμης και ένα ειδικό σύνολο εντολών (Warren 1983). Η αρχιτεκτονική αυτή έγινε γνωστή στην επιστημονική κοινότητα του λογικού προγραμματισμού ως Warren Abstract Machine (WAM) και αποτέλεσε ένα ντε-φάκτο σταθερότυπο (de-facto standard) πάνω στο οποίο βασίστηκε η υλοποίηση πολλών μεταγλωττιστών και διερμηνέων της Prolog που αναπτύχθηκαν από διάφορες ομάδες.

Κατά την επόμενη δεκαετία υπήρξε ένα ολοένα αυξανόμενο ενδιαφέρον από διάφορα ερευνητικά κέντρα της Ευρώπης, της Ιαπωνίας και της Αμερικής για την εξέλιξη και την υλοποίηση της γλώσσας. Σημαντικό σταθμό στην εξέλιξη της Prolog αποτέλεσε το πρόγραμμα 5ης γενιάς υπολογιστών (5th Generation Computer Systems) της Ιαπωνίας (MotoOka, 1981; Shapiro, 1983). Το πρόγραμμα αυτό είχε ως στόχο τη δημιουργία ενός επαναστατικά νέου τύπου υπολογιστικού συστήματος που θα ήταν κατάλληλο για επεξεργασία γνώσης (μηχανή γνώσης). Η κεντρική μονάδα επεξεργασίας αυτού του συστήματος επιλέχθηκε να βασιστεί σε μία παράλληλη αρχιτεκτονική επαγωγής συμπερασμών (Parallel Inference Machine/PIM) και να χρησιμοποιεί ως γλώσσα μηχανής μία υψηλού επιπέδου παράλληλη γλώσσα λογικού προγραμματισμού με την ονομασία KL1. Παρόλο που το πρόγραμμα των Ιαπώνων αποδείχθηκε ιδιαίτερα φιλόδοξο και δεν πέτυχε στη δεκαετή διάρκειά του το συνολικό τελικό του στόχο, τα επιμέρους επιτεύγματα του έδωσαν μεγάλη ώθηση στην έρευνα στον τομέα του λογικού προγραμματισμού σε διεθνές επίπεδο.

Σημαντική συνεισφορά στην ανάπτυξη του Λογικού Προγραμματισμού και της Prolog είχαν διάφορες ερευνητικές ομάδες, όπως αυτές των Πορτογάλων (Πανεπιστήμιο Νέας Λισαβόνας) και των Ούγγρων (πανεπιστήμιο Βουδαπέστης) και στη συνέχεια και Αμερικάνων όπως αυτή του ερευνητικού κέντρου SRI.

Το 1995, αρκετά αργά σε σχέση με την πρώτη της υλοποίηση, η Prolog απέκτησε την τυπική περιγραφή της κατά ISO (*ISO/IEC 13211-1*). Η τυπική περιγραφή της γλώσσας παρουσιάστηκε από την σχετική ομάδα εργασίας που είχε οριστεί το 1987 (*ISO/IEC JTC1/SC22 group WG17*) και βασίζεται σε μεγάλο μέρος στο συντακτικό της γλώσσας DEC-10 Prolog του Εδιμβούργου (Deransart et.al 1996).

Τέλος αξίζει να αναφερθεί ότι σημαντικό σταθμό στην εξέλιξη του Λογικού Προγραμματισμού αποτέλεσε η ίδρυση το 1986 της διεθνούς ένωσης του Λογικού Προγραμματισμού (*Association for Logic Programming* - http://www.logicprogramming.org*)*

1.3 Η Prolog και οι εφαρμογές της Τεχνητής Νοημοσύνης

Υπάρχουν μία σειρά χαρακτηριστικών της Prolog, που την ανέδειξαν ως κατάλληλη γλώσσα για την ανάπτυξη εφαρμογών ΤΝ. Τα χαρακτηριστικά αυτά θα οριστούν και θα συζητηθούν διεξοδικά στα κεφάλαια του βιβλίου που ακολουθούν. Εδώ αναφέρονται με έναν γενικό εισαγωγικό τρόπο για να γίνει δυνατή η συσχέτισή τους με τους λόγους που ο λογικός προγραμματισμός και η RROLOG χρησιμοποιήθηκαν ως αποτελεσματικά εργαλεία ανάπτυξης ευφυών συστημάτων. Τα χαρακτηριστικά αυτά είναι το συντακτικό και η σημασιολογία της γλώσσας, ο μηχανισμός ενοποίησης και ο μηχανισμός της οπισθοδρόμησης:

- **Συντακτικό - Δηλωτική σημασιολογία:** Κατ' αρχήν, όπως αναφέρθηκε, το συντακτικό και η σημασιολογία που διαθέτει η Prolog βασίζεται στην τυπική λογική, η οποία έχει χρησιμοποιηθεί ευρύτατα για την αναπαράσταση γνώσης και την υλοποίηση μεθόδων συλλογιστικής που εξελίχθηκαν διαχρονικά στην Τεχνητή Νοημοσύνη. Τεχνικές αναπαράστασης γνώσης και συλλογιστικής, όπως οι **κανόνες παραγωγής (production rules)**, τα **πλαίσια (frames)** και τα **σημασιολογικά δίκτυα (semantic nets)** υλοποιούνται με μεγάλη ευκολία στη γλώσσα Prolog. Επιπλέον το υψηλό επίπεδο αφαίρεσης που προσδίδει στη γλώσσα η δηλωτική φύση της την καθιστά ιδιαίτερα "ανθρωποκεντρική", καθώς δίνει τη δυνατότητα στον προγραμματιστή να επικεντρωθεί άμεσα στον τρόπο περιγραφής του προβλήματος. Επίσης τα απλό συντακτικό της Prolog την καθιστά γλώσσα ελάχιστη αλλά επεκτάσιμη και επιπλέον μεταγλώσσα, χαρακτηριστικά ιδιαίτερα χρήσιμα για την αναπαράσταση γνώσης και συλλογιστικής.
- **Μηχανισμός Ενοποίησης όρων:** Η Prolog διαθέτει ένα μηχανισμό ενοποίησης (unification), ο οποίος συνδέεται με τον μηχανισμό της ανάλυσης και ο οποίος επιτρέπει την εύρεση του ποιο γενικού κοινού στιγμιότυπου δύο όρων ή ατομικών τύπων, αντικαθιστώντας μεταβλητές κατάλληλα. Η χρήση της ενοποίησης καθιστά τη δυνατότητα **ταιριάσματος μορφότυπων (pattern matching)**, ουσιαστικά ενσωματωμένο μηχανισμό της γλώσσας δίνοντάς της με τον τρόπο αυτό τη δυνατότητα για την υλοποίηση αυτού του είδους επίλυσης προβλημάτων που χρησιμοποιείται σε πολλές εφαρμογές της Τεχνητής Νοημοσύνης. Ο μηχανισμός της ενοποίησης αποτελεί ταυτόχρονα και τον μηχανισμό περάσματος παραμέτρων της γλώσσας, ο οποίος είναι ιδιαίτερα ευφυής. Κατά την κλήση των διαδικασιών δίνει τη δυνατότητα τα ορίσματά τους να είναι διπλής κατεύθυνσης: είτε ορίσματα εισόδου είτε ορίσματα εξόδου. Με τον τρόπο αυτόν οι διαδικασίες αυτές, χωρίς να αλλάξει ο ορισμός τους, μπορούν να χρησιμοποιηθούν με πολλούς διαφορετικούς τρόπους συλλογισμού. Έτσι, παραδείγματος χάριν, ένα πρόγραμμα που έχει οριστεί για να συνενώνει δύο λίστες (παράθεση λιστών), μπορεί να χρησιμοποιηθεί και αντίστροφα για να παραγάγει όλα τα ζεύγη λιστών, η συνένωση των οποίων έχει ως αποτέλεσμα αυτή την λίστα.
- **Μηχανισμός οπισθοδρόμησης:** Η διαδικασία εκτέλεσης με βάση την υλοποίηση της αρχής της ανάλυσης της Prolog βασίζεται επίσης σε ένα αυτόματο μηχανισμό οπισθοδρόμησης (backtracking). Η δυνατότητα της αυτόματης οπισθοδρόμησης καθιστά εξαιρετικά εύκολη την περιγραφή **αλγορίθμων αναζήτησης**, οι οποίοι θεωρούνται κεντρικής σημασίας σε πάρα πολλές εφαρμογές της Τεχνητής Νοημοσύνης. Και στην περίπτωση αυτή ο προγραμματιστής περιγράφει αλγόριθμους αναζήτησης σε ένα πολύ υψηλότερο επίπεδο αφαίρεσης σε σχέση με τις άλλες γλώσσες προγραμματισμού. Όταν ο αλγόριθμος αναζήτησης φθάνει σε αδιέξοδο μονοπάτι, ο μηχανισμός οπισθοδρόμησης, αυτόματα, προκαλεί αναζήτηση σε νέο μονοπάτι. Επιπλέον ο ίδιος μηχανισμός παρέχει το επιπλέον πλεονέκτημα της εύρεσης περισσοτέρων της μίας λύσεων εφόσον αυτές υπάρχουν. Καθώς λοιπόν ένας ιδιαίτερα μεγάλος αριθμός προβλημάτων της Τεχνητής Νοημοσύνης αναπαριστάνονται ως προβλήματα εύρεσης του κατάλληλου μονοπατιού σε κάποιο χώρο αναζήτησης ή αντίστοιχα ως προβλήματα αναζήτησης σε γράφους, διάφορα συστήματα βασισμένα στην Prolog αναπτύχθηκαν για την επίλυση προβλημάτων χώρου αναζήτησης και της θεωρίας των γράφων. Η Prolog, επίσης, αποδείχθηκε ιδιαίτερα χρήσιμη για ανάπτυξη προγραμμάτων για παίγνια ή εφαρμογών που βασίζονται σε αλγόριθμους παιγνίων, καθώς ο χώρος αναζήτησης που δημιουργείται στην περίπτωση ενός παιγνίου μπορεί να αντιμετωπιστεί αποτελεσματικά από τους μηχανισμούς που προαναφέρθηκαν.

Επιπλέον πεδία εφαρμογών στα οποία η γλώσσα Prolog έχει χρησιμοποιηθεί με επιτυχία και αξίζει να σημειωθούν είναι τα ακόλουθα:

- **Η Επεξεργασία φυσικής Γλώσσας:** Οι δυνατότητες ταιριάσματος μορφότυπων και η δηλωτική φύση των ορισμών γραμματικής, κάνουν την Prolog εύχρηστο και ισχυρό εργαλείο για την επεξεργασία φυσικής γλώσσας. Άλλωστε όπως αναφέρθηκε ένα σύστημα επεξεργασίας φυσικής γλώσσας οδήγησε στη "γέννηση" της Prolog.
- **Τα Έμπειρα Συστήματα /Συστήματα βασισμένα στη Γνώση:** Η υλοποίηση των συστημάτων αυτών βασίζεται κατά κύριο λόγο σε μία βάση γνώσης (knowledge base) και σε μία μηχανή επαγωγής συμπερασμάτων (inference engine). Τις περισσότερες φορές, η βάση γνώσης εκφράζεται με τη μορφή κανόνων παραγωγής, η υλοποίηση των οποίων γίνεται άμεσα με τη χρήση των κανόνων της Prolog. Επίσης ο μηχανισμός επαγωγής συμπερασμάτων της Prolog που βασίζεται στην αρχή της ανάλυσης είναι αντίστοιχος με την τεχνική της ανάστροφης συλλογιστικής (backward reasoning) που χρησιμοποιείται, σε πολλές περιπτώσεις, για την υλοποίηση της μηχανής επαγωγής συμπερασμάτων. Άλλες μορφές συλλογιστικής είναι επίσης εύκολο να υλοποιηθούν με τη βοήθεια των μεταγλωσσικών δυνατοτήτων της Prolog. Μία ειδική κατηγορία συστημάτων γνώσης είναι τα **Συστήματα Λήψης Αποφάσεων (Decision Support Systems)** για τα οποία η Prolog επίσης αποτελεί κατάλληλο εργαλείο ανάπτυξης τους, καθώς οι κανόνες απόφασης στους οποίους βασίζονται μπορούν να υλοποιηθούν κατάλληλα.
- **Κατάστρωση Πλάνων και Χρονοπρογραμματισμός (Planning and Scheduling):** Μία άλλη περιοχή της Τεχνητής Νοημοσύνης στην οποία η Prolog αποτελεί κατάλληλο εργαλείο, είναι αυτή της κατάστρωσης πλάνων και του χρονοπρογραμματισμού. Ο μηχανισμός ενοποίησης με τις δυνατότητες ταιριάσματος περιπτώσεων, η δυνατότητα εφαρμογής τεχνικών παραγωγής και δοκιμής (generate and test) και η χρήση των κανόνων της γλώσσας αποτελούν στοιχεία κατάλληλα για αυτού του είδους τις εφαρμογές. Σημαντικό ρόλο επίσης στην υλοποίηση αυτών των εφαρμογών παίζουν και οι μηχανισμοί επαύξησης της Prolog για τη δυνατότητα έκφρασης και επίλυσης προβλημάτων με ικανοποίηση περιορισμών (constraint satisfaction).
- **Βάσεις Δεδομένων:** Η Prolog έχει επίσης συνεισφέρει στην ευφυή αναζήτηση πληροφορίας σε μεγάλες βάσεις δεδομένων, λόγω της φύσης της κατά την οποία ένα οποιοδήποτε πρόγραμμά της ενεργοποιείται με τη βοήθεια στόχων/ερωτήσεων. Η γλώσσα Datalog, που αποτελεί ένα υποσύνολο της Prolog με απόλυτη δηλωτική σημασιολογία, έχει χρησιμοποιηθεί ως γλώσσα ερωτήσεων σε επαγωγικές βάσεις δεδομένων (deductive databases).
- **Σημασιολογικός Ιστός (Semantic Web):** Την τελευταία δεκαετία οι προσπάθειες συνδυασμού των τεχνικών του λογικού προγραμματισμού με τις τεχνολογίες του σημασιολογικού ιστού, έχουν δημιουργήσει μία καινούργια μορφή προγραμματισμού με την επωνυμία Περιγραφικός Λογικός Προγραμματισμός (Description Logic Programming). Η ανάπτυξη του βασίστηκε στην διαπίστωση ότι η χρήση λογικών κανόνων που αναφέρονται σε κάποιο συγκεκριμένο πεδίο μπορεί να βελτιώσει τον σημασιολογικό ιστό. Ο περιγραφικός λογικός προγραμματισμός παρέχει ένα βαθμό εκφραστικότητας σημαντικά μεγαλύτερο από αυτόν της περιγραφικής λογικής του RDF-σχήματος.

1.4 Υλοποιήσεις της Prolog

Από την εισαγωγή της γλώσσας Prolog στις αρχές της δεκαετίας του '70 μέχρι σήμερα, καταγράφεται ένα τεράστιο πλήθος υλοποιήσεων της, καθώς και περιβαλλόντων ανάπτυξης που την υποστηρίζουν. Αναφέρουμε ενδεικτικά μερικές χαρακτηριστικές από αυτές τις υλοποιήσεις (με τον κίνδυνο να αδικήσουμε πολλές άλλες):

- Η SWI-Prolog (http://www.swi-prolog.org), αποτελεί μια ελεύθερη έκδοση της γλώσσας, υλοποιεί το ISO/IEC standard υποσύνολο της και αποτελεί ένα από τα πλέον χρησιμοποιούμενα περιβάλλοντα της Prolog από την ερευνητική και εκπαιδευτική κοινότητα. Έχει επίσης χρησιμοποιηθεί ευρύτατα για ανάπτυξη σημαντικών εμπορικών εφαρμογών. Διαθέτει βιβλιοθήκες που υποστηρίζουν, μεταξύ άλλων, προγραμματισμό με περιορισμούς, ανάπτυξη εφαρμογών επεξεργασίας φυσικής γλώσσας, εφαρμογών του διαδικτύου και σημασιολογικού ιστού. Διαθέτει επίσης μηχανισμούς διασύνδεσης με

τη γλώσσα προγραμματισμού Java και ODBC για σύνδεση με βάσεις δεδομένων. Η πρώτη υλοποίηση της γλώσσας ξεκίνησε το 1987 στο πανεπιστήμιο του Άμστερνταμ και από τότε έχουν καταγραφεί πάνω από ένα εκατομμύριο χρήστες που την έχουν εγκαταστήσει και χρησιμοποιήσει.

- Η ECLiPSe (http://eclipseclp.org/), αποτελεί ένα υπερσύνολο της γλώσσας Prolog που περιλαμβάνει χαρακτηριστικά που την καθιστούν ιδιαίτερα κατάλληλη για ανάπτυξη εφαρμογών που βασίζονται στον λογικό προγραμματισμό με περιορισμούς. Είναι επίσης κατάλληλη για την αντιμετώπιση προβλημάτων μεγάλης συνδυαστικής πολυπλοκότητας και γενικότερα αποτελεί ένα ισχυρό εργαλείο μοντελοποίησης συστημάτων. Επίσης διαθέτει υποσυστήματα διασύνδεσης με άλλα περιβάλλοντα προγραμματισμού (C++ και Java). Στην ανάπτυξη της ECLiPSe συμμετείχαν ιδιαίτερα σημαντικά κέντρα έρευνας και ανάπτυξης της επιστήμης των υπολογιστών, όπως το ECRC (European Computer-Industry Research Centre) του Μονάχου και το IC-Parcat (Centre for Planning and Resource Control) του Imperial College του Λονδίνου. Σήμερα αποτελεί σύστημα λογισμικού ανοικτού κώδικα που προσφέρεται με άδεια MPL (Mozilla Public License).
- Η YAP (Yet Another Prolog) (http://www.dcc.fc.up.pt/~vsc/Yap/), είναι μία υλοποίηση της standard Prolog που διαθέτει έναν μεταγλωττιστή υψηλής αποδοτικότητας, ο οποίος βασίζεται στην εξέλιξη της αρχιτεκτονικής WAM. Συμπληρώνει 30 χρόνια ζωής και η συνεχής ανάπτυξή της υποστηρίζεται από το Εργαστήριο Τεχνητής Νοημοσύνης και Επιστήμης Υπολογιστών του πανεπιστημίου του Πόρτο και του Ινστιτούτου Μηχανικής των Συστημάτων και Υπολογιστών του πανεπιστημίου του Rio de Janeiro της Βραζιλίας. Διατίθεται ως ελεύθερο λογισμικό με άδεια GPL (General Public License).
- Η B-Prolog (hhtp://www.picat-lang.org/bprolog/), είναι επίσης μία υψηλής αποδοτικότητας υλοποίηση της standard Prolog που διατίθεται ελεύθερα για μη εμπορικές εφαρμογές. Περιλαμβάνει επεκτάσεις, όπως η δυνατότητα χρήσης πινάκων (arrays), πινάκων κατακερματισμού (hash tables), δηλωτικών δομών επανάληψης, ο ορισμός κανόνων ενεργειών (action rules) για το χειρισμό συμβάντων και η επίλυση προβλημάτων περιορισμών (finite-domain constraint solving).
- Η SICStus Prolog (https://sicstus.sics.se), αποτελεί μια κορυφαία εμπορική υλοποίηση της γλώσσας, με πολύ ισχυρά χαρακτηριστικά, όπως πολύ αποδοτική υποστήριξη περιορισμών, σύγχρονο περιβάλλον ανάπτυξης, και μια ιστορία εξέλιξης πάνω από 20 χρόνια. Αναπτύσσεται και υποστηρίζεται από το Σουηδικό Ινστιτούτο Επιστήμης Υπολογιστών (Swedish Institute of Computer Science), από όπου και το όνομά της.

Πρέπει να αναφέρουμε ότι η Prolog και ο λογικός προγραμματισμός γενικότερα, στην πλέον των 30 ετών ιστορία τους, έχει κατά καιρούς δημιουργήσει μεγάλες προσδοκίες ως μία εναλλακτική σχολή ανάπτυξης λογισμικού, έχουν όμως δεχτεί και μεγάλη αμφισβήτηση του κατά πόσο μπορούν να ξεπεράσουν τα πειραματικά όρια της ερευνητικής κοινότητας της Τεχνητής Νοημοσύνης που ασχολείται με τη χρήση και την ανάπτυξή τους. Κατά τη γνώμη των συγγραφέων αυτού του βιβλίου ο μεγάλος αριθμός υλοποιήσεων της Prolog και τα σημαντικά κέντρα στα οποία αυτή αναπτύσσεται αποτελούν μια από τις αποδείξεις της σημαντικότητας της γλώσσας και του λόγου για τον οποίο περιβάλλοντα προγραμματισμού που βασίζονται στην Prolog εξελίσσονται συνεχώς.

Για να παραφράσουμε δημιουργικά το απόφθεγμα του Alan Perlis "*Μία γλώσσα που δεν επηρεάζει τον τρόπο που σκέφτεται κανείς για τον προγραμματισμό δεν αξίζει να τη γνωρίσει*", που περιέλαβε στο άρθρο του "Epigrams on Programming" (Perlis, 1982), θα λέγαμε ότι αξίζει να γνωρίσει κανείς τη γλώσσα προγραμματισμού Prolog και τις τεχνικές του λογικού προγραμματισμού που τη συνοδεύουν γιατί θα κατακτήσει έναν εναλλακτικό δημιουργικό τρόπο σκέψης για την ανάπτυξη εφαρμογών λογισμικού. Αυτός ο τρόπος σκέψης θα τον βοηθήσει ανεξάρτητα από τη γλώσσα και τα προγραμματιστικά εργαλεία που θα χρησιμοποιήσει στο μέλλον.

Για τους προγραμματιστές που εργάζονται συστηματικά στην ανάπτυξη λογισμικού και για να αποδεχθούν ένα προγραμματιστικό εργαλείο αναζητούν αν αυτό έχει χρησιμοποιηθεί από σημαντικές εταιρείες του χώρου, αναφέρουμε δύο χαρακτηριστικά παραδείγματα: τη χρήση της Prolog από την IBM στο

πλαίσιο της ανάπτυξης του συστήματος Watson, καθώς και τη χρήση της από την Oracle στο πλαίσιο εξέλιξης της γλώσσας Java.

Το σύστημα της IBM, Watson είναι ένα έξυπνο σύστημα ερωτοαποκρίσεων ανοικτού πεδίου που βασίζεται σε μία εξελιγμένη μορφή επικοινωνίας ανθρώπου-μηχανής (Ferrucci, 2012). Η υλοποίηση συστημάτων ερωτοαποκρίσεων ανοικτού πεδίου (συστημάτων δηλαδή που δεν είναι γνωστό από την αρχή το πεδίο στο οποίο θα γίνει η επικοινωνία) αποτελεί σήμερα μία από τις μεγαλύτερες προκλήσεις της επιστήμης των υπολογιστών και σχετίζεται με τους τομείς της Τεχνητής Νοημοσύνης όπως είναι η ανάκτηση της πληροφορίας από τον παγκόσμιο ιστό, η επεξεργασία φυσικής γλώσσας, η αναπαράσταση γνώσης, η μηχανική μάθηση και η συλλογιστική. Η Prolog επιλέχθηκε από την IBM ως η πιο κατάλληλη γλώσσα προγραμματισμού για την υλοποίηση του συστήματος επεξεργασίας της φυσικής γλώσσας.

Στην τελευταία υλοποίηση της αντικειμενοστραφούς γλώσσας προγραμματισμού Java (έκδοση SE8) χρησιμοποιούνται φράσεις της Prolog για τον ορισμό των κανόνων ελέγχου τύπου (type checking) για την επαλήθευση της ορθότητας των μεταγλωττισμένων κλάσεων που φορτώνονται στην εικονική μηχανή της Java (JVM Specification).

Για τους προγραμματιστές που τους αρέσει η ανάπτυξη έξυπνων παιχνιδιών προτείνουμε να γράψουν τον κώδικα για την επίλυση του Sudoku στη γλώσσα της προτίμησής τους και στη συνέχεια να τον συγκρίνουν με τον αντίστοιχο της Prolog.

Όλα τα προγράμματα του βιβλίου έχουν υλοποιηθεί και δοκιμαστεί με την SWI-Prolog, είναι όμως συμβατά και με όλες τις υλοποιήσεις της Prolog που βασίζονται στο ISO/IEC standard. Ειδικότερα τα παραδείγματα του κεφαλαίου 11, λόγω των ιδιαίτερων χαρακτηριστικών τους, έχουν υλοποιηθεί με την ECLiPSe Prolog. Ο αναγνώστης του βιβλίου μπορεί να ανακτήσει τον κώδικα που αντιστοιχεί στα παραδείγματα των κεφαλαίων του βιβλίου από το σύνδεσμο http://users.uom.gr/~iliass/LPTechniques, όπου θα βρει και τυχόν παρατηρήσεις για τη συμβατότητα του με τις διάφορες υλοποιήσεις της γλώσσας.

Βιβλιογραφία

Βιβλία αναφοράς για τη γλώσσα προγραμματισμού Prolog είναι αυτά των (Clocksin and Mellish, 2003) του (Bratko, 2011), των (Shapiro and Sterling, 1986), των (Covington et al., 1988) και του (O'Keefe, 1990). Το πρώτο σημαντικό ελληνικό βιβλίο για την Prolog γράφτηκε από τους Μαρία Κατζουράκη, Μανόλη Γεργατσούλη και Σταύρο Κόκκοτο το 1991 (Κατζουράκη κ.α., 1991). Τα προγράμματα του βιβλίου είναι γραμμένα στη Δ-Prolog που βασίζεται στον πρώτο ελληνικό διερμηνέα της γλώσσας που αναπτύχθηκε στο Ινστιτούτο Πληροφορικής και Επικοινωνιών του ΕΚΕΦΕ "ΔΗΜΟΚΡΙΤΟΣ". Η πρώτη ενδιαφέρουσα συλλογή προγραμματιστικών προβλημάτων σε Prolog μπορεί να βρεθεί στο βιβλίο των Πορτογάλων Coelho, Cotta και Pereira (Coelho et.al., 1982). Πολύ ενδιαφέροντα στοιχεία για την ιστορική εξέλιξη του λογικού προγραμματισμού και της γλώσσας Prolog μπορούν να βρεθούν στις αναφορές των (Cohen, 1988), (Kowalski, 1988), (Kowalski, 2013), καθώς επίσης και στο βιβλίο του Van Caneghem (Van Caneghem, 1986).

Παραπέμπουμε τον αναγνώστη του βιβλίου που ενδιαφέρεται να εμβαθύνει στη σχέση της μαθηματικής λογικής με το λογικό προγραμματισμό στις αναφορές του Kowalski (Kowalski, 1979) και του Lloyd (Lloyd, 1984).

Πολλά παραδείγματα ανάπτυξης εφαρμογών της Τεχνητής Νοημοσύνης με τη χρήση της γλώσσας Prolog μπορούν να βρεθούν στο βιβλίο των (Βλαχάβας κ.α., 2011). Επίσης μία λίστα εμπορικών εφαρμογών της Τεχνητής Νοημοσύνης, που υλοποιήθηκαν με Prolog, συνοδευομένη με μία σύντομη περιγραφή για την κάθε περίπτωση, μπορεί να βρεθεί στην εργασία του (Tsadiras, 2008).

Τέλος, μία εκτενής συγκριτική μελέτη των πλεονεκτημάτων και τα μειονεκτημάτων της Prolog μπορεί να βρεθεί στην αναφορά του (Bobrow, 1985).

Βλαχάβας, Ι. και Κεφαλάς, Π. και Βασιλειάδης, Ν. και Κόκκορας, Φ. και Σακελλαρίου Η. (2011). *Τεχνητή Νοημοσύνη*. Γ' Έκδοση. ISBN: 978-960-8396-64-7. Εκδόσεις Πανεπιστημίου Μακεδονίας.

Κατζουράκη, Μ. και Γεργατσούλης, Μ. και Κόκκοτος, Σ. (1991). *PROγραμματίζοντας στη LOGική (Με εφαρμογές στην Τεχνητή Νοημοσύνη)*, Εκδόσεις ΕΠΥ, Αθήνα.

Battani, G. and Meloni, H. (1973). *Interpreteur du Language de Programmation Prolog*. Groupe d' Intelligence Artificielle, Univ. d' Aix-Marseille, Luminy, France.

Bobrow, D.G. (1985). If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms. *IEEE trans. on Software Engineering*, Vol. 11, No. 11.

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Coelho, H. and Cotta, J.C. and Pereira, L.M. (1982). *How to Solve it with Prolog*. Laboratorio National de Engenharia Civil, Lisboa, Portugal.

Cohen, J. (1988). A view of the origins and development of Prolog. *Communications of the ACM*, Vol. 31, No.1.

Clocksin, W. F. and Mellish, C. S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

Colmerauer, A. and Kanoui, H. and Pasero, R. and Roussel, P. (1973). Une Systeme de Communication Homme-Machine en Francais. Research Report, Groupe d' Intelligence Artificielle, Univ. d' Aix-Marseille, Luminy, France.

Colmerauer, A. and Roussel, P. (1993). The birth of Prolog, *ACM SIGPLAN Notices* 28 (3):37.

Covington, M. A. and Nute, D. and Vellino A. (1988). *Prolog Programming in Depth*. Scott Foresman & Co. ISBN 9780673186591.

Covington, M. A. (1994). *Natural language processing for Prolog programmers*. Englewood Cliffs, N.J.: Prentice Hall.

Deransart, P. and Ed-Dbali, A. and Cervoni L. (1996). *Prolog: The Standard – Reference Manual*. Springer-Verlag

Ferrucci D. A. (2012). Introduction to "this is watson". *IBM Journal of Research and Development*, 56(3):235–249.

Hewitt C. (1969). PLANNER: A Language for Prooving Theorems in Robots. In Proceedings of IJCAI, Wanshington D.C., May 1969.

Kowalski, R. and Kuehner D. (1971). Linear Resolution with Selection Function. *Artificial Intelligence*, Vol. 2, pp. 227-60.

Kowalski, R. (1979). Logic for Problem Solving. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Kowalski, R. (1988). The early years of logic programming. *Communications of the ACM* 31:38.

Kowalski, R. (2013). Logic Programming in the 1970s. In: P. Cabalar and T.C. Son (eds.) LPNMR 2013. Springer Verlag.

Lloyd, J. W. (1984). *Foundations of logic programming*. Berlin: Springer-Verlag.

Moto-oka T. et al. (1981). *Challenge for Knowledge Information Processing Systems (preliminary report on Fifth Generation Computer Systems)*. In Proceedings of the International Conference on Fifth Generation Computer Systems (JIPDEC), North Holland.

O'Keefe, R. A. (1990), *The craft of Prolog*, Cambridge, Mass: MIT Press

Pereira, L.M. and Pereira, F. and Warren D.H.D. (1978). User's Guide to DECsystem-10 Prolog. University of Edinburg, Department of Artificial Intelligence.

Perlis, A. J. (1982). Epigrams on Programming, *ACM SIGPLAN Notices* Vol. 17, No. 9, p.p. 7-13

Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle. *Journal of ACM*. Vol. 12. pp. 23-41.

Shapiro, E. Y. (1983). The Fifth Generation Project - A Trip Report. *Communications of the ACM*, Vol. 26, No, 9.

Sterling, L. and Shapiro, E. Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass: MIT Press.

Sussman, G.J and Winograd, T. and Charniak, E. (1970). *MicroPLANNER Reference Manual.* AI Memo 203, Cambridge MA, MIT Artificial Intelligence Lab.

Tsadiras, A. (2009), *Using Prolog for Developing Real World Artificial Intelligence Applications*. Encyclopedia of Information Science and Technology, 2nd Edition, edited by Dr. Mehdi

Van Caneghem, M. (1986). *L'Anatomie de Prolog*. InterEditions, Paris

Warren, D.H.D. and Pereira, L. M. and Pereira F. (1977). *Prolog - the language and its implementation compared with Lisp*. ACM SIGART Bulletin archive, Issue 64. Proceedings of the 1977 symposium on Artificial intelligence and programming languages, pp 109 – 115.

Warren, D.H.D. (1983). *An abstract Prolog instruction set*. Technical Report 309, SRI International, Menlo Park, CA, October 1983.

ΚΕΦΑΛΑΙΟ 2: Δηλωτικός Προγραμματισμός

Λέξεις Κλειδιά
Διαδικαστικός προγραμματισμός, Δηλωτικός προγραμματισμός, Διαδικαστική γλώσσα προγραμματισμού, Προστακτική γλώσσα προγραμματισμού, Δηλωτική γλώσσα προγραμματισμού.

Περίληψη
Το κεφάλαιο ξεκινά με την παρουσίαση της διάσημης πλέον "εξίσωσης" του R. Kowalski που περιγράφει την έννοια του δηλωτικού προγραμματισμού. Εξηγεί πως οι γλώσσες που ανήκουν στην κατηγορία αυτή επιτρέπουν στον προγραμματιστή απλά να περιγράφει το προς επίλυση πρόβλημα ενώ οι μηχανισμοί εκτέλεσης να αναλαμβάνουν την επίλυσή του. Η αναπαράσταση του προβλήματος απαιτείται να μην εμφανίζει αμφισημίες και να μπορεί να έχει μια εκφραστική ικανότητα η οποία επιτρέπει την αποδοτική αποτύπωση όλων των πτυχών του προς επίλυση προβλήματος: ένας τέτοιος φορμαλισμός παρέχεται από την κατηγορηματική λογική και τις γλώσσες λογικού προγραμματισμού. Το κεφάλαιο παραθέτει απλά παραδείγματα που τονίζουν την παραπάνω προσέγγιση και αποτελούν έναυσμα για την εισαγωγή στην κατηγορηματική λογική, στο λογικό προγραμματισμό και στη γλώσσα Prolog που ακολουθεί.

Μαθησιακοί Στόχοι

Το κεφάλαιο αυτό αποτελεί έναν προπομπό για τον τρόπο με τον οποίο σκέφτεται κάποιος που χρησιμοποιεί το λογικό προγραμματισμό για την ανάπτυξη εφαρμογών. Με την ολοκλήρωσή του ο αναγνώστης θα γνωρίζει:

- τη διαφορά ανάμεσα στο διαδικαστικό και το δηλωτικό προγραμματισμό και των σχετικών γλωσσικών εννοιών και μηχανισμών που καθορίζουν αυτή τη διαφοροποίηση, και
- τη διπλή δυνατότητα ερμηνείας των προγραμμάτων του λογικού προγραμματισμού (διαδικαστική και δηλωτική).

Μονοπάτι Μάθησης

Το κεφάλαιο αυτό λειτουργώντας ως προπομπός, αναφέρεται σε έννοιες που θα αναλυθούν διεξοδικά σε επόμενα κεφάλαια του βιβλίου. Ο αναγνώστης μπορεί να επανέλθει για μία δεύτερη ανάγνωση του κεφαλαίου όταν θα έχει εξοικειωθεί με το λογικό προγραμματισμό και την Prolog και θα έχει δοκιμάσει αρκετά προγράμματα από αυτά που αναφέρονται στα επόμενα κεφάλαια.

2.1 Διαδικαστικός και Δηλωτικός Προγραμματισμός

Στις συμβατικές γλώσσες προγραμματισμού, όπως η Java και η C/C++, τα προγράμματα υλοποιούν αλγορίθμους, αναμιγνύοντας στον κώδικά τους τη λογική της λύσης του προβλήματος με τη διαδικασία ελέγχου της ροής εκτέλεσης του αλγορίθμου για την επίλυση του συγκεκριμένου προβλήματος. Συνυπάρχουν δηλαδή στον κώδικα του προγράμματος δύο βασικά στοιχεία:

- Η **λογική (logic)** του αλγορίθμου, που περιγράφει τι θέλουμε το πρόγραμμα να πετύχει: ποια είναι δηλαδή η λύση του προβλήματος.
- Ο **έλεγχος (control)**, που περιγράφει τη σειρά των βημάτων που πρέπει να ακολουθηθούν για να βρεθεί η λύση του προβλήματος.

Η συνύπαρξη των δύο αυτών στοιχείων για την υλοποίηση ενός αλγορίθμου, περιγράφεται στη συμβολική εξίσωση του R. Kowalski η οποία παρουσιάστηκε το 1979:

Αλγόριθμος = Λογική + Έλεγχος (Algorithm = Logic + Control)

Η εξίσωση αυτή ήρθε να συμπληρώσει μία άλλη πολύ γνωστή συμβολική εξίσωση που είχε εισάγει λίγα χρόνια πριν, το 1976 ο N. Wirth:

Αλγόριθμοι + Δομές Δεδομένων = Προγράμματα (Algorithms + Data Structures = Programs)

Για να κατανοήσουμε πώς συνδέεται η έννοια της λογικής ενός αλγορίθμου με τον έλεγχο της ροής εκτέλεσής του, ας δούμε για παράδειγμα τον κώδικα υλοποίησης του προβλήματος υπολογισμού του N παραγοντικό (N!) στη γλώσσα Java: Μία αφηρημένη μαθηματική περιγραφή του προβλήματος, που μπορεί να ορίσει τη λογική του αλγορίθμου είναι η εξής:

- Εάν N = 0 τότε N! = 1.
- Εάν N>0 τότε N! = 1x2x3x . . . x(N-1)xN.

Το παρακάτω τμήμα κώδικα σε Java υλοποιεί τον αλγόριθμο με τη βοήθεια της μεθόδου factorial:

```
int factorial(int N)throws wrongInput {
  if (N==0) return 1;
  int NP=1;
  if (N>0)
    for (int Counter=1; i<=N; i=i+1)
      NP=NP*Counter;
  else throw new wrongInput("Το παραγοντικό δεν ορίζεται");
    return NP;}
```

Ο παραπάνω κώδικας περιλαμβάνει τη σειρά των εντολών που πρέπει να εκτελεστούν για να υπολογιστεί το παραγοντικό ενός δοθέντος αριθμού. Η λογική του αλγορίθμου "κρύβεται" ανάμεσα στις εντολές ελέγχου της ροής του προγράμματος (επιλογής "if-then-else" και επανάληψης "for"). Ο αναγνώστης για να κατανοήσει τον κώδικα ενός προγράμματος, όπως το παραπάνω, πρέπει να "μιμηθεί" τη διαδικασία εκτέλεσης του από τη μηχανή, ελέγχοντας βήμα-βήμα τι σημαίνει. Για το λόγο αυτό, ο προγραμματισμός αυτός χαρακτηρίζεται με τον όρο "διαδικαστικός προγραμματισμός" (procedural programming). Επιπλέον, επειδή κάθε εντολή ερμηνεύεται ως μία διαταγή εκτέλεσης προς τη μηχανή, χρησιμοποιείται εναλλακτικά και ο όρος "διατακτικός" ή "προστακτικός" προγραμματισμός (imperative programming). Η πολυπλοκότητα ερμηνείας του κώδικα στις διαδικαστικές γλώσσες προγραμματισμού αυξάνει καθώς αυξάνει το μέγεθος του κώδικα.

Αντίθετα, όπως θα περιγράψουμε αναλυτικά στο Κεφάλαιο 6, ο κώδικας της Prolog που υλοποιεί το παραπάνω πρόβλημα του παραγοντικού μπορεί να οριστεί με τη βοήθεια του αναδρομικού ορισμού του ως εξής:

```
factorial(0,1).
factorial(N,F):-
  N>0,
  N1 is N-1,
  factorial(N1,F1),
  F is N*F1.
```

Στο παραπάνω πρόγραμμα απουσιάζουν παντελώς εντολές ελέγχου της ροής του προγράμματος και ο κώδικας της Prolog δεν κάνει τίποτε περισσότερο από το να αποδίδει τον αναδρομικό μαθηματικό ορισμό του N παραγοντικού:

- Εάν N = 0 τότε N! = 1
- Εάν N>0 τότε N! =N x (N-1)!

Με αυτό τον τρόπο δηλώνεται ο ορισμός του λογικού μόνο μέρους του προβλήματος και με την έννοια αυτή η προσέγγιση αυτή χαρακτηρίζεται ως "δηλωτικός προγραμματισμός" (declarative programming).

Με βάση τα παραπάνω, ως ένα πρώτο συμπέρασμα μπορούμε να αναφέρουμε ότι ο δηλωτικός προγραμματισμός, σε αντίθεση με τον διαδικαστικό, επικεντρώνεται στην περιγραφή του "Ποιο" είναι το πρόβλημα και όχι στο "Πώς" αυτό θα επιλυθεί. Προκύπτουν έτσι δύο βασικοί τρόποι να περιγράψουμε τη σημασιολογία μίας γλώσσας προγραμματισμού ή ενός προγράμματος γραμμένου σε μία γλώσσα. (α) Η δηλωτική σημασιολογία (declarative semantics): χαρακτηρίζουμε τη σημασία ενός προγράμματος με βάση

του “τι κάνει” και (β) η διαδικαστική σημασιολογία (procedural semantics): χαρακτηρίζουμε τη σημασία ενός προγράμματος με βάση του “πως το κάνει” ή ισοδύναμα του “τι γίνεται στη διάρκεια της εκτέλεσης του προγράμματος”.

Στη συνέχεια θα αναλύσουμε τα βασικά χαρακτηριστικά των μηχανισμών των γλωσσών προγραμματισμού που συντελούν σε αυτή τη διαφοροποίηση ανάμεσα στις δύο προσεγγίσεις.

2.2 Βασικά χαρακτηριστικά διαφοροποίησης του δηλωτικού από τον διαδικαστικό προγραμματισμό

Η προσπάθεια ανάπτυξης προγραμμάτων με τρόπο ώστε να εκφράζεται η λογική (logic) του υπολογισμού χωρίς να περιγράφεται ο έλεγχος της ροής του (control), διατρέχει όλη την ιστορία της εξέλιξης των γλωσσών προγραμματισμού. Έτσι, παραδείγματος χάριν, θα μπορούσε κανείς να συγκρίνει τις πρώτες γλώσσες προγραμματισμού με τη γλώσσα μηχανής ή τη συμβολική γλώσσα (Assembly). Η δυνατότητα του υπολογισμού μίας μαθηματικής έκφρασης με τη βοήθεια μίας εντολής αντικατάστασης στην οποία αυτή παρατίθεται ως έχει στο δεξιό της μέρος, π.χ.:

```
D = B*B-4*A*C  /* Υπολογισμός της διακρίνουσας μιας δευτεροβάθμιας
εξίσωσης*/
```

αποτέλεσε μία υψηλού αφαιρετικού επιπέδου περιγραφή του ποια είναι η προς επίλυση έκφραση, στην οποία αποκρύπτεται το πώς αυτή υπολογίζεται στο επίπεδο της περιγραφής του υπολογισμού της, με τη βοήθεια κώδικα γραμμένου σε γλώσσα μηχανής. Επίσης η εισαγωγή των υποπρογραμμάτων στις γλώσσες προγραμματισμού, ο δομημένος προγραμματισμός (structured programming) και στη συνέχεια ο αντικειμενοστρεφής προγραμματισμός (object-oriented programming) αποτέλεσαν τεχνικές ανάπτυξης προγραμμάτων με τις οποίες γίνεται προσπάθεια διαχωρισμού και απόκρυψης του κώδικα υλοποίησης από το τι κάνει η προγραμματιστική δομή που περιγράφει έναν υπολογισμό.

Η ποιοτική όμως διαφορά ανάμεσα στον διαδικαστικό προγραμματισμό και στον δηλωτικό προγραμματισμό είναι ότι στην πρώτη περίπτωση ο διαχωρισμός των λεπτομερειών υλοποίησης από το ποιο είναι το προς επίλυση πρόβλημα ή το μέρος ενός προβλήματος, ανατίθεται στον προγραμματιστή μέσω γλωσσικών μηχανισμών ενώ στη δεύτερη περίπτωση η διαδικασία αυτή αναλαμβάνεται από την ίδια τη γλώσσα, στο επίπεδο του μηχανισμού εκτέλεσης της (μέσω του μεταγλωττιστή ή διερμηνέα). Στο δηλωτικό Προγραμματισμό δηλαδή, ο προγραμματιστής δηλώνει τι ισχύει για το πρόβλημα (αντικείμενα, ιδιότητες, συσχετίσεις, περιορισμοί) και το “σύστημα”, το οποίο μπορεί να είναι είτε ένας μεταγλωττιστής (compiler) είτε ένας διερμηνέας (interpreter), αναλαμβάνει μόνο του να συνδυάσει τις δηλώσεις και να επιλύσει το πρόβλημα.

Δεν μπορεί να υπάρξει ένας αυστηρός ορισμός που να ορίσει με απόλυτο τρόπο το ποιες γλώσσες είναι γλώσσες δηλωτικού προγραμματισμού και ποιες όχι. Χαρακτηριστικές όμως κατηγορίες γλωσσών δηλωτικού προγραμματισμού αποτελούν οι γλώσσες λογικού προγραμματισμού, όπως η Prolog, οι γλώσσες συναρτησιακού προγραμματισμού, όπως η Lisp και η Haskell, και οι γλώσσες ερωταποκρίσεων βάσεων δεδομένων, όπως η SQL. Στο Λογικό Προγραμματισμό οι δηλώσεις είναι λογικές εκφράσεις και το “σύστημα” είναι ένας αποδείκτης θεωρημάτων (theorem prover). Στο Συναρτησιακό Προγραμματισμό οι δηλώσεις είναι ορισμοί συναρτήσεων και το “σύστημα” είναι μια μηχανή ροής δεδομένων (dataflow machine). Στην SQL ως “σύστημα” μπορεί να θεωρηθεί ο μηχανισμός βελτιστοποίησης των συστήματος διαχείρισης βάσεων δεδομένων που επεξεργάζεται τις εντολές της γλώσσας ερωτήσεων.

Τα ακόλουθα χαρακτηριστικά γλωσσών προγραμματισμού μπορούν να χρησιμοποιηθούν ως βασικά κριτήρια για την κατάταξη τους στην κατηγορία του δηλωτικού προγραμματισμού, καθώς και της διαφοροποίησής τους από τις προστακτικές γλώσσες προγραμματισμού.

Τρόπος ορισμού του προβλήματος

Όπως έχει αναλυθεί προηγουμένως, οι προστακτικές γλώσσες προγραμματισμού περιγράφουν τους υπολογισμούς που σχετίζονται με την επίλυση ενός προβλήματος με τη βοήθεια αυστηρών ακολουθιών

εντολών που επικεντρώνονται στο "πώς" αυτό θα επιλυθεί. Στην ακολουθία αυτών των εντολών εμπεριέχεται από κοινού τόσο το λογικό μέρος (logic) των αλγορίθμων που χρησιμοποιούνται όσο και η διαδικασία της ροής ελέγχου (control) τους.

Αντίθετα, στις δηλωτικές γλώσσες τα προγράμματα δομούνται ως σύνολα ορισμών (π.χ. σχέσεων στο λογικό προγραμματισμό ή συναρτήσεων στο συναρτησιακό προγραμματισμό) που προδιαγράφουν σε ένα υψηλό επίπεδο "τι" είναι αυτό που θα επιλυθεί, επικεντρώνοντας στο λογικό μέρος περιγραφής ενός προβλήματος.

Ο τρόπος ορισμού ενός προβλήματος στο δηλωτικό προγραμματισμό, πέραν του ότι συντελεί ο κώδικας που παράγεται να είναι πιο ευανάγνωστος, έχει ευεργετικά αποτελέσματα και στη μείωση του μεγέθους του. Έχει αποδειχθεί ότι το μέγεθος ενός κώδικα που παράγεται από κάποιο προγραμματιστή σε ένα συγκεκριμένο χρονικό διάστημα είναι πρακτικά ίδιο ανεξάρτητα από τη γλώσσα προγραμματισμού που χρησιμοποιείται Όπως θα δούμε στα επόμενα κεφάλαια μία γλώσσα λογικού προγραμματισμού, όπως η Prolog παρέχει το πλεονέκτημα του σύντομου κώδικα, με ευεργετικά αποτελέσματα στην ποιότητα του λογισμικού.

Ο ρόλος των μεταβλητών (και η εντολή αντικατάστασης)

Οι προστακτικές γλώσσες προγραμματισμού χρησιμοποιούν ευρύτατα τις μεταβλητές οι οποίες αναπαριστάνονται με κάποιο όνομα και αντιστοιχίζονται με τιμές οι οποίες αλλάζουν κατά τη διάρκεια της εκτέλεσης του προγράμματος στα πλαίσια των εκφράσεων που εμφανίζονται. Συνήθως μία έκφραση εμφανίζεται στο δεξιό μέρος μίας εντολής καταχώρησης ή αντικατάστασης (assignment statement) και αφού υπολογιστεί, η τιμή της αλλάζει την τιμή της μεταβλητής που βρίσκεται στο αριστερό μέρος. Η διαδικασία αυτή είναι συνεχής κατά τη διάρκεια εκτέλεσης ενός προγράμματος, κατά την οποία κάθε νέα ανάθεση διαγράφει την παλιά τιμή της μεταβλητής βάζοντας μία νέα στη θέση της. Η διαδικασία αυτή αναφέρεται ως καταστροφική καταχώρηση (destructive assignment) κάνοντας την έννοια της μεταβλητής να μην έχει καμία σχέση με αυτήν που χρησιμοποιούμε στα μαθηματικά. Είναι χαρακτηριστικό το παράδειγμα της εντολής καταχώρησης:

```
X=X+1; ή X++;
```

η οποία χρησιμοποιείται για να δημιουργηθεί η επόμενη μιας τιμής και να αντιστοιχηθεί με το όνομα μιας μεταβλητής.

Αντίθετα, στις δηλωτικές γλώσσες (όπως είναι οι καθαρά συναρτησιακές γλώσσες και οι γλώσσες λογικού προγραμματισμού), σε κάθε μεταβλητή μπορεί να καταχωρηθεί μία και μόνο μία τιμή, η οποία δεν μπορεί να αλλάξει κατά τη διάρκεια της εκτέλεσης. Η διαδικασία αυτή χαρακτηρίζεται ως μη-καταστροφική καταχώρηση (non-destructive assignment). Αν θέλει κανείς να χρησιμοποιήσει την επόμενη τιμή μιας μεταβλητής δεν έχει παρά να τη συσχετίσει με το όνομα μίας νέας μεταβλητής.

Η εντολή αντικατάστασης / καταχώρησης αποκαλύπτει μεταξύ άλλων και το πρόβλημα της έλλειψης διαφάνειας αναφοράς (referential transparency): Μία μεταβλητή μέσω της εντολής αντικατάστασης μπορεί να πάρει διαφορετικές τιμές σε διαφορά σημεία του κώδικα του προγράμματος, χάνοντας με αυτό τον τρόπο τις παλιές τιμές. Ας υποθέσουμε ότι έχουμε το παρακάτω τμήμα κώδικα:

```
X = A + B;
... //εντολές που μεσολαβούν
Y = Z + A + B;
```

Στο ερώτημα αν θα μπορούσαμε να αντικαταστήσουμε το A+B της δεύτερης εντολής αντικατάστασης με το X (όπως θα κάναμε αν λύναμε ένα σύστημα εξισώσεων στην άλγεβρα) η απάντηση είναι σαφώς αρνητική, καθώς θα μπορούσε να συμβεί ένα από τα ακόλουθα: (α) να έχουν αλλάξει οι τιμές των μεταβλητών X, A και B από κάποια ενδιάμεση εντολή ή κλήση διαδικασίας, (β) η εντολή X = A+B να μην είχε εκτελεστεί, καθώς μπορεί να βρισκόταν σε μία δομή επιλογής "if", σε κλάδο που δεν είχε εκτελεστεί. Ως αποτέλεσμα το τι σημαίνουν αυτές οι μεταβλητές εξαρτάται από το σημείο στο οποίο βρίσκονται κάθε φορά, ή με άλλα λόγια αν για κάποια μεταβλητή ρωτήσουμε αυτή σε τι αναφέρεται ή τι τιμή έχει η απάντηση δεν μπορεί να δοθεί χωρίς να εξετάσουμε τι θα συμβεί κατά τη διάρκεια της εκτέλεσης του προγράμματος σε όλο το πεδίο δράσης της. Η έλλειψη διαφάνειας αναφοράς καθιστά πολύ δύσκολη την τροποποίηση και την απόδειξη της ορθότητας ενός προγράμματος προστατικής γλώσσας.

Το μοντέλο διαχείρισης της μνήμης

Ο κώδικας ενός προγράμματος γραμμένου σε μία προστακτική γλώσσα προγραμματισμού παραπέμπει στο αρχικό μοντέλο εκτέλεσης που βασίζεται στη Von Neuman αρχιτεκτονική ενός υπολογιστή. Σύμφωνα με αυτό το μοντέλο η κεντρική μονάδα επεξεργασίας εκτελεί τις εντολές του προγράμματος ανακτώντας και αποθηκεύοντας συνεχώς δεδομένα από και προς μία μνήμη ανάγνωσης - εγγραφής. Το πρόγραμμα δηλαδή για να μεταφέρει δεδομένα από ένα τμήμα του σε ένα άλλο χρησιμοποιεί τη μνήμη ως μεσάζοντα. Αυτός ο τρόπος της μεταφοράς δεδομένων δημιουργεί πολλές φορές παρενέργειες (side effects) και εναπόκειται στον προγραμματιστή να φροντίσει ένα τμήμα προγράμματος να μην τροποποιήσει δεδομένα της μνήμης που χρησιμοποιεί ένα άλλο τμήμα του προγράμματος. Η εισαγωγή μηχανισμών για τον ορισμό και τη χρήση δομών δεδομένων στις προστακτικές γλώσσες προγραμματισμού, παρόλο που έχει περιορίσει τις πιθανότητες των παρενεργειών, δεν έχει αλλάξει το μοντέλο εκτέλεσης που θεωρεί τη μνήμη ως ένα απαραίτητο ενδιάμεσο χώρο για τη μεταφορά δεδομένων. Έτσι, παραδείγματος χάριν, ένας αλγόριθμος ταξινόμησης ενός πίνακα αναδιατάσσει τα δεδομένα του και με την ολοκλήρωση της εκτέλεσής του η αρχική μορφή τους έχει χαθεί. Αν η χρήση των αρχικών δεδομένων χρειάζεται εκ νέου, μετά την ολοκλήρωση της ταξινόμησης είναι στην ευθύνη του προγραμματιστή να αντιγράψει τα αρχικά στοιχεία του πίνακα σε έναν άλλο, να τα αποθηκεύσει δηλαδή στην ουσία σε μία άλλη περιοχή της μνήμης.

Οι δηλωτικές γλώσσες προγραμματισμού αποφεύγουν το πρόβλημα των παρενεργειών καθώς δεν χρησιμοποιούν ρητά τη μνήμη ως έναν ενδιάμεσο χώρο αποθήκευσης δεδομένων. Σε ένα πρόγραμμα δηλωτικής γλώσσας ο μόνος τόπος μεταφοράς δεδομένων είναι με πέρασμα παραμέτρων. Στο συναρτησιακό προγραμματισμό το αποτέλεσμα μίας συνάρτησης μπορεί να μεταβιβαστεί σε μία δεύτερη εάν η κλήση της πρώτης έχει τοποθετηθεί ως παράμετρος σε ένα από τα ορίσματα της δεύτερης. Στο λογικό προγραμματισμό η τιμή μιας μεταβλητής μπορεί να περάσει σε μία άλλη είτε μέσω της ενοποίησης, που αποτελεί το μηχανισμό περάσματος παραμέτρων, είτε γιατί η ίδια μεταβλητή διαμοιράζεται σε δύο κλήσεις διαφορετικών κατηγορημάτων, όπως συμβαίνει με τη μεταβλητή X ανάμεσα στα κατηγορήματα p1 και p2 στο παρακάτω παράδειγμα:

```
p(X,Y,Z,T) :- p1(X,Y,Z), . . ., p2(T,X).
```

Τοπικότητα / Πεδίο Δράσης των μεταβλητών

Στις προστακτικές γλώσσες προγραμματισμού η οποιαδήποτε διαδικασία υπολογισμού πραγματοποιείται μέσω της εκτέλεσης μίας ακολουθίας εντολών στο πλαίσιο ενός πεδίου δράσης (scope) στο οποίο είναι γνωστές τόσο οι μεταβλητές όσο και οι άλλες οντότητες της γλώσσας. Το πεδίο δράσης είναι συνήθως ευρύ και μπορεί να είναι είτε όλο το πρόγραμμα είτε κάποιο από τα υποπρογράμματά του. Κάθε εντολή αναπαριστά ένα μόνο βήμα ενός αλγορίθμου, το οποίο εκτελείται στο πλαίσιο αυτού του πεδίου δράσης. Ως αποτέλεσμα η ορθότητα μίας μεμονωμένης εντολής δεν μπορεί ελεγχθεί αφεαυτής, καθώς πρέπει να εξεταστεί η συμπεριφορά ολόκληρου του αλγόριθμου μέσα στον οποίο βρίσκεται. Στις δηλωτικές γλώσσες προγραμματισμού το πεδίο δράσης των μεταβλητών περιορίζεται δραστικά σε μικρότερες υπολογιστικές οντότητες. Όπως θα δούμε, στην Prolog το πεδίο δράσης περιορίζεται στο πλαίσιο κάθε μεμονωμένης φράσης του προγράμματος με ευεργετικά αποτελέσματα στον έλεγχο της ορθότητας της.

2.3 Η διπλή ερμηνεία των λογικών προγραμμάτων

Ένας από τους λόγους που η Prolog αποτέλεσε μία επιτυχημένη γλώσσα προγραμματισμού είναι η δυνατότητα ερμηνείας των φράσεων Horn του λογικού προγραμματισμού, οι οποίες μπορούν να ερμηνεύονται τόσο δηλωτικά όσο και διαδικαστικά. Ας θεωρήσουμε μία φράση-κανόνα του λογικού προγραμματισμού γραμμένη στην αφαιρετική της μορφή:

$$H :- B_1, B_2, ..., B_n$$

Με βάση τη δηλωτική σημασιολογία η παραπάνω φράση ερμηνεύεται ως εξής: Εφόσον το B_1 είναι αληθές και το B_2 είναι αληθές και ... και το B_n είναι αληθές τότε συνεπάγεται ότι και το Η είναι αληθές (το “,” στην παραπάνω φράση ερμηνεύεται ως ο λογικός τελεστής ΚΑΙ/AND).

Ταυτόχρονα όμως η παραπάνω φράση μπορεί να ερμηνευτεί και διαδικαστικά. Τα Η, B_1, B_2, . . . και B_n, ερμηνεύονται ως τα υποπρογράμματα-διαδικασίες (procedures) της γλώσσας, όπου το Η θεωρείται η επικεφαλίδα της διαδικασίας και B_1, B_2, . . . , B_n αποτελούν το σώμα της, δηλαδή τον κώδικα που την υλοποιεί. Η εκτέλεση του προγράμματος ξεκινά με την ερώτηση:

```
?- H.
```

που ερμηνεύεται ως η κλήση του υποπρογράμματος Η. Για να ολοκληρωθεί η εκτέλεση του υποπρογράμματος Η εκτελείται ο κώδικας που το υλοποιεί δηλαδή καλούνται οι διαδικασίες B_1, B_2, . . . και B_n:

```
?- B1, B2, ...,Bn.
```

Στην περίπτωση αυτή το "," ερμηνεύεται κατά κάποιο τρόπο ως ο διαχωριστής των κλήσεων με τρόπο αντίστοιχο (αλλά όχι και ισοδύναμο όπως θα δούμε στα επόμενα κεφάλαια) που ερμηνεύεται ο διαχωριστής ";" σε διαδικαστικές γλώσσες προγραμματισμού όπως η C++ και η Java.

2.4 Παρατηρήσεις

Στον "ιδανικό" δηλωτικό προγραμματισμό με τη βοήθεια της Prolog στο πρόγραμμα εμφανίζεται αποκλειστικά το λογικό μέρος του και ο έλεγχος του προγράμματος προκύπτει αυτόματα κατά τη διαδικασία εκτέλεσης από το μηχανισμό της γλώσσας. Η περίπτωση αυτή αναφέρεται ως προγραμματισμός με "καθαρή" (pure) Prolog. Όπως θα δούμε στη συνέχεια, για πρακτικούς λόγους, έχουν προστεθεί στην Prolog μηχανισμοί που μπορούν να καθορίσουν τη διαδικασία ελέγχου της ροής του προγράμματος. Ο προγραμματιστής που θα ασχοληθεί συστηματικά με την ανάπτυξη λογισμικού σε Prolog καλό είναι να γνωρίζει αυτή τη διαφοροποίηση και να χρησιμοποιεί αυτούς τους μηχανισμούς με προσοχή και μόνον όταν είναι απαραίτητο. Για να επανέλθουμε στην εξίσωση του Kowalski (Αλγόριθμος = Λογική + Έλεγχος), καλό είναι ο προγραμματιστής, όταν απαιτηθεί να επέμβει στη διαδικασία του ελέγχου του προγράμματος να το κάνει σε τμήμα κώδικα που είναι ανεξάρτητο από το τμήμα που ορίζει τη λογική του αλγορίθμου επίλυσης του προβλήματος.

Ένα βασικό πρόβλημα που αντιμετωπίζουν όσοι προγραμματιστές έρχονται πρώτη φορά σε επαφή με τον δηλωτικό προγραμματισμό, είναι ότι έχουν "εθιστεί" στην ανάπτυξη κώδικα που περιγράφει με λεπτομέρεια το πώς εκτελείται ένα πρόγραμμα. Το γεγονός αυτό τους κάνει να αισθάνονται χαρούμενοι - ακόμα και περήφανοι που έχουν την τεχνική ικανότητα και την ευελιξία να ελέγχουν με τον κώδικα που παράγουν τις λεπτομέρειες εκτέλεσης ενός προγράμματος και δεν είναι διατεθειμένοι να αφήσουν αυτή τη διαδικασία σε μία μαγική ή κρυφή διαδικασία που θα προκύψει από τον ίδιο μηχανισμό της γλώσσας.

Βιβλιογραφία

Η αναλυτική περιγραφή του διαχωρισμού ενός αλγορίθμου στο λογικό μέρος του και στο τμήμα που περιγράφει τον έλεγχο της εκτέλεσης του μπορεί να βρεθεί στην ομώνυμη εργασία του Robert Kowalski (Kowalski, 1979). Ο αναγνώστης που θέλει να συγκρίνει τον διαχωρισμό αυτό με την κλασσική προσέγγιση ανάπτυξης αλγορίθμων των διαδικαστικών γλωσσών προγραμματισμού μπορεί να διαβάσει το βιβλίο αναφοράς του Niklaus Wirth (Wirth, 1971). Μία πολύ καλή ανάλυση των εννοιών και των μηχανισμών γλωσσών που σχετίζονται με το δηλωτικό προγραμματισμό και της σύγκρισής τους με τις αντίστοιχες έννοιες των διαδικαστικών γλωσσών μπορεί να βρεθεί στο βιβλίο των Peter Van Roy και Seif Haridi (Van Roy and Haridi, 2004). Εκτενής ανάλυση τόσο της δηλωτικής όσο και της διαδικαστικής ερμηνείας του Λογικού προγραμματισμού μπορεί να βρεθεί στο βιβλίο του Robert Kowalski (Kowalski, 1979b), καθώς και στην εργασία των Van Emden και Kowalski (Van Emden and Kowalski 1976).

Kowalski, R. (1979). Algorithm = Logic + control, *Communications of the ACM*, Vol. *22*, No. 7, p.p. 424-436.

Kowalski, R. (1979b). *Logic for Problem Solving*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Van Roy, P. and Haridi, S. (2004). *Concepts, Techniques, and Models of Computer Programming*, The MIT Press

Van Emden M. H. and Kowalski R. (1976) The Semantics of Predicate Logic as a Programming Language, *Journal of the ACM (JACM)*, Vol *23*, No 4, p.p. 733-742

Wirth N. (1976). *Algorithms + Data Structures = Program*s. Prentice-Hall Series in Automatic Computation.

ΚΕΦΑΛΑΙΟ 3: Κατηγορηματική Λογική Πρώτης Τάξεως και Λογικά Προγράμματα

Λέξεις Κλειδιά
Μαθηματική Λογική, Προτασιακή Λογική, Κατηγορηματική Λογική, Προτάσεις Horn, Λογικά Προγράμματα

Περίληψη
Το κεφάλαιο αποτελεί μια σύντομη εισαγωγή στην Μαθηματική Λογική και ειδικότερα στην κατηγορηματική λογική πρώτης τάξης, η οποία αποτελεί την μαθηματική βάση του Λογικού Προγραμματισμού. Συγκεκριμένα στο κεφάλαιο αυτό παρουσιάζονται τα βασικά στοιχεία της Λογικής, όπως είναι η σύνταξη των όρων της κατηγορηματικής λογικής, οι τελεστές και οι αποδεικτικές διαδικασίες. Δίνεται έμφαση στην αρχή της ανάλυσης (resolution principle) και αναδεικνύεται η σημαντικότητά της για την αυτοματοποιημένη επίλυση προβλημάτων, προκειμένου η μαθηματική λογική να μπορεί να χρησιμοποιηθεί ως γλώσσα προγραμματισμού. Τέλος, το κεφάλαιο κλείνει με την παρουσίαση των κανονικών μορφών της Λογικής, κατά Kowalski και Horn, που χρησιμοποιούνται στην γλώσσα Prolog.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός:

- Να κατανοεί τις έννοιες της μαθηματικής λογικής, γενικότερα, καθώς και των δύο μορφών της που θα αναπτυχθούν στο κεφάλαιο, της προτασιακής και της κατηγορηματικής λογικής.
- Να μπορεί να διατυπώνει αξιωματικές προτάσεις στην προτασιακή και κατηγορηματική λογική χρησιμοποιώντας τους λογικούς τελεστές και τους ποσοδείκτες.
- Να μπορεί να αποδεικνύει προτάσεις - θεωρήματα στην προτασιακή και κατηγορηματική λογική χρησιμοποιώντας τους κανόνες συμπερασμού της κάθε λογικής.
- Να κατανοεί τους μετασχηματισμούς που μπορεί να υποστεί ένα σύνολο λογικών προτάσεων προκειμένου να μετατραπεί σε ένα σύνολο προτάσεων Horn στην μορφή Kowalski και να αποτελέσει λογικό πρόγραμμα.

Παράδειγμα κίνητρο

Οι άνθρωποι σκεφτόμαστε με βάση τις πεποιθήσεις μας ή την γνώση μας για τον κόσμο, γενικότερα, καθώς και την αντίληψη μας (που προέρχεται από τα αισθητήρια όργανα) για την τρέχουσα κατάσταση του περιβάλλοντος και είτε προσπαθούμε να καταλάβουμε καλύτερα το περιβάλλον, για παράδειγμα τι άλλο μπορεί να συμβαίνει χωρίς να το παρατηρούμε άμεσα, ή να προβλέψουμε τι μπορεί να συμβεί αργότερα, προκειμένου να είμαστε προετοιμασμένοι να το αντιμετωπίσουμε.

Για παράδειγμα, ακούω (παρατήρηση μέσω αισθητηρίων οργάνων) ότι βροντάει, χωρίς να είμαι σε θέση να δω αν βρέχει. Όμως, γνωρίζω (πεποίθηση για τον κόσμο) ότι όποτε βροντάει, βρέχει επίσης. Άρα συνδυάζοντας τα παραπάνω, συμπεραίνω ότι αυτήν την στιγμή βρέχει (καλύτερη κατανόηση του περιβάλλοντος). Όταν βρέχει παίρνω ομπρέλα. Άρα θα πάρω και σε λίγο που θα βγω έξω (προετοιμασία για μελλοντική δράση).

Το εργαλείο της σκέψης που χρησιμοποιούμε για τα παραπάνω ονομάζεται συλλογιστική και η λογική είναι η επιστήμη που αναλύει και μελετάει τις μεθόδους με τις οποίες συλλογιζόμαστε, ανεξάρτητα από το περιεχόμενο των συλλογισμών. Για παράδειγμα, αν όποτε έχει ήλιο παίρνω καπέλο και ξέρω ότι τώρα έχει ήλιο, τότε συμπεραίνω ότι πρέπει να πάρω καπέλο. Ο συλλογισμός αυτός μοιάζει με τον προηγούμενο, παρόλο που το περιεχόμενό τους είναι διαφορετικό.

Στην συμβολική λογική το περιεχόμενο των συλλογισμών αντικαθίσταται από γενικά - αφαιρετικά σύμβολα, προκειμένου να μελετηθεί η μορφή του συλλογισμού ευκολότερα. Για παράδειγμα και οι δύο προηγούμενοι συλλογισμοί μπορούν να αναπαρασταθούν, χρησιμοποιώντας σύμβολα λογικών πράξεων, ως εξής: από το P και το EAN P TOTE Q, συμπεραίνω ότι ισχύει το Q.

Η παραπάνω αναπαράσταση των συλλογισμών θεωρεί ότι κάθε σκέψη (λογική πρόταση) μπορεί να αναπαρασταθεί από ένα σύμβολο, το οποίο όμως δεν έχει εσωτερική δομή. Η συγκεκριμένη μορφή της λογικής ονομάζεται προτασιακή. Τι συμβαίνει όμως όταν θέλουμε να μελετήσουμε το εξής: εχθές έβρεχε ενώ προχθές δεν έβρεχε. Αν κάποιος προσπαθεί να αποφανθεί για τον αν πήρα ομπρέλα εχθές ή προχθές, θα πρέπει ουσιαστικά να χρησιμοποιήσει τον παραπάνω συλλογισμό για δύο διαφορετικά χρονικά στιγμιότυπα και να μπορεί να ξεχωρίσει και τα δεδομένα (βρέχει) και τα συμπεράσματα (πήρε ομπρέλα) για τις δύο αυτές χρονικές στιγμές. Για να γίνει αυτό θα πρέπει οι προτάσεις να έχουν ως παράμετρο τον χρόνο για να ξεχωρίζουν μεταξύ τους. Έτσι, ο συλλογισμός πλέον αναπαρίσταται ως εξής: Για οποιαδήποτε χρονική στιγμή X, αν βρέχει την στιγμή X, τότε παίρνω ομπρέλα την στιγμή X. Την χρονική στιγμή "εχθές", έβρεχε. Άρα συμπεραίνω ότι την χρονική στιγμή "εχθές", πήρα την ομπρέλα μου.

Η προηγούμενη επέκταση των προτάσεων με παραμέτρους ονομάζεται κατηγορηματική λογική και είναι εντελώς απαραίτητη για να μπορούμε να εκφράσουμε γενικές απόψεις για τον κόσμο, ανεξάρτητες από συγκεκριμένες χρονικές στιγμές, πρόσωπα ή καταστάσεις. Στην συνέχεια, οι γενικές αυτές απόψεις εφαρμόζονται σε συγκεκριμένες χρονικές στιγμές, πρόσωπα ή καταστάσεις και καταλήγουμε σε συγκεκριμένα συμπεράσματα. Στην συμβολική μορφή του κατηγορηματικού λογισμού, ο παραπάνω συλλογισμός εκφράζεται ως εξής: ∀X βρέχει(X)→ομπρέλα(X) και βρέχει(εχθές), άρα ομπρέλα(εχθές), όπου το σύμβολο ∀ σημαίνει για οποιαδήποτε τιμή της μεταβλητής X, δηλαδή για οποιαδήποτε χρονική στιγμή X.

Τι συμβαίνει όμως να πρέπει να συνδυαστούν μεταξύ τους πολλές τέτοιες λογικές προτάσεις και παρατηρήσεις για τον κόσμο έτσι ώστε να παραχθούν χρήσιμα συμπεράσματα; Για παράδειγμα, αν γνωρίζω ότι εχθές βροντούσε (βροντές(εχθές)) και συνδυάζοντας δύο συνεπαγωγές ∀X βροντές(X)→βρέχει(X) και ∀X βρέχει(X)→ομπρέλα(X), συμπεραίνω ότι βρέχει(εχθές) και ομπρέλα(εχθές). Τίθεται δηλαδή θέμα εφαρμογής μιας ακολουθιακής σειράς συνεπαγωγών, οι οποίες μπορούν να διακλαδώνονται ή να συναντώνται σε διάφορα στάδια της διαδικασίας. Προκειμένου να μπορεί να αυτοματοποιηθεί η διαδικασία εξαγωγής συμπερασμάτων θα πρέπει να τυποποιηθεί η δυνατότητα έκφρασης λογικών προτάσεων, τα επιμέρους βήματα της αποδεικτικής διαδικασίας αλλά και ο έλεγχος της εκτέλεσης αυτών των βημάτων. Με άλλα λόγια θα είναι εφικτός ο προγραμματισμός της απόδειξης έτσι ώστε να καταλήξει στα επιθυμητά συμπεράσματα. Ο λογικός προγραμματισμός ουσιαστικά γεφυρώνει την μαθηματική λογική με την αλγοριθμική εκτέλεση υπολογισμών, αντιμετωπίζοντας την δεύτερη ως μια διαδικασία απόδειξης λογικών θεωρημάτων βασισμένη στην πρώτη.

Μονοπάτι Μάθησης

Το κεφάλαιο αυτό αποτελεί μία σύντομη αλλά πλήρη εισαγωγή στις έννοιες της μαθηματικής λογικής αλλά και των υποσυνόλων της που οδηγούν στην λεγόμενη "**υπολογιστική λογική**" (**computational logic**), με άλλα λόγια στο υποσύνολο της μαθηματικής λογικής, το οποίο μπορεί να χρησιμοποιηθεί ως γλώσσα προγραμματισμού. Ανάλογα με το πόση έμφαση θέλει κάποιος να δοθεί στην λογική, ως γενική μέθοδο αναπαράστασης και επίλυσης προβλημάτων, το κεφάλαιο μπορεί να χρησιμοποιηθεί είτε εξ ολοκλήρου, είτε κάποια κομμάτια του μπορεί να παραληφθούν σε μια πρώτη ανάγνωση. Για παράδειγμα, αν κάποιος θέλει απλά να κατανοήσει πώς οδηγούμαστε από την λογική στον λογικό προγραμματισμό, οι απολύτως απαραίτητες ενότητες είναι:

- η εισαγωγή,
- η προτασιακή λογική, δίνοντας έμφαση περισσότερο στις κανονικές μορφές, τις ισοδυναμίες και στην λογική απόδειξη, και όχι τόσο στους πίνακες αλήθειας, ταυτολογίες, κλπ.,
- η κατηγορηματική λογική, δίνοντας έμφαση περισσότερο στις κανονικές μορφές, πλην της Skolem, και στους μηχανισμούς εξαγωγής συμπερασμάτων, την αντικατάσταση και την ενοποίηση, και όχι

τόσο στον τρόπο με τον οποίο εκφράζουμε την γνώση σε κατηγορηματική λογική με την επιλογή κατάλληλων ποσοδεικτών, κλπ.

3.1 Εισαγωγή

Η λογική παρέχει έναν τρόπο για την αποσαφήνιση και την τυποποίηση της διαδικασίας της ανθρώπινης σκέψης και προσφέρει μια σημαντική και εύχρηστη μεθοδολογία για την αναπαράσταση και επίλυση προβλημάτων. Η ανάγκη χρήσης μιας αυστηρά ορισμένης γλώσσας, με τη μαθηματική έννοια, προήλθε από την ακαταλληλότητα της φυσικής γλώσσας για χρήση σε υπολογιστικά συστήματα. Αντίθετα, η λογική προσφέρει μια σαφή, ακριβή και απλή στη σύνταξη γλώσσα, καθώς και τη δυνατότητα παραγωγής νέας γνώσης από την ήδη υπάρχουσα.

Η *μαθηματική λογική* (*mathematical logic*) είναι η συστηματική μελέτη των *έγκυρων ισχυρισμών* (*valid arguments*) ή *συλλογισμών*, όπως αναφέρθηκε στην αρχή του κεφαλαίου, με χρήση εννοιών από τα μαθηματικά. Ένας *ισχυρισμός* (*argument*) αποτελείται από συγκεκριμένες *δηλώσεις* (ή *προτάσεις*), τις *υποθέσεις* (*premises*), από τις οποίες παράγονται άλλες δηλώσεις, τα *συμπεράσματα* (*conclusions*). Για παράδειγμα, παρακάτω φαίνεται ένας ισχυρισμός εκφρασμένος στην λογική:

Όποτε βρέχει παίρνω ομπρέλα,	*(Δήλωση)*
Σήμερα βρέχει,	*(Δήλωση)*
επομένως, Σήμερα θα πάρω ομπρέλα	*(Συμπέρασμα)*

Στη **συμβολική λογική** (**symbolic logic**) χρησιμοποιούνται σύμβολα αντί λέξεων της φυσικής γλώσσας προκειμένου να μελετηθούν οι ισχυρισμοί ανεξάρτητα από το περιεχόμενο των λογικών προτάσεων, εκφράζοντάς τους στη συμβολική τους μορφή. Έτσι ο προηγούμενος ισχυρισμός σε συμβολική λογική αναπαρίσταται ως:

P:	$\forall X$ *βρέχει(X)*$\rightarrow$*ομπρέλα(X)*
Q:	*βρέχει(σήμερα)*
R:	*ομπρέλα(σήμερα)*
$P \wedge Q$	$\models R$

Η "γλώσσα" της Λογικής, όπως και κάθε άλλη γλώσσα, είτε φυσική είτε τεχνητή, απαιτεί τον ορισμό της σύνταξης (syntax) και της σημασιολογίας (semantics) της. Η σύνταξη καθορίζει τις επιτρεπτές ακολουθίες συμβόλων και η σημασιολογία καθορίζει τις μεταξύ τους σχέσεις. Η σύνταξη προϋποθέτει τον καθορισμό του αλφάβητου της γλώσσας, δηλαδή του συνόλου των συμβόλων με τα οποία μπορεί να κατασκευαστούν αποδεκτές ακολουθίες (προτάσεις). Η ερμηνεία αντιστοιχεί τα σύμβολα της γλώσσας στις οντότητες του κόσμου που αναπαρίσταται και επιτρέπει την απόδοση λογικών τιμών στις προτάσεις της γλώσσας, δηλαδή το χαρακτηρισμό τους ως αληθείς ή ψευδείς.

Στις ενότητες που ακολουθούν δίνεται μια συνοπτική περιγραφή της λογικής που χρησιμοποιείται για τον λογικό προγραμματισμό, ξεκινώντας με την απλούστερη μορφή λογικής, την προτασιακή, προκειμένου να γίνουν κατανοητές οι βασικές έννοιες και μέθοδοι της λογικής, συνεχίζοντας με την κατηγορηματική λογική πρώτης τάξης, η οποία αποτελεί την πιο πλήρη μορφή της λογικής και καταλήγοντας στις προτάσεις Horn, οι οποίες αποτελούν το υποσύνολο της κατηγορηματικής λογικής, πάνω στο οποίο βασίζεται ο λογικός προγραμματισμός και η γλώσσα Prolog.

3.2 Προτασιακή Λογική

Η *προτασιακή λογική* (*propositional logic*) αποτελεί την απλούστερη μορφή μαθηματικής λογικής και η παρουσίασή της κρίνεται αναγκαία, προκειμένου να γίνουν κατανοητές οι βασικές έννοιες και μέθοδοι της λογικής. Στην προτασιακή λογική κάθε γεγονός του πραγματικού κόσμου αναπαριστάται με μια λογική πρόταση, η οποία χαρακτηρίζεται είτε ως *αληθής* (*T - true*) ή ως *ψευδής* (*F - false*), μπορεί δηλαδή να έχει δύο

λογικές τιμές. Οι λογικές προτάσεις αναπαριστώνται συνήθως από λατινικούς χαρακτήρες *P*, *Q*, *R*, κτλ., και ονομάζονται *άτομα* (*atoms*). Τα άτομα μπορούν να συνδυαστούν με τη χρήση *λογικών συμβόλων* ή *συνδετικών* (*connectives*) και οι σύνθετες προτάσεις που προκύπτουν ονομάζονται *ορθά δομημένοι τύποι* (*well formed formulae*). Ο Πίνακας 3.1 παρουσιάζει τα συνδετικά της προτασιακής λογικής. Θα πρέπει να σημειωθεί ότι η σύνταξη περιλαμβάνει και σημεία στίξης, όπως παρενθέσεις "(" και ")" και κόμμα ",".

Πίνακας 3.1: *Συνδετικά της προτασιακής λογικής*

Σύμβολο	**Ονομασία / Επεξήγηση**
∧	σύζευξη (λογικό “ΚΑΙ” / “AND”)
∨	διάζευξη (λογικό “Η” / “OR”)
¬	άρνηση (λογικό “ΟΧΙ” / “NOT”)
→	συνεπαγωγή (“ΕΑΝ ΤΟΤΕ” / “IF THEN”)
↔	ισοδυναμία (“ΑΝ ΚΑΙ ΜΟΝΟ ΑΝ” / “IF AND ONLY IF”)

Για παράδειγμα, έστω ότι απαιτείται η αναπαράσταση της ακόλουθης γνώσης με προτασιακή λογική:

1η Πρόταση: *"βρέχει"*

2η Πρόταση: *"παίρνω ομπρέλα"*

3η Πρόταση: *"εάν βρέχει, τότε παίρνω ομπρέλα"*

Σε κάθε πρόταση που θέλουμε να αναπαραστήσουμε, αντιστοιχεί ένας λατινικός χαρακτήρας. Για παράδειγμα:

P: *"βρέχει"*

Q: *"παίρνω ομπρέλα"*

Η 3η πρόταση αναπαριστάται με τη χρήση του συνδετικού της συνεπαγωγής και της 1ης και 2ης πρότασης, όπως φαίνεται παρακάτω:

$P \rightarrow Q$: *"εάν βρέχει, τότε παίρνω ομπρέλα"*

Η *σημασιολογία* της προτασιακής λογικής αντιστοιχεί μία τιμή αληθείας (αληθές ή ψευδές, ***T*** ή ***F*** αντίστοιχα) σε μια οποιαδήποτε απλή ή σύνθετη έκφραση - πρόταση, βασισμένη σε μια ερμηνεία της γλώσσας. Μια *ερμηνεία* (*interpretation*) *I* αντιστοιχεί τιμές αληθείας στις απλές προτάσεις (άτομα) και επεκτείνεται σε σύνθετες εκφράσεις - προτάσεις με χρήση ενός *πίνακα αληθείας* (*truth table*) για να μεταχειριστεί τους λογικούς συνδέσμους. Στον Πίνακα 3.2 φαίνεται ο πίνακας αλήθειας των συνδετικών της προτασιακής λογικής.

Πίνακας 3.2: *Πίνακας αλήθειας των συνδετικών της προτασιακής λογικής*

P	**Q**	**¬P**	**P ∨ Q**	**P ∧ Q**	**P → Q**	**P ↔ Q**
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F

F	F	T	F	F	T	T

Έστω η ερμηνεία $I=\{I(P)=T, I(Q)=T\}$ που αποδίδει λογικές τιμές στα άτομα P και Q. Σύμφωνα με αυτήν την ερμηνεία και τον παραπάνω πίνακα αλήθειας ο τύπος $P \rightarrow Q$ είναι αληθής. Στην περίπτωση αυτή λέμε ότι ο τύπος $P \rightarrow Q$ *ικανοποιείται* από την ερμηνεία I ή εναλλακτικά ότι η ερμηνεία αποτελεί ένα *μοντέλο* (model) του τύπου. Διακρίνονται οι ακόλουθες ενδιαφέρουσες περιπτώσεις τύπων:

- *Ταυτολογία* (*tautology*) είναι ένας τύπος που είναι αληθής κάτω από οποιαδήποτε ερμηνεία, όπως για παράδειγμα ο τύπος $P \vee \neg P$. Εάν ο τύπος F είναι ταυτολογία τότε γράφεται $\models F$, και στον πίνακα αλήθειας του όλες οι γραμμές (ερμηνείες) είναι πάντα αληθείς (Πίνακας 3.3).
- *Αντίφαση* (*contradiction*) είναι ένας τύπος που είναι ψευδής κάτω από οποιαδήποτε ερμηνεία, όπως για παράδειγμα ο τύπος $P \wedge \neg P$. Εάν ο τύπος F είναι αντίφαση τότε γράφεται $F \models \bot$, και στον πίνακα αλήθειας του όλες οι γραμμές (ερμηνείες) είναι πάντα ψευδείς (Πίνακας 3.3).
- *Λογική συνεπαγωγή* (*implication*): Ένας τύπος Q συνεπάγεται λογικά από τον τύπο P εάν κάθε μοντέλο του P είναι επίσης και μοντέλο του Q, δηλαδή όποτε ο P είναι αληθής, τότε και ο Q είναι αληθής. Η περίπτωση συμβολίζεται ως $P \models Q$. Για παράδειγμα, $(P \wedge Q) \models (P \vee Q)$. Στον πίνακα αλήθειας (Πίνακας 3.4) η πρώτη γραμμή (με πράσινο χρώμα) δείχνει ότι όποτε ο πρώτος τύπος είναι αληθής, τότε είναι και ο δεύτερος. Σημειώνεται ότι δεν μας ενδιαφέρουν οι περιπτώσεις που ο πρώτος τύπος είναι ψευδής, ή όταν ο δεύτερος τύπος είναι επίσης αληθής. Αντίθετα, η συνεπαγωγή $(P \wedge Q) \models (\neg P \vee \neg Q)$ δεν ισχύει (πρώτη γραμμή - κόκκινο χρώμα), γιατί υπάρχει περίπτωση (η μοναδική) στην οποία ο πρώτος τύπος είναι αληθής, ενώ ο δεύτερος ψευδής.
- *Λογική ισοδυναμία* (*equivalence*): Δύο τύποι P και Q ονομάζονται ισοδύναμοι εάν οι πίνακες αλήθειας τους είναι οι ίδιοι κάτω από οποιαδήποτε ερμηνεία. Η λογική ισοδυναμία ορίζεται με το σύμβολο "$\Leftrightarrow$", π.χ. $P \Leftrightarrow Q$. Για παράδειγμα, ένας από τους νόμους του de Morgan αποτελεί λογική ισοδυναμία $\neg(P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q)$. Στον πίνακα αλήθειας (Πίνακας 3.4) όλες οι γραμμές (όλες οι ερμηνείες) είναι ίσες για τους δύο τύπους.

Πίνακας 3.3: *Πίνακας αλήθειας παραδειγμάτων ταυτολογίας και αντίφασης*

		Ταυτολογία	**Αντίφαση**
P	**¬P**	**P ∨ ¬P**	**P ∧ ¬P**
T	F	T	F
F	T	T	F

Πίνακας 3.4: *Πίνακας αλήθειας παραδειγμάτων λογικής συνεπαγωγής και ισοδυναμίας*

P	**Q**	**P ∧ Q**	**P ∨ Q**	**¬P ∨ ¬Q**	**¬(P ∧ Q)**
T	T	T	T	F	F
T	F	F	T	T	T
F	T	F	T	T	T
F	F	F	F	T	T

Οι παραπάνω ορισμοί επεκτείνονται σε σύνολα τύπων. Έτσι ένα σύνολο τύπων S ονομάζεται *ταυτολογία*, όταν κάθε ερμηνεία του συνόλου S ικανοποιεί κάθε τύπο του S. Ένα σύνολο τύπων ονομάζεται *ικανοποιήσιμο* (*satisfiable*) εάν υπάρχει μια τουλάχιστον ερμηνεία που να ικανοποιεί όλους τους τύπους του S, ενώ *μη-ικανοποιήσιμο* (*unsatisfiable*) ή *αντίφαση* εάν δεν υπάρχει δυνατή ερμηνεία που να ικανοποιεί όλους τους τύπους του S. Μια πρόταση P *λογικά συνεπάγεται* (*implication ή entailment*) από ένα σύνολο S όταν κάθε ερμηνεία η οποία ικανοποιεί το S ικανοποιεί επίσης και το P. Η περίπτωση αυτή συμβολίζεται με $S \models P$. Με παρόμοιο τρόπο ορίζεται η λογική συνεπαγωγή μεταξύ συνόλων προτάσεων της λογικής. Τέλος, δύο σύνολα προτάσεων S και F ονομάζονται *λογικά ισοδύναμα* εάν $S \models F$ και $F \models S$ δηλαδή έχουν ακριβώς τα ίδια μοντέλα.

Συχνά συγχέεται η λογική ισοδυναμία με το συνδετικό της ισοδυναμίας (Πίνακας 3.2). Η λογική ισοδυναμία ($\Leftrightarrow$) αφορά στη σημασιολογία των υπό εξέταση προτάσεων και αναφέρεται στη σχέση που έχουν τα μοντέλα των συνόλων των προτάσεων στις οποίες αναφέρεται. Το συνδετικό της ισοδυναμίας ($\leftrightarrow$) αποτελεί μέρος της σύνταξης της γλώσσας και παίρνει τιμές κάτω από ορισμένη ερμηνεία των προτάσεων που συμμετέχουν στον τύπο. Η ίδια ακριβώς διαφορά υπάρχει ανάμεσα στη λογική συνεπαγωγή ($\models$) και στο συνδετικό της συνεπαγωγής ($\rightarrow$).

Παραδείγματα εφαρμογής

Να αποδειχθεί ότι ο τύπος $P \rightarrow (Q \rightarrow P)$ είναι ταυτολογία με την χρήση πίνακα αλήθειας. Κατασκευάζεται παρακάτω ο πίνακας αλήθειας των εκφράσεων $Q \rightarrow P$ και $P \rightarrow (Q \rightarrow P)$, στον οποίο φαίνεται ότι στην τελευταία στήλη, όλες οι γραμμές είναι αληθείς, δηλαδή ο τύπος είναι αληθής υπό οποιαδήποτε ερμηνεία, άρα είναι ταυτολογία.

Q	**P**	**Q→P**	**P→(Q→P)**
T	T	T	T
F	T	T	T
T	F	F	T
F	F	T	T

Να αποδειχθεί ότι ο τύπος $\neg(P \vee \neg(P \wedge Q))$ είναι αντίφαση με την χρήση πίνακα αλήθειας. Κατασκευάζεται παρακάτω ο πίνακας αλήθειας των εκφράσεων $P \wedge Q$, $\neg(P \wedge Q)$, $P \vee \neg(P \wedge Q)$ και $\neg(P \vee \neg(P \wedge Q))$, στον οποίο φαίνεται ότι στην τελευταία στήλη, όλες οι γραμμές είναι ψευδείς, δηλαδή ο τύπος είναι ψευδής υπό οποιαδήποτε ερμηνεία, άρα είναι αντίφαση.

P	**Q**	**P∧Q**	**¬(P∧Q)**	**P∨¬(P∧Q)**	**¬(P∨¬(P∧Q))**
T	T	T	F	T	F
F	T	F	T	T	F
T	F	F	T	T	F
F	F	F	T	T	F

Να αποδειχθεί ότι ο τύπος (P→Q)→P δεν είναι ούτε ταυτολογία ούτε αντίφαση με την χρήση πίνακα αλήθειας. Κατασκευάζεται παρακάτω ο πίνακας αλήθειας των εκφράσεων P→Q και (P→Q)→P, στον οποίο φαίνεται ότι στην τελευταία στήλη υπάρχει τουλάχιστον μια γραμμή που είναι αληθής και τουλάχιστον μία

ψευδής, συνεπώς ο τύπος δεν είναι ούτε αληθής υπό οποιαδήποτε ερμηνεία, ούτε ψευδής υπό οποιαδήποτε ερμηνεία, άρα δεν είναι ούτε ταυτολογία, ούτε αντίφαση.

P	**Q**	**P→Q**	**(P→Q)→P**
T	T	T	T
T	F	F	T
F	T	T	F
F	F	T	F

Να αποδειχθεί ότι $(P \rightarrow Q) \land \neg Q \models \neg P$, δηλαδή ο τύπος $\neg P$ αποτελεί λογική συνεπαγωγή του τύπου $(P \rightarrow Q) \land \neg Q$. Κατασκευάζεται παρακάτω ο πίνακας αλήθειας των εκφράσεων $P \rightarrow Q$, $\neg Q$, $(P \rightarrow Q) \land \neg Q$ και $\neg P$, στον οποίο φαίνεται ότι σε όποιες γραμμές ο πρώτος τύπος είναι αληθής (προτελευταία στήλη, τελευταία γραμμή), είναι και ο δεύτερος αληθής (τελευταία στήλη, τελευταία γραμμή).

P	**Q**	**P→Q**	**¬Q**	**(P→Q) ∧ ¬Q**	**¬P**
T	T	T	F	F	F
T	F	F	T	F	F
F	T	T	F	F	T
F	F	T	T	T	T

Να αποδειχθεί η ισοδυναμία $\neg(P \rightarrow Q) \Leftrightarrow P \land \neg Q$. Κατασκευάζεται παρακάτω ο πίνακας αλήθειας των εκφράσεων $\neg Q$, $P \rightarrow Q$, $\neg(P \rightarrow Q)$ και $P \land \neg Q$, στον οποίο φαίνεται ότι ο πίνακας αλήθειας των δύο τελευταίων τύπων είναι ίδιος σε κάθε γραμμή, δηλαδή υπό οποιαδήποτε ερμηνεία είναι αληθείς και ψευδείς και οι δύο ταυτόχρονα.

P	**Q**	**¬Q**	**P→Q**	**¬(P→Q)**	**P ∧ ¬Q**
T	T	F	T	F	F
T	F	T	F	T	T
F	T	F	T	F	F
F	F	T	T	F	F

Λογικές Ισοδυναμίες και Κανονικές Μορφές

Υπάρχει ένα ενδιαφέρον σύνολο ισοδυναμιών της προτασιακής λογικής που χρησιμοποιούνται για τη μετατροπή μιας πρότασης σε κάποια ισοδύναμή της και χρησιμοποιούνται στην εξαγωγή συμπερασμάτων. Ο Πίνακας 3.5 παραθέτει τις πιο γνωστές ισοδυναμίες, οι οποίες προφανώς είναι αληθείς κάτω από οποιαδήποτε ερμηνεία. Για παράδειγμα, στον Πίνακα 3.4 φαίνεται η απόδειξη της ισοδυναμίας (3).

Πίνακας 3.5: *Ισοδυναμίες της προτασιακής λογικής*

	Ισοδυναμία	**Ονομασία**
(1)	$P \Leftrightarrow \neg\neg P$	νόμος της διπλής άρνησης
(2)	$(\neg P \vee \neg Q) \Leftrightarrow \neg(P \wedge Q)$	νόμος De Morgan
(3)	$(\neg P \wedge \neg Q) \Leftrightarrow \neg(P \vee Q)$	νόμος De Morgan
(4)	$(P \vee Q) \wedge R \Leftrightarrow (P \wedge R) \vee (Q \wedge R)$	επιμερισμός ως προς τη σύζευξη
(5)	$(P \wedge Q) \vee R \Leftrightarrow (P \vee R) \wedge (Q \vee R)$	επιμερισμός ως προς τη διάζευξη
(6)	$(P \rightarrow Q) \Leftrightarrow \neg P \vee Q$	
(7)	$(P \leftrightarrow Q) \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$	

Οι δύο τελευταίες ισοδυναμίες υποδηλώνουν ότι οποιοσδήποτε τύπος της προτασιακής λογικής μπορεί να μετατραπεί σε έναν ισοδύναμο χωρίς τη χρήση των συνδετικών της συνεπαγωγής και της ισοδυναμίας. Η παρατήρηση αυτή οδηγεί στον ορισμό των *κανονικών μορφών* (*canonical forms*), δηλαδή μορφών των τύπων της λογικής στις οποίες δεν εμφανίζονται καθόλου κάποια συνδετικά και που ακολουθούν μια συγκεκριμένη δομή. Οι κανονικές μορφές της λογικής, αν και είναι περισσότερο δυσνόητες, μπορεί να αποδειχθούν ιδιαίτερα χρήσιμες για την εύρεση της λογικής τιμής μιας πολύπλοκης έκφρασης αλλά και την εξαγωγή νέας γνώσης. Κλασικά παραδείγματα τέτοιων μορφών είναι η διαζευκτική και η συζευκτική μορφή της λογικής, στις οποίες χρησιμοποιούνται μόνο τα συνδετικά της σύζευξης, διάζευξης και άρνησης.

Στη *διαζευκτική κανονική μορφή της λογικής* (*disjunctive normal form*), οι προτάσεις αποτελούνται από διαζεύξεις τύπων που μπορεί να είναι μόνο *λεκτικά* (*literals*), δηλαδή άτομα ή αρνήσεις ατόμων και συζεύξεις λεκτικών, όπως φαίνεται στο ακόλουθο παράδειγμα:

$$(Q \wedge R \wedge \neg S) \vee (V \wedge W) \vee \ldots \vee (X \wedge Z) \vee \neg Q$$

Αντίστοιχα στη *συζευκτική μορφή* της λογικής (*conjunctive normal form*) οι προτάσεις αποτελούνται από συζεύξεις διαζεύξεων, δηλαδή έχουν τη μορφή:

$$(Q \vee R \vee \neg S) \wedge (V \vee W) \wedge \ldots \wedge (X \vee Z) \wedge \neg Q$$

Ένα παράδειγμα μετατροπής ενός τύπου σε κανονική μορφή είναι το ακόλουθο. Έστω, ο τύπος $P \wedge (P \rightarrow Q)$, ο οποίος μετασχηματίζεται σε κανονική διαζευκτική μορφή ως εξής:

$P \wedge (P \rightarrow Q) \Leftrightarrow$ *Ισοδυναμία (6)*

$P \wedge (\neg P \vee Q) \Leftrightarrow$ *Ισοδυναμία (4)*

$(P \wedge \neg P) \vee (P \wedge Q)$

Η κατανόηση της τελευταίας μορφής του τύπου είναι πιο δύσκολη από την αρχική με την συνεπαγωγή, πλην όμως καθώς έχει σχηματιστεί μια αντίφαση στην πρώτη παρένθεση, μπορεί να απλοποιηθεί ο τύπος και να καταλήξουμε σε ένα τύπου πολύ απλούστερο, ακόμα και από τον αρχικό.

$(P \wedge \neg P) \vee (P \wedge Q) \Leftrightarrow$ *Αντίφαση*

$F \vee (P \wedge Q) \Leftrightarrow$ *Ιδιότητα της διάζευξης*

$P \wedge Q$

Κάθε τύπος της προτασιακής λογικής μπορεί να μετατραπεί σε μια από τις δύο κανονικές μορφές, χρησιμοποιώντας τις ισοδυναμίες του Πίνακα 3.5. Σκοπός είναι η απαλοιφή των συνδετικών της ισοδυναμίας και συνεπαγωγής, καθώς και η κατάλληλη ομαδοποίηση των ατόμων μέσω των ισοδυναμιών του επιμερισμού. Απώτερος σκοπός είναι η απλοποίηση των αρχικών τύπων, όπως είδαμε παραπάνω, ή ακόμα και η κατάληξη σε έναν απλοποιημένο τύπο που θα αποτελεί ταυτολογία ή αντίφαση, χωρίς να χρειαστεί να κατασκευαστεί ο πίνακας αλήθειας για τον αρχικό πολύπλοκο τύπο.

Παραδείγματα εφαρμογής

Με την χρήση των λογικών ισοδυναμιών του Πίνακα 3.5 να αποδειχθούν οι ταυτολογίες, αντιφάσεις και λογικές ισοδυναμίες της προηγούμενης ενότητας.

Να αποδειχθεί ότι ο τύπος $P \rightarrow (Q \rightarrow P)$ είναι ταυτολογία.

$P \rightarrow (Q \rightarrow P) \Leftrightarrow$	*Ισοδυναμία (6)*
$\neg P \vee (\neg Q \vee P) \Leftrightarrow$	*Αντιμεταθετική και Προσεταιριστική ιδιότητα*
$(\neg P \vee P) \vee \neg Q \Leftrightarrow$	*Γνωστή ταυτολογία*
$T \vee \neg Q \Leftrightarrow$	*Ιδιότητα της διάζευξης*
T	

Να αποδειχθεί ότι ο τύπος $\neg(P \vee \neg(P \wedge Q))$ είναι αντίφαση.

$\neg(P \vee \neg(P \wedge Q)) \Leftrightarrow$	*Ισοδυναμία (3) - νόμος de Morgan*
$\neg P \wedge \neg\neg(P \wedge Q) \Leftrightarrow$	*Ισοδυναμία (1) - διπλή άρνηση*
$\neg P \wedge (P \wedge Q) \Leftrightarrow$	*Προσεταιριστική ιδιότητα*
$(\neg P \wedge P) \wedge Q \Leftrightarrow$	*Γνωστή αντίφαση*
$F \wedge Q \Leftrightarrow$	*Ιδιότητα της σύζευξης*
F	

Να αποδειχθεί η ισοδυναμία $\neg(P \rightarrow Q) \Leftrightarrow P \wedge \neg Q$. Ξεκινάμε από το αριστερό μέρος της ισοδυναμίας και καταλήγουμε στο δεξιό μέρος:

$\neg(P \rightarrow Q) \Leftrightarrow$	*Ισοδυναμία (6)*
$\neg(\neg P \vee Q) \Leftrightarrow$	*Ισοδυναμία (3) - νόμος de Morgan*
$\neg\neg P \wedge \neg Q \Leftrightarrow$	*Ισοδυναμία (1) - διπλή άρνηση*
$P \wedge \neg Q$	

Η μέθοδος της χρήσης των ισοδυναμιών δεν μπορεί να βοηθήσει στην απόδειξη των λογικών συνεπαγωγών, όπως για παράδειγμα $(P \rightarrow Q) \wedge \neg Q \vDash \neg P$. Έστω, για παράδειγμα, ότι ξεκινάμε από το αριστερό μέρος της συνεπαγωγής:

$(P \rightarrow Q) \wedge \neg Q \Leftrightarrow$	*Ισοδυναμία (6)*
$(\neg P \vee Q) \wedge \neg Q \Leftrightarrow$	*Ισοδυναμία (4) - επιμεριστική ιδιότητα*
$(\neg P \wedge \neg Q) \vee (Q \wedge \neg Q) \Leftrightarrow$	*Γνωστή αντίφαση*
$(\neg P \wedge \neg Q) \vee F \Leftrightarrow$	*Ιδιότητα της διάζευξης*
$\neg P \wedge \neg Q$	

Ο τελευταίος τύπος δεν απλοποιείται παραπάνω. Η απόδειξη της συνεπαγωγής μπορεί να γίνει με την παρατήρηση ότι αν αληθεύει η σύζευξη του παραπάνω τύπου, αυτό σημαίνει ότι αληθεύει καθένας από τους συζευγμένους τύπους, άρα όταν το $\neg P \wedge \neg Q$ είναι αληθές, τότε είναι και το $\neg P$ αληθές. Χρειάζεται δηλαδή

μία συμπληρωματική διαδικασία χρήσης γνωστών λογικών συνεπαγωγών για να αποδειχθεί η αρχική συνεπαγωγή. Η διαδικασία αυτή ονομάζεται λογική απόδειξη, χρησιμοποιείται για την απόδειξη των λογικών συνεπαγωγών και παρουσιάζεται στην επόμενη ενότητα.

Εξαγωγή Συμπερασμάτων

Στην λογική, η εξαγωγή συμπερασμάτων αφορά είτε στη δημιουργία όλων των τύπων P που λογικά συνεπάγονται (*entailments*) από ένα δεδομένο σύνολο τύπων S, ή στο να διαπιστωθεί εάν ένας τύπος P λογικά συνεπάγεται από το σύνολο αυτό, δηλαδή εάν $S \models P$. Η εξαγωγή συμπερασμάτων στην προτασιακή λογική υλοποιείται είτε με πίνακες αλήθειας ή με τη λογική απόδειξη. Καθώς οι πίνακες αλήθειας δεν είναι πρακτικοί όταν ο αριθμός των ατόμων είναι μεγάλος, στη συνέχεια θα επικεντρωθούμε μόνο στην λογική απόδειξη.

Η *απόδειξη* (*proof*) είναι μια σειρά από βήματα, καθένα από τα οποία είναι η εφαρμογή ενός *κανόνα συμπερασμού* (*rule of inference*) της λογικής σε προηγούμενα συμπεράσματα και υποθέσεις, για τη δημιουργία μιας νέας πρότασης με απώτερο σκοπό την παραγωγή της αποδεικτέας πρότασης ή την κατάληξη σε άτοπο. Ένας κανόνας συμπερασμού είναι μια λογική συνεπαγωγή, η οποία από ένα σύνολο τύπων που ονομάζονται *υποθέσεις*, καταλήγει σε έναν τύπο που ονομάζεται *συμπέρασμα* και ο οποίος αποτελεί λογική συνεπαγωγή των υποθέσεων. Το γεγονός ότι ένας τύπος P μπορεί να αποδειχθεί από ένα αρχικό σύνολο τύπων S, βάσει ενός συνόλου κανόνων συμπερασμού Δ, συμβολίζεται ως $S \vdash_\Delta P$. Υπάρχουν αρκετοί κανόνες συμπερασμού, οι πιο γνωστοί από τους οποίους παρατίθενται στον Πίνακα 3.6.

Πίνακας 3.6: *Κανόνες συμπερασμού της προτασιακής λογικής*

	Κανόνας Συμπερασμού	**Ονομασία**
(1)	$P_1 \wedge P_2 \wedge \ ... \wedge P_N \vdash P_i$	απαλοιφή σύζευξης (and elimination)
(2)	$P_1, P_2, ... P_N \vdash P_1 \wedge P_2 \wedge ... \wedge P_N$	εισαγωγή συζεύξεων (and introduction)
(3)	$P_1 \vdash P_1 \vee P_2 \vee ... \vee P_N$	εισαγωγή διαζεύξεων (or introduction)
(4)	$\neg\neg P \vdash P$	απαλοιφή διπλής άρνησης (double negation elimination)
(5)	$P, P \rightarrow Q \vdash Q$	τρόπος του θέτειν (modus ponens)
(6)	$P \vee Q, \neg Q \vee R \vdash P \vee R$	αρχή της ανάλυσης (resolution)

Μια διαδικασία απόδειξης (*proof procedure*) αποτελείται από ένα σύνολο κανόνων συμπερασμού Δ και έναν αλγόριθμο εφαρμογής τους, βάσει των οποίων εξάγονται (αποδεικνύονται) τα απαιτούμενα συμπεράσματα. Υπάρχουν δύο σημαντικές έννοιες σε κάθε διαδικασία απόδειξης. Η πρώτη αφορά στην ορθότητα της παραγόμενης γνώσης ενώ η δεύτερη στην ικανότητα της διαδικασίας να εξαγάγει όλα τα δυνατά συμπεράσματα. Έτσι μια αποδεικτική διαδικασία ονομάζεται *ορθή* (*sound*) όταν όλα τα συμπεράσματα που εξάγονται αποτελούν και λογικές συνεπαγωγές του αρχικού συνόλου των τύπων, δηλαδή για κάθε P όπου $S \vdash_\Delta P$, ισχύει και $S \models P$. *Πλήρης* (*complete*) ονομάζεται μια αποδεικτική διαδικασία η οποία για κάθε τύπο P ο οποίος λογικά συνεπάγεται από ένα σύνολο τύπων S, μπορεί να "κατασκευάσει" μια απόδειξη, δηλαδή για κάθε P για το οποίο ισχύει $S \models P$, ισχύει και το $S \vdash_\Delta P$.

Η ορθότητα μιας διαδικασίας απόδειξης εξασφαλίζεται από το γεγονός ότι σε κάθε βήμα της εφαρμόζει κάποιον κανόνα συμπερασμού που είναι ορθός. Η ορθότητα των κανόνων συμπερασμού εξασφαλίζεται με την χρήση πινάκων αλήθειας. Για παράδειγμα, στον Πίνακα 3.7 αποδεικνύεται η ορθότητα του κανόνα συμπερασμού modus ponens, καθώς στην πρώτη γραμμή του πίνακα αλήθειας, που είναι και η μοναδική στην οποία όλες οι προτάσεις της υπόθεσης αληθεύουν, αληθεύει και η πρόταση - συμπέρασμα Q. Άρα η λογική συνεπαγωγή είναι αληθής, συνεπώς ο κανόνας συμπερασμού είναι ορθός.

Πίνακας 3.7: *Απόδειξη της ορθότητας του κανόνα συμπερασμού modus ponens*

P	**P → Q**	**Q**
T	T	T
T	F	F
F	T	T
F	T	F

Για τον λογικό προγραμματισμό, εκτός από την ορθότητα και πληρότητα, μας ενδιαφέρει και η αυτοματοποίηση της εξαγωγής συμπερασμάτων από ένα σύνολο τύπων, αλλά και η αποδοτικότητά του (efficiency). Αυτή συνίσταται στο να μπορεί να βρεθεί (μέσω μιας διαδικασίας αναζήτησης ουσιαστικά) η ακολουθία των βημάτων (κανόνων συμπερασμού) που απαιτούνται προκειμένου από τις αρχικές προτάσεις (που θεωρούνται αληθείς) να κατασκευαστεί μια απόδειξη που να καταλήγει στην αποδεικτέα πρόταση (ή προτάσεις). Το μέγεθος του χώρου αναζήτησης εξαρτάται από τον αριθμό των αρχικών προτάσεων, τον αριθμό των αποδεικτέων προτάσεων, αλλά και από τον αριθμό των διαφορετικών βημάτων (κανόνων συμπερασμού) που μπορούν να εφαρμοστούν σε κάθε φάση της απόδειξης. Συνεπώς, όσο λιγότερες επιλογές κανόνων συμπερασμού υπάρχουν (διατηρώντας ωστόσο την πληρότητα της αποδεικτικής διαδικασίας) τόσο μικρότερος θα είναι ο χώρος αναζήτησης, για το ίδιο πρόβλημα. Η βέλτιστη περίπτωση είναι όταν το σύνολο κανόνων συμπερασμού Δ έχει μόνο έναν κανόνα.

Για παράδειγμα, ας θεωρήσουμε το ακόλουθο πρόβλημα απόδειξης. Από τις προτάσεις P, Q, $P \wedge Q \rightarrow R$, να αποδειχθεί η πρόταση R. Προφανώς, για να μπορέσει να εφαρμοστεί ο κανόνας συμπερασμού modus ponens, θα πρέπει να "σχηματιστεί" ο σύνθετος τύπος $P \wedge Q$. Αυτό μπορεί να γίνει μέσω της χρήσης του κανόνα συμπερασμού της εισαγωγής συζεύξεων. Η όλη διαδικασία απεικονίζεται παρακάτω:

$$P, Q \vdash_{\text{and introduction}} P \wedge Q$$

$$P \wedge Q, P \wedge Q \rightarrow R \vdash_{\text{modus ponens}} R$$

Συνεπώς, από το παραπάνω παράδειγμα γίνεται αντιληπτό ότι ο modus ponens δεν μπορεί να είναι ο μοναδικός κανόνας συμπερασμού σε μια αποδεικτική διαδικασία, καθώς η διαδικασία δεν θα ήταν πλήρης. Το ερώτημα λοιπόν που τίθεται είναι εάν υπάρχει κάποιος κανόνας συμπερασμού ο οποίος να μπορεί μόνος του να στηρίξει μια πλήρη αποδεικτική διαδικασία.

Ο κανόνας συμπερασμού της *αρχής της ανάλυσης* (*resolution principle*) που διατυπώθηκε από τον Robinson το 1965 έχει την παραπάνω ικανότητα. Στην αρχή της ανάλυσης (Πίνακας 3.6, κανόνας συμπερασμού 6) τα Q και $\neg Q$ ονομάζονται *συμπληρωματικά ζεύγη* (*complementary pairs*) ενώ η νέα πρόταση $P \vee R$ ονομάζεται *αναλυθέν* (*resolvent*).

Για να εφαρμοστεί ο κανόνας της ανάλυσης, το σύνολο των προτάσεων πρέπει να είναι εκφρασμένο σαν ένα σύνολο διαζεύξεων. Κάθε τέτοια διάζευξη αποτελείται από άτομα ή αρνήσεις ατόμων και αναφέρεται σαν *φράση* (*clause*). Ουσιαστικά αυτό που απαιτείται είναι η μετατροπή όλων των αρχικών προτάσεων στη συζευκτική μορφή της λογικής. Αυτό επιτυγχάνεται εύκολα με τη χρήση των ισοδυναμιών του Πίνακα 3.5. Στο προηγούμενο παράδειγμα, η πρόταση $P \wedge Q \rightarrow R$ μετασχηματίζεται ως εξής:

$$P \wedge Q \rightarrow R \Leftrightarrow \neg(P \wedge Q) \vee R \Leftrightarrow \neg P \vee \neg Q \vee R$$

η οποία είναι σε συζευκτική κανονική μορφή, όπως και οι υπόλοιπες δύο αρχικές προτάσεις - φράσεις P και Q. Η αποδεικτική διαδικασία με μοναδικό κανόνα συμπερασμού την αρχή της ανάλυσης, καταλήγει στο αποδεικτέο R ως εξής:

$$P, \neg P \vee \neg Q \vee R \vdash_{\text{resolution}} \neg Q \vee R$$

$$Q , \neg Q \vee R \vdash_{\text{resolution}} R$$

Το πλεονέκτημα των παραπάνω είναι προφανές. Μπορεί να υπάρξει αποδεικτική διαδικασία η οποία θα εφαρμόζει μηχανιστικά πάντα έναν μοναδικό κανόνα συμπερασμού σε κάθε βήμα της απόδειξης, συνεπώς η διαδικασία μπορεί εύκολα να αυτοματοποιηθεί. Στην αντίθετη περίπτωση, όταν υπάρχουν πολλοί κανόνες συμπερασμού, θα πρέπει σε κάθε βήμα της απόδειξης είτε να εξετάζονται όλοι οι κανόνες, είτε να υπάρχει κάποια ευρετική (heuristic) διαδικασία επιλογής του καταλληλότερου κάθε φορά κανόνα, κάτι το οποίο δεν εγγυάται την αποτελεσματικότητα της διαδικασίας σε όλες τις περιπτώσεις. Βέβαια, το "τίμημα" που πληρώνει κάποιος για να έχει μια αποδεικτική διαδικασία βασισμένη αποκλειστικά στην αρχή της ανάλυσης είναι το "κόστος" μετασχηματισμού όλων των προτάσεων της υπόθεσης σε συζευκτική κανονική μορφή.

Παρόλα τα πλεονεκτήματα της αρχής της ανάλυσης, όπως φάνηκε από το παραπάνω παράδειγμα, εξακολουθούν να υπάρχουν περιπτώσεις για τις οποίες δεν μπορεί να "κατασκευαστεί" απόδειξη, συνεπώς η αντίστοιχη αποδεικτική διαδικασία εξακολουθεί να μην είναι πλήρης. Για παράδειγμα, η αρχή της ανάλυσης δεν μπορεί να εφαρμοστεί καν στην απόδειξη $P \wedge Q \vdash P \vee Q$, ακόμα και αν η σύζευξη της υπόθεσης οδηγήσει στις προτάσεις P και Q, καθώς δεν υπάρχουν καθόλου διαζεύξεις και συμπληρωματικά ζεύγη σε αυτές.

Αποδείχθηκε από τον Robinson ότι αν συνδυαστεί η αρχή της ανάλυσης με την "*εις άτοπο απαγωγή*" (*refutation* ή *proof by contradiction*) προκύπτει μια αποδεικτική διαδικασία ορθή και πλήρης. Ένα τέτοιο σύστημα αποδεικνύει την αλήθεια μιας πρότασης, εισάγοντας στην υπόθεση την άρνηση της αποδεικτέας πρότασης και προσπαθεί να καταλήξει σε άτοπο, εφαρμόζοντας διαδοχικά την αρχή της ανάλυσης σε συμπληρωματικά ζεύγη. Αν η διαδικασία καταλήξει σε άτοπο, που εκφράζεται με την κενή πρόταση □ ή ⊥, τότε θεωρούμε την αποδεικτέα πρόταση αληθή, ενώ σε αντίθετη περίπτωση η πρόταση δεν συνεπάγεται λογικά από την υπόθεση. Η κενή πρόταση (ή αντίφαση) εξάγεται από ένα ζεύγος της μορφής Q, ¬Q.

Στο πρώτο παράδειγμα παραπάνω, η άρνηση της αποδεικτέας πρότασης R εισάγεται στην υπόθεση και η αποδεικτική διαδικασία έχει ως εξής:

$$\neg R, \ \neg P \vee \neg Q \vee R \vdash_{\text{resolution}} \ \neg P \vee \neg Q$$

$$\neg P \vee \neg Q, Q \vdash_{\text{resolution}} \ \neg P$$

$$\neg P, P \vdash_{\text{resolution}} \square$$

Συνεπώς, η πρόταση $\neg R$ προκάλεσε την αντίφαση, άρα κατ' ανάγκην ισχύει το R. Φυσικά στην παραπάνω διαδικασία (όπως και σε όλες τις αποδεικτικές διαδικασίες) υπονοείται ότι το σύνολο των προτάσεων της υπόθεσης είναι συνεπές, δηλαδή δεν περιέχει αντιφάσεις. Η διαπίστωση της συνέπειας της υπόθεσης μπορεί επίσης να ελεγχθεί με την αρχή της ανάλυσης, πλην όμως στα συστήματα λογικού προγραμματισμού κάτι τέτοιο δεν γίνεται και θεωρείται ως ευθύνη του προγραμματιστή.

Σχετικά με την "επίμαχη" απόδειξη $P \wedge Q \vdash P \vee Q$, ακολουθείται η εξής διαδικασία. Η υπόθεση μετασχηματίζεται στην εισαγωγή δύο προτάσεων P και Q. Η άρνηση της αποδεικτέας πρότασης $P \vee Q$ εισάγεται στην υπόθεση, παίρνοντας επίσης δύο προτάσεις $\neg P$ και $\neg Q$:

$$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$$

Η απόδειξη με την αρχή της ανάλυσης έχει ως εξής:

$$\neg P, P \vdash_{\text{resolution}} \square$$

Παραδείγματα εφαρμογής

Στο τέλος της προηγούμενης ενότητας διαπιστώθηκε ότι η απόδειξη μιας λογικής συνεπαγωγής δεν είναι δυνατόν να πραγματοποιηθεί απρόσκοπτα με την χρήση ισοδυναμιών. Συνεπώς τώρα θα δούμε πώς το ίδιο παράδειγμα μπορεί να επιλυθεί εύκολα με τη χρήση της λογικής απόδειξης.

Να αποδειχθεί ότι $(P \rightarrow Q) \wedge \neg Q \models \neg P$. Στην προηγούμενη ενότητα είχε αποδειχθεί (με την χρήση των ισοδυναμιών) ότι: $(P \rightarrow Q) \wedge \neg Q \Leftrightarrow \neg P \wedge \neg Q$. Άρα αρκεί να δείξουμε ότι $\neg P \wedge \neg Q \models \neg P$, κάτι που αποδεικνύεται με την χρήση του κανόνα συμπερασμού της απαλοιφής της σύζευξης: $\neg P \wedge \neg Q \vdash_{\text{and elimination}} \neg P$.

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε την αρχή της ανάλυσης με την βοήθεια της απαγωγής σε άτοπο. Το πρώτο βήμα είναι να μετατραπούν οι αρχικές προτάσεις σε συζευκτική μορφή. Αφού $(P \rightarrow Q) \wedge \neg Q \Leftrightarrow \neg P \wedge \neg Q$, αυτό μας δίνει τις προτάσεις (clauses) $\neg P$ και $\neg Q$. Η άρνηση της προς απόδειξη πρότασης είναι η $\neg\neg P \Leftrightarrow P$. Ο συνδυασμός P και $\neg P$ οδηγεί στην κενή πρόταση $\square$ (άτοπο), άρα το $\neg P$ είναι όντως λογική συνεπαγωγή του αρχικού τύπου.

Ένα πιο ολοκληρωμένο παράδειγμα, είναι το ακόλουθο. Έστω το ακόλουθο σύνολο προτάσεων S:

$$Emerald \rightarrow Precious \quad (1)$$

$$Gem \wedge Green \rightarrow Emerald \quad (2)$$

$$Precious \vee Rare \rightarrow Wanted \quad (3)$$

$$Gem \quad (4)$$

$$Green \quad (5)$$

Να αποδειχθεί $S \vdash_{\Delta} Wanted$, με την χρήση της αποδεικτικής διαδικασίας Δ, η οποία συμπεριλαμβάνει τους κανόνες συμπερασμού: (α) οποιουδήποτε του Πίνακα 3.6, (β) την αρχή της ανάλυσης, αποκλειστικά, και (γ) την αρχή της ανάλυσης σε συνδυασμό με την απαγωγή σε άτοπο.

Για την αποδεικτική διαδικασία (α):

$$(4), (5) \vdash_{\text{and insertion}} Gem \wedge Green \quad (6)$$

$$(6), (2) \vdash_{\text{modus ponens}} Emerald \quad (7)$$

$$(7), (1) \vdash_{\text{modus ponens}} Precious \quad (8)$$

$$(8) \vdash_{\text{or insertion}} Precious \vee Rare \quad (9)$$

$$(9), (3) \vdash_{\text{modus ponens}} Wanted \quad (10)$$

Για την αποδεικτική διαδικασία (β), μετασχηματίζουμε τους τύπους (1)-(3) σε συζευκτική κανονική μορφή (οι 4 και 5 είναι ήδη):

$$\neg Emerald \vee Precious \quad (1')$$

$$\neg(Gem \wedge Green) \vee Emerald \Leftrightarrow \neg Gem \vee \neg Green \vee Emerald \quad (2')$$

$$\neg(Precious \vee Rare) \vee Wanted \Leftrightarrow (\neg Precious \wedge \neg Rare) \vee Wanted \Leftrightarrow (\neg Precious \vee Wanted) \wedge (\neg Rare \vee Wanted) \quad (3')$$

Ο τύπος (3') παραπάνω αντιστοιχεί στις δύο ακόλουθες προτάσεις (clauses):

$$\neg Precious \vee Wanted \quad (3.1)$$

$$\neg Rare \vee Wanted \quad (3.2)$$

Έτσι, στην αποδεικτική διαδικασία συμμετέχουν τελικά οι ακόλουθες προτάσεις:

$$\neg Emerald \vee Precious \quad (1')$$

$$\neg Gem \vee \neg Green \vee Emerald \quad (2')$$

$$\neg Precious \vee Wanted \quad (3.1)$$

$$\neg Rare \vee Wanted \quad (3.2)$$

$$Gem \quad (4)$$

$$Green \quad (5)$$

Η αποδεικτική διαδικασία (β) έχει ως εξής:

$$(4), (2') \vdash_{\text{resolution}} \neg Green \vee Emerald \quad (6')$$

$$(6), (5) \vdash_{\text{resolution}} Emerald \quad (7')$$

$$(7), (1') \vdash_{\text{resolution}} Precious \quad (8')$$

$$(8), (3.1) \vdash_{\text{resolution}} Wanted \quad (9')$$

Για την αποδεικτική διαδικασία (γ) εισάγουμε στο σύνολο των προτάσεων την άρνηση της πρότασης προς απόδειξη:

$$\neg Wanted \quad (6")$$

Έτσι το σύνολο των προτάσεων πλέον περιλαμβάνει τις προτάσεις (1'), (2'), (3.1), (3.2), (4), (5) και (6''). Η αποδεικτική διαδικασία (γ) έχει ως εξής:

$$(6"),(3.1) \vdash_{\text{resolution}} \neg Precious \quad (7'')$$

$$(7''),(1') \vdash_{\text{resolution}} \neg Emerald \quad (8'')$$

$$(8''),(2') \vdash_{\text{resolution}} \neg Gem \vee \neg Green \quad (9'')$$

$$(9''),(4) \vdash_{\text{resolution}} \neg Green \quad (10'')$$

$$(10''),(5) \vdash_{\text{resolution}} \square$$

Το τελευταίο βήμα αποδεικνύει ότι η πρόταση (6'') ήταν ψευδής, άρα αποδεικνύεται η άρνησή της.

Επίλογος

Τα πλεονεκτήματα της χρήσης της προτασιακής λογικής για την αναπαράσταση γνώσης είναι η απλότητα στη σύνταξη και το γεγονός ότι μπορεί να καταλήξει πάντα σε συμπέρασμα (*καταληκτική - decidable*). Σημαντικό όμως μειονέκτημα είναι η έλλειψη γενικότητας που οδηγεί σε ογκώδεις αναπαραστάσεις γνώσης, καθώς κάθε γενική αλήθεια, όπως για παράδειγμα "*Οι Πληροφορικοί είναι προγραμματιστές*" πρέπει να αναπαριστάται με χωριστές λογικές προτάσεις για κάθε ξεχωριστή οντότητα του πραγματικού κόσμου: "*Ο Ηλίας είναι Πληροφορικός και είναι προγραμματιστής*", "*Ο Πέτρος είναι Πληροφορικός και είναι προγραμματιστής*", κ.ο.κ.

3.3 Κατηγορηματική Λογική

Εκτός από το πρόβλημα της ογκώδους αναπαράστασης της γνώσης, η προτασιακή λογική υπονοεί ότι ο κόσμος αποτελείται μόνο από προτάσεις οι οποίες είναι αληθείς ή ψευδείς, χωρίς καμία δυνατότητα διαχωρισμού και προσπέλασης των οντοτήτων του κόσμου στα οποία αναφέρονται οι συγκεκριμένες προτάσεις Έτσι για παράδειγμα στην πρόταση "*Οι Πληροφορικοί είναι προγραμματιστές*" δεν υπάρχει κανένας διαχωρισμός ανάμεσα στα αντικείμενα του πραγματικού κόσμου "*Πληροφορικοί* " και στην ιδιότητα αυτών "*προγραμματιστές*", αλλά ούτε και δίνεται η δυνατότητα προσπέλασης αυτών για τη δημιουργία νέας γνώσης. Έτσι, ακόμη και αν ήταν αληθής η πρόταση "*Ο Πέτρος είναι Πληροφορικός*", η προτασιακή λογική δε θα μπορούσε να καταλήξει στο (εύλογο) συμπέρασμα "*Ο Πέτρος είναι προγραμματιστής*" συνδυάζοντας τις δύο προτάσεις.

Η *κατηγορηματική λογική* (*predicate logic*) αποτελεί επέκταση της προτασιακής λογικής και δίνει λύση στο προηγούμενο πρόβλημα. Σε αυτήν την μορφή της λογικής ο κόσμος περιγράφεται σαν ένα σύνολο *αντικειμένων*, *ιδιοτήτων* και *σχέσεων* που προσδίδονται σε αυτά, δίνοντας έτσι τη δυνατότητα για αναπαραστάσεις που είναι κοντινότερα στην ανθρώπινη εμπειρία. Έτσι, η κατηγορηματική λογική

αντιμετωπίζει το πρόβλημα της μη-προσπελασιμότητας των στοιχείων των ατομικών προτάσεων της προτασιακής λογικής. Για παράδειγμα, στην κατηγορηματική λογική, η πρόταση "*Ο Πέτρος είναι Πληροφορικός*" αναπαριστάται με *πληροφορικός*(*πέτρος*), επιτρέποντας έτσι την προσπέλαση των στοιχείων της συγκεκριμένης οντότητας (*πέτρος*) από τους κανόνες εξαγωγής συμπερασμάτων για τη δημιουργία νέων προτάσεων. Άλλη σημαντική επέκταση σε σχέση με την απλή προτασιακή λογική είναι η ύπαρξη μεταβλητών, που αυξάνει σημαντικά την εκφραστική ικανότητά της, καθώς επιτρέπει την αναπαράσταση "γενικής" γνώσης, όπως για παράδειγμα "*όλοι οι Πληροφορικοί είναι προγραμματιστές*".

Η κατηγορηματική λογική επεκτείνει την προτασιακή λογική εισάγοντας *όρους* (*terms*), *κατηγορήματα* (*predicates*) και *ποσοδείκτες* (*quantifiers*). Το σύνολο των συμβόλων της κατηγορηματικής λογικής (αλφάβητο), περιέχει:

- *Σταθερές*, όπως για παράδειγμα *a*, *b*, *c*, *a_1*, *a_2*, κτλ. Στη σύνταξη που θα ακολουθηθεί στο παρόν κεφάλαιο τα ονόματα των σταθερών θα ξεκινούν πάντα με πεζά γράμματα ή αριθμούς, όπως ακριβώς και στην γλώσσα Prolog.
- *Συναρτησιακά σύμβολα*, όπως για παράδειγμα *f*, *g*, *father-of*, κτλ. Σε κάθε σύμβολο συνάρτησης αντιστοιχεί ένας αριθμός που ονομάζεται *τάξη* (*arity*). Η τάξη ισούται με το πλήθος των *ορισμάτων* (*arguments*) ή αλλιώς, *παραμέτρων* (*parameters*), της συνάρτησης. Έτσι, έχουμε *μοναδιαίες* (*unary*) συναρτήσεις (με ένα όρισμα), *δυαδικές* (*binary*) συναρτήσεις (με δύο ορίσματα) και, γενικά, *ν-αδικές* συναρτήσεις (με ν στο πλήθος ορίσματα).
- *Μεταβλητές*, όπως για παράδειγμα *X*, *Y*, X_1, X_2, *Man*, κτλ. Οι μεταβλητές αναπαριστώνται από κεφαλαία σύμβολα του λατινικού αλφάβητου, ή τουλάχιστον τα ονόματα τους ξεκινούν με κεφαλαίο γράμμα, όπως στην Prolog.
- *Σύμβολα κατηγορημάτων*, όπως *p*, *q*, *color*, κτλ. Κάθε σύμβολο κατηγορήματος έχει μια συγκεκριμένη τάξη.
- *Συνδετικά*: Τα συνδετικά της κατηγορηματικής λογικής είναι όμοια με εκείνα της προτασιακής λογικής και με την ίδια σημασιολογία (Πίνακες 3.1 και 3.2).
- *Ποσοδείκτες*: τον *υπαρξιακό ποσοδείκτη* "∃" (*existential quantifier*) και τον *καθολικό ποσοδείκτη* "∀" (*universal quantifier*).
- *Σύμβολα στίξης*: "(", ")" και ",".
- *Σύμβολα αλήθειας*: *T* (αληθές) και *F* (ψευδές).

Ένας *όρος* (*term*) της κατηγορηματικής λογικής μπορεί να είναι είτε μία σταθερά (όπως *νίκος*, *τίγρης*), είτε μία μεταβλητή (όπως *X*, *Χρώμα*), ή ένας *συναρτησιακός όρος* (*functional term*) της μορφής $f(t_1,t_2,...,t_n)$, όπου *f* είναι ένα συναρτησιακό σύμβολο τάξης *n* και τα ορίσματα t_1, t_2,...,t_n είναι επίσης όροι. Παραδείγματα συναρτησιακών όρων αποτελούν οι εκφράσεις:

πατέραςΤου(νίκου), πατέραςΤου(πατέραςΤου(νίκου)),

εργαζόμενος(γιάννης, επάγγελμα(καθηγητής)), κτλ.

Ένας *ατομικός τύπος* (*atomic formula*) έχει τη μορφή:

$p(a_1, a_2, ..., a_n)$

όπου το *p* είναι ένα σύμβολο κατηγορήματος (ή κατηγόρημα) τάξης *n* και τα a_1, a_2, ..., a_n *ορίσματα* (*arguments*). Κάθε όρισμα είναι ένας όρος, δηλαδή σταθερά, μεταβλητή, ή συναρτησιακός όρος.

Όπως και στην προτασιακή λογική, η σύνδεση προτάσεων για τη δημιουργία *ορθά δομημένων τύπων* γίνεται με τη χρήση συνδετικών. Επιπρόσθετα όμως στην κατηγορηματική λογική οι ορθά δομημένοι τύποι περιέχουν και ποσοδείκτες. Για παράδειγμα, ο τύπος που ακολουθεί είναι ένας ορθά δομημένος τύπος:

$\forall X$ *πληροφορικός(X)* $\rightarrow$ *προγραμματιστής(X)*

Για την επεξήγηση του τύπου και την απόδοση λογικής τιμής απαιτείται να ορίσουμε τη σημασιολογία της κατηγορηματικής λογικής.

Σημασιολογία

Η σημασιολογία της κατηγορηματικής λογικής προϋποθέτει την ύπαρξη ενός *αφηρημένου κόσμου* (*abstract world*) ή *πεδίου* (*domain*) γνώσης στον οποίο αναφέρονται τα σύμβολα της λογικής. Ένας τέτοιος "κόσμος" αποτελείται από τις οντότητες / αντικείμενα, τις ιδιότητές τους και τις συσχετίσεις μεταξύ τους. Οι ιδιότητες μπορούν να θεωρηθούν ως "απλουστευμένες" συσχετίσεις που αφορούν μόνο μία οντότητα. Μια ερμηνεία αντιστοιχεί τους όρους και ατομικούς τύπους της λογικής στα αντικείμενα και στις συσχετίσεις του κόσμου.

Έτσι, οι σταθερές αντιστοιχούνται στις οντότητες του κόσμου, όπως για παράδειγμα η σταθερά *πέτρος* αντιστοιχεί στην οντότητα "πέτρος" του κόσμου. Οι συναρτησιακοί όροι αναφέρονται επίσης σε οντότητες του κόσμου, στις οποίες όμως δε δίνουμε ένα συγκεκριμένο όνομα αλλά χρησιμοποιούμε μια περίφραση για να αναφερθούμε σε αυτές. Για παράδειγμα, αν θέλουμε να αναφερθούμε στο επίθετο της οντότητας "πέτρος", αντί να αντιστοιχίσουμε μια συγκεκριμένη σταθερά (το πραγματικό του επίθετο δηλαδή) χρησιμοποιούμε ένα συναρτησιακό όρο της μορφής *επίθετο*(*πέτρος*). Κάτι τέτοιο συνήθως είναι χρήσιμο όταν δεν γνωρίζουμε ή δεν μας ενδιαφέρει η πραγματική (σταθερή) τιμή του συναρτησιακού όρου, αλλά για την πλήρη αναπαράσταση της γνώσης του κόσμου θα πρέπει να αναπαραστήσουμε κάπως και αυτήν την πληροφορία. Μια τέτοια απεικόνιση όρων σε αντικείμενα του κόσμου ονομάζεται *ανάθεση όρων* (*term assignment*).

Ένας ατομικός τύπος απεικονίζει μια σχέση ανάμεσα σε μια διατεταγμένη πλειάδα (tuple) αντικειμένων στα οποία αναφέρονται τα ορίσματά του, και μπορεί να είναι αληθής, όταν η σχέση ισχύει, ή ψευδής στην αντίθετη περίπτωση. Για παράδειγμα, η γνώση ότι "*ο πέτρος είναι ο προϊστάμενος της άννας*" αναπαριστάται ως *προϊστάμενος*(*πέτρος*, *άννα*).

Μεταβλητές και ποσοδείκτες

Στην *κατηγορηματική λογική πρώτης τάξης* (*first order predicate logic*) οι μεταβλητές αναφέρονται μόνο σε οντότητες / αντικείμενα του κόσμου και όχι σε συναρτησιακά σύμβολα ή κατηγορήματα. Αυτό συντακτικά μεταφράζεται ως: "*μεταβλητές μόνο μέσα στις παρενθέσεις και όχι πριν*"! Υπάρχουν λογικές ανώτερης τάξης (higher order logics) που επεκτείνουν το πεδίο τιμών των μεταβλητών και στο υπόλοιπο αλφάβητο της γλώσσας, η παρουσίαση των οποίων όμως είναι έξω από τους σκοπούς του παρόντος κεφαλαίου, καθώς η γλώσσα Prolog βασίζεται σε ένα υποσύνολο της κατηγορηματικής λογικής πρώτης τάξης.

Οι μεταβλητές της κατηγορηματικής λογικής πρώτης τάξης θεωρούνται όροι, ακριβώς όπως οι σταθερές και οι συναρτησιακοί όροι. Η σωστή ερμηνεία τους όμως επιβάλλει την ποσοτικοποίησή τους από έναν από τους ποσοδείκτες. Ο λόγος θα γίνει προφανής από το παράδειγμα που ακολουθεί. Έστω οι ακόλουθες προτάσεις:

$$\textit{πληροφορικός}(X) \rightarrow \textit{προγραμματιστής}(X)$$

$$\textit{άνθρωπος}(X) \wedge \textit{πληροφορικός}(X)$$

Το τι ακριβώς αναπαριστούν οι προηγούμενες προτάσεις δεν είναι σαφές. Για παράδειγμα, η πρώτη μπορεί να σημαίνει ότι "*όλοι οι πληροφορικοί είναι προγραμματιστές*" ή ότι για κάποιες οντότητες του κόσμου ισχύει η εν λόγω ισοδυναμία. Αντίστοιχα η δεύτερη πρόταση μπορεί να σημαίνει ότι "*όλες οι οντότητες είναι άνθρωποι και πληροφορικοί*" ή ότι "*κάποιος ή κάποιοι είναι άνθρωποι και πληροφορικοί*". Η αποσαφήνιση της σημασίας των δύο αυτών εκφράσεων απαιτεί την εισαγωγή κατάλληλων ποσοδεικτών. Όπως προαναφέρθηκε στην κατηγορηματική λογική υπάρχουν δύο ποσοδείκτες:

- Ο *υπαρξιακός ποσοδείκτης* "∃" (*existential quantifier*). Ο τύπος $(\exists X)(\varphi(X))$ διαβάζεται "υπάρχει X, τέτοιο ώστε ο τύπος $\varphi(X)$ να είναι αληθής" και σημαίνει ότι υπάρχει τουλάχιστον ένα στοιχείο στο πεδίο τέτοιο, ώστε όταν το X αναφέρεται σε αυτό, ο τύπος $\varphi(X)$ καθίσταται αληθής.
- Ο *καθολικός ποσοδείκτης* "∀" (*universal quantifier*). Ο τύπος $(\forall X)(\varphi(X))$ διαβάζεται "για κάθε X, ο $\varphi(X)$ είναι αληθής" και σημαίνει ότι καθένα από τα στοιχεία του πεδίου τον καθιστά αληθή, όταν το X αναφέρεται σε αυτά τα στοιχεία. Δηλαδή ο τύπος $(\forall X)(\varphi(X))$ είναι αληθής, όταν κάθε στοιχείο του πεδίου τον καθιστά αληθή.

Έτσι μια σωστότερη αναπαράσταση της παραπάνω γνώσης είναι:

(∀X)(πληροφορικός(X) → προγραμματιστής(X)) *"όλοι οι πληροφορικοί είναι προγραμματιστές"*

(∃X)(άνθρωπος(X) ∧ πληροφορικός(X)) *"κάποιος άνθρωπος είναι πληροφορικός"*

Από το παραπάνω παράδειγμα μπορούμε να αντλήσουμε τις ακόλουθες παρατηρήσεις:

- Ο καθολικός ποσοδείκτης συνήθως συνοδεύει τις προτάσεις στις οποίες ο πιο σημαντικός τελεστής είναι η συνεπαγωγή (ή η ισοδυναμία).
- Ο υπαρξιακός ποσοδείκτης συνήθως συνοδεύει τις προτάσεις στις οποίες ο πιο σημαντικός τελεστής είναι είτε η σύζευξη, είτε η διάζευξη, αλλά όχι η συνεπαγωγή ή η ισοδυναμία.

Ο συνδυασμός του καθολικού ποσοδείκτη με σύζευξη ή διάζευξη ουσιαστικά υποδηλώνει κάποια λογική σχέση που πρέπει να πληρούν όλες οι οντότητες του πεδίου της γνώσης, άρα δρα ως μια λογική δέσμευση ή περιορισμός πάνω στα στοιχεία του πεδίου. Συνήθως, λογικές εκφράσεις τέτοιας μορφής αναφέρονται ως περιορισμοί ακεραιότητας (δεδομένων) (data integrity constraints). Για παράδειγμα, αν υιοθετούσαμε την χρήση του καθολικού ποσοδείκτη στην δεύτερη πρόταση παραπάνω, το αποτέλεσμα θα ήταν να περιορίζαμε το γνωστικό πεδίο μόνο σε οντότητες οι οποίες θα ήταν άνθρωποι και πληροφορικοί, ταυτόχρονα, καθώς μόνο αυτές θα ικανοποιούσαν τον συγκεκριμένο περιορισμό ακεραιότητας, πέραν φυσικά των υπολοίπων λογικών προτάσεων που ενδέχεται να υπάρχουν.

Δύο ακόμα σημαντικές παρατηρήσεις, οι οποίες μπορούν να βοηθήσουν στην κατανόηση των ποσοδεικτών είναι οι ακόλουθες:

- Για να αληθεύει η πρόταση (∃*X*)(*άνθρωπος*(*X*) ∧ *πληροφορικός*(*X*)), θα πρέπει να υπάρχει οπωσδήποτε τουλάχιστον ένας άνθρωπος και να είναι και πληροφορικός. Από την άλλη, η πρόταση είναι ψευδής όταν: (α) δεν υπάρχει κανείς άνθρωπος στο πεδίο της γνώσης, ή (β) δεν υπάρχει κανείς πληροφορικός στο πεδίο της γνώσης, ή (γ) και τα δύο. Συνεπώς, ο υπαρξιακός ποσοδείκτης είναι "σφιχτός" στην αλήθεια και "χαλαρός" στο ψεύδος.
- Αντίθετα, η πρόταση (∀*X*)(*πληροφορικός*(*X*) → *προγραμματιστής*(*X*)), είναι αληθής όταν: (α) για κάθε πληροφορικό που υπάρχει στο πεδίο της γνώσης, αυτός είναι και προγραμματιστής (T→T), (β) για οποιοδήποτε άλλο X του πεδίου της γνώσης, που δεν είναι πληροφορικός, χωρίς να ενδιαφέρει αν είναι προγραμματιστής ή όχι (F→T, F→F), και (γ) όταν δεν υπάρχει κανείς πληροφορικός στο πεδίο της γνώσης. Από την άλλη, η πρόταση είναι ψευδής όταν υπάρχει έστω και ένας πληροφορικός που δεν είναι προγραμματιστής (T↛F). Συνεπώς, ο καθολικός ποσοδείκτης είναι "χαλαρός" στην αλήθεια και "σφιχτός" στο ψεύδος.

Παραδείγματα εφαρμογής

Να διατυπωθεί σε κατηγορηματική λογική η πρόταση "*όλοι έχουν μητέρα*". Συνήθως σε τέτοιες περιπτώσεις ορίζουμε πρώτα τα κατηγορήματα που θα χρησιμοποιηθούν, δηλαδή τα ονόματά τους, τον αριθμό παραμέτρων (τάξη - arity) και τη φυσική σημασία του καθενός από αυτά και μετά διατυπώνεται η πρόταση σε κατηγορηματική λογική προσέχοντας την επιλογή των ποσοδεικτών που ποσοτικοποιούν τις μεταβλητές της πρότασης. Στην συγκεκριμένη περίπτωση επιλέγουμε το κατηγόρημα *mother*(*X,Y*), του οποίου η σημασία είναι "*Η X είναι μητέρα του(της) Y*". Όσον αφορά την επιλογή των ποσοδεικτών για τις μεταβλητές *X* και *Y*, κάνουμε τις εξής σκέψεις. Η λέξη "*όλοι*" παραπέμπει στον καθολικό ποσοδείκτη (∀), χωρίς αμφιβολία. Άρα η μεταβλητή *Y* είναι καθολικά ποσοτικοποιημένη Άλλες λέξεις της φυσικής γλώσσας που παραπέμπουν στον καθολικό ποσοδείκτη είναι "*όποιος*", "*οτιδήποτε*", "*κάθε*", κλπ. Για την επιλογή του ποσοδείκτη της μεταβλητής *X* παρατηρούμε ότι στην συγκεκριμένη φράση η λέξη "*μητέρα*" είναι στον ενικό αριθμό, άρα υπονοείται ο υπαρξιακός ποσοδείκτης (∃). Έτσι λοιπόν η παραπάνω φράση αναπαρίσταται στην κατηγορηματική λογική ως:

∀*Y*∃*X mother(X,Y)*

και ερμηνεύεται "*Για κάθε οντότητα Y του πεδίου γνώσης, υπάρχει μια οντότητα X με την οποία η οντότητα X συνδέεται μέσω της σχέσης μητέρας - παιδιού*". Εδώ φυσικά γίνεται η παραδοχή ότι όλες οι οντότητες του συγκεκριμένου πεδίου της γνώσης είναι άνθρωποι ή ζωντανά όντα, έτσι ώστε να μην χρειάζεται να αποσαφηνιστεί το είδος των τιμών που μπορεί να πάρουν οι μεταβλητές *X* και *Y*.

Επίσης, θα εξηγήσουμε γιατί άλλες επιλογές για τους ποσοδείκτες θα ήταν λάθος. Για παράδειγμα, αν επιλέγαμε τον καθολικό ποσοδείκτη για την μεταβλητή *X*, η πρόταση

$$\forall Y \forall X\ mother(X,Y)$$

θα ερμηνευόταν ως "*όλοι (X και Y) συνδέονται μεταξύ τους με την σχέση mother*", δηλαδή αν διαλέξω δύο οποιεσδήποτε οντότητες *X* και *Y* από το πεδίο γνώσης πρέπει υποχρεωτικά να επαληθεύουν την σχέση μητέρας - παιδιού μεταξύ τους, κάτι που είναι προφανέστατα λάθος.

Η επιλογή υπαρξιακών ποσοδεικτών και για τις δύο μεταβλητές καταλήγει σε μία πρόταση που δεν είναι ισοδύναμη με την πρόταση σε φυσική γλώσσα. Συγκεκριμένα, η πρόταση:

$$\exists Y \exists X\ mother(X,Y)$$

ερμηνεύεται ως "*κάποια οντότητα Y έχει με κάποια άλλη οντότητα X την σχέση μητέρας - παιδιού*". Η παραπάνω πρόταση δεν είναι λάθος, αλλά δεν είναι και ακριβώς ισοδύναμοι με την αρχική, γιατί είναι πολύ συγκεκριμένη και καλύπτει ένα ζευγάρι οντοτήτων με την συγκεκριμένη σχέση, ενώ η αρχική μας πρόταση αναφερόταν και κάλυπτε όλα τα αντίστοιχα ζευγάρια. Δηλαδή, το πρόβλημα της συγκεκριμένης πρότασης έγκειται στην πληρότητα κάλυψης του πεδίου γνώσης και όχι στην ορθότητά της.

Τέλος, αν επιλέγαμε την τοποθέτηση του υπαρξιακού ποσοδείκτη πριν από τον καθολικό, δηλαδή:

$$\exists X \forall Y\ mother(X,Y)$$

τότε η σημασία της πρότασης είναι αρκετά διαφορετική: "*Υπάρχει μία οντότητα X η οποία συνδέεται με όλες τις άλλες οντότητες Y με την σχέση μητέρας - παιδιού*", δηλαδή υπονοεί ότι όλοι έχουν την ίδια μητέρα, κάτι που είναι επίσης λανθασμένο. Στην "ορθή" διατύπωση, προηγουμένως, η υπαρξιακά ποσοτικοποιημένη μεταβλητή βρισκόταν δεξιότερα από την καθολικά ποσοτικοποιημένη μεταβλητή, επομένως η σημασία της ελεγχόταν από αυτήν, δηλαδή οι τιμές που μπορεί να πάρει εξαρτώνται από την εκάστοτε τιμή της καθολικά ποσοτικοποιημένης μεταβλητής. Συνεπώς, η σειρά τοποθέτησης των υπαρξιακών ποσοδεικτών, ειδικά αν βρίσκονται μέσα στην εμβέλεια καθολικά ποσοτικοποιημένων μεταβλητών είναι πολύ σημαντική και θα πρέπει να ελέγχεται προσεκτικά.

Παραλλαγή της παραπάνω πρότασης είναι η ακόλουθη: "*Όλοι οι άνθρωποι έχουν μητέρα*". Στην πρόταση αυτοί διευκρινίζεται ότι το "*όλοι*" αναφέρεται μόνο σε όσες οντότητες του πεδίου γνώσης χαρακτηρίζονται από την ιδιότητα "*άνθρωποι*", άρα επιτρέπει να υπάρχουν και άλλες οντότητες που δεν είναι άνθρωποι ή ζωντανά όντα. Η απόδοση μιας ιδιότητας ή τύπου σε μία οντότητα γίνεται συνήθως με την χρήση κατηγορημάτων τάξης ένα: *human*(*X*). Καθώς τώρα υπάρχουν δύο κατηγορήματα στην συγκεκριμένη πρόταση, *human* και *mother*, τίθεται θέμα του λογικού συνδετικού με το οποίο θα συνδυαστούν. Συνήθως ο καθολικός ποσοδείκτης (∀), ο οποίος υπονοείται από την ύπαρξη της λέξης "*όλοι*", συνδυάζεται με την συνεπαγωγή (→). Έτσι λοιπόν η πρόταση διατυπώνεται σε κατηγορηματική λογική ως:

$$\forall Y\ human(Y) \rightarrow \exists X\ mother(X,Y)$$

και ερμηνεύεται ως εξής: "*Για κάθε οντότητα Y η οποία χαρακτηρίζεται από την ιδιότητα άνθρωπος, υπάρχει κάποια οντότητα X, με την οποία συνδέεται με την σχέση μητέρας - παιδιού*". Η παρουσία της ατομικής πρότασης *human*(*Y*) στο αριστερό μέρος της συνεπαγωγής μας εξασφαλίζει ότι μόνο για τους ανθρώπους μας ενδιαφέρει η αλήθεια της σχέσης *mother*, ενώ μας είναι αδιάφορη για όλα τα υπόλοιπα *X*, κάτι που αποτελεί χαρακτηριστική ιδιότητα της συνεπαγωγής. Κανένα άλλο συνδετικό δεν μπορεί να επιτύχει αυτήν την σημασιολογία, παρά μόνο μέσω της ισοδυναμίας (6) του Πίνακα 3.5. Για παράδειγμα, η χρήση της σύζευξης δημιουργεί μια διαφορετικής ερμηνείας πρόταση:

$$\forall Y\ human(Y) \wedge \exists X\ mother(X,Y)$$

η οποία ερμηνεύεται ως "*όλες οι οντότητες είναι άνθρωποι και έχουν κάποια μητέρα*", δηλαδή για να αληθεύει, αποκλείει την ύπαρξη οντοτήτων του πεδίου γνώσης άλλων εκτός των ανθρώπων. Αυτό όμως σημαίνει ότι ισοδυναμεί με το προηγούμενο παράδειγμα στο οποίο υπονοούνταν ότι όλες οι οντότητες ήταν άνθρωποι.

Ένα ακόμα παράδειγμα, στο ίδιο πνεύμα, είναι η πρόταση "*όλοι οι άνθρωποι έχουν πατέρα και μητέρα*". Αυτή η πρόταση αποδίδεται ως:

$$\forall Y\ human(Y) \rightarrow \exists X\ mother(X,Y) \wedge \exists Z\ father(Z,Y)$$

Εδώ είναι χαρακτηριστική η χρήση δύο διαφορετικών υπαρξιακά ποσοτικοποιημένων μεταβλητών *X* και *Z*, καθώς ο πατέρας και η μητέρα αποτελούν διαφορετικές οντότητες. Η χρήση της ίδιας μεταβλητής στις δύο ατομικές προτάσεις δεν είναι λάθος, καθώς η εμβέλεια του κάθε υπαρξιακού ποσοδείκτη είναι μόνο η ατομική πρόταση:

$$\forall Y\ human(Y) \rightarrow \exists X\ mother(X,Y) \wedge \exists X\ father(X,Y)$$

Στην παραπάνω πρόταση οι δύο εμφανίσεις του X δεν υπονοούν την ίδια οντότητα, καθώς υπάρχουν δύο υπαρξιακοί ποσοδείκτες με τοπική εμβέλεια στις σχέσεις *mother* και *father*. Παρόλα αυτά, είναι πιο ασφαλής η χρήση διαφορετικών μεταβλητών, καθώς η χρήση της ίδιας μεταβλητής μπορεί να οδηγήσει σε λανθασμένες ερμηνείες, όπως για παράδειγμα να ερμηνευθεί το X ως κοινή οντότητα, δηλαδή να θεωρηθεί ότι η παραπάνω πρόταση είναι ισοδύναμη με την ακόλουθη:

$$\forall Y\ human(Y) \rightarrow \exists X\ (\ mother(X,Y) \wedge father(X,Y)\ )$$

κάτι που δεν ισχύει, όπως θα εξηγηθεί στην επόμενη ενότητα.

Η πρόταση "*Ο Κώστας είναι παππούς*" είναι πρόταση που αφορά σε συγκεκριμένη οντότητα, συνεπώς θα χρησιμοποιηθούν μόνο υπαρξιακοί ποσοδείκτες. Κάποιος είναι παππούς αν το παιδί του έχει παιδιά, δηλαδή είναι γονέας. Εδώ θεωρούμε ότι υπάρχει η σχέση *parent*(*X*,*Y*) η οποία αληθεύει όταν η οντότητα *X* είναι γονέας της οντότητας *Y*. Η πρόταση διαμορφώνεται ως εξής:

$$\exists Z\ (\ father(kostas,Z) \wedge \exists Y\ parent(Z,Y)\ )$$

και ερμηνεύεται ως "*Υπάρχει κάποια οντότητα Z της οποίας πατέρας είναι ο Κώστας και για την οντότητα αυτή υπάρχει οντότητα Y η οποία είναι παιδί της*". Για τον Κώστα χρησιμοποιήσαμε την σχέση *father* και όχι *parent* γιατί είναι φανερό ότι είναι άνδρας. Θα μπορούσε όμως να χρησιμοποιηθεί και η σχέση *parent* χωρίς διαφορά στο νόημα, στο συγκεκριμένο παράδειγμα. Εναλλακτικά, αν θεωρήσουμε ότι δεν υφίσταται η σχέση *parent*, αλλά μόνο οι σχέσεις *father* και *mother*, τότε η παραπάνω πρόταση θα μπορούσε να διατυπωθεί ως εξής:

$$\exists Z\ (\ father(kostas,Z) \wedge \exists Y\ (\ father(Z,Y) \vee mother(Z,Y)\ )\ )$$

Η διάζευξη μεταξύ των σχέσεων *father*, *mother* εκφράζει ουσιαστικά την γνώση πως γονέας μπορεί να είναι κάποιος είτε ως πατέρας ή ως μητέρα. Προφανώς οι δύο σχέσεις δεν μπορούν να συναληθεύουν, πλην όμως με την χρήση της διάζευξης κάτι τέτοιο είναι "τεχνικά" δυνατό. Αυτή η τελευταία παρατήρηση αποκλείει προφανώς την χρήση άλλων συνδετικών, όπως για παράδειγμα της σύζευξης, καθώς τότε θα έπρεπε υποχρεωτικά κάποιος να είναι ταυτόχρονα πατέρας και μητέρα, που είναι άτοπο (νοηματικά). Η χρήση του υπαρξιακού ποσοδείκτη μπροστά από την διάζευξη σημαίνει ότι "*Υπάρχει κάποια οντότητα Y, η οποία είναι πατέρας ή μητέρα της οντότητας X*". Προφανώς αυτό δεν σημαίνει ότι πρόκειται για την ίδια οντότητα *Y*, καθώς η χαλαρότητα της διάζευξης δεν υποχρεώνει τους δύο όρους να συναληθεύουν για την ίδια οντότητα *Y*. Εναλλακτικά θα μπορούσε να διατυπωθεί η παραπάνω πρόταση με δύο διαφορετικές μεταβλητές *Y* και *W* οι οποίες θα ήταν υπαρξιακά ποσοτικοποιημένες ξεχωριστά:

$$\exists Z\ (\ father(kostas,Z) \wedge (\ \exists Y\ father(Z,Y) \vee \exists W\ mother(Z,W)\ )\ )$$

Η παραπάνω διατύπωση (της δεύτερης παρένθεσης) ερμηνεύεται ως "*Υπάρχει κάποια οντότητα Y, η οποία είναι πατέρας της οντότητας X ή υπάρχει μια οντότητα W, η οποία είναι μητέρα της οντότητας X*" και είναι ισοδύναμη με την αρχική λόγω της δυνατότητας μετονομασίας των μεταβλητών και της μεταφοράς του υπαρξιακού ποσοδείκτη στην περίπτωση της διάζευξης, όπως θα εξηγηθεί στην επόμενη ενότητα.

Η πρόταση "*όλες οι μητέρες είναι γονείς*" έχει όλες τις λέξεις στον πληθυντικό αριθμό, κάτι που υπονοεί ότι θα χρησιμοποιηθούν μόνο καθολικοί ποσοδείκτες για την αναπαράστασή της. Αν υποθέσουμε ότι στη διάθεσή μας υπάρχουν οι σχέσεις *mother* και *parent* τάξης δύο, όπως και προηγουμένως, τότε η παραπάνω πρόταση διατυπώνεται ως εξής:

$$\forall X\ \forall Y\ mother(X,Y) \rightarrow parent(X,Y)$$

και ερμηνεύεται ως "*Όλες οι οντότητες X και Y οι οποίες συνδέονται με την σχέση μητέρας - παιδιού, συνδέονται και με την σχέση γονέα - παιδιού*". Εδώ βλέπουμε ότι χρειάστηκε η ποσοτικοποίηση της

μεταβλητής *Y* (του παιδιού), παρόλο που στην αρχική πρόταση δεν αναφερόταν καθόλου. Αυτό συμβαίνει γιατί οι ιδιότητες *μητέρα* και *γονέας* ισχύουν για κάποια οντότητα *X* μόνο αν συμμετέχει σε κάποια σχέση μητέρα - παιδιού ή γονέα - παιδιού, αντίστοιχα, με κάποια άλλη οντότητα *Y*. Η χρήση του υπαρξιακού ποσοδείκτη για την μεταβλητή Υ δεν θα ήταν λάθος, πλην όμως στερείται γενικότητας, άρα έχει πρόβλημα πληρότητας. Συγκεκριμένα, η ακόλουθη πρόταση:

$$\forall X \exists Y \; mother(X,Y) \rightarrow parent(X,Y)$$

θα ερμηνευόταν ως "*Για κάθε οντότητα X, αν υπάρχει κάποια οντότητα Y με την οποία υπάρχει σχέση μητέρας - παιδιού, τότε με την ίδια οντότητα υπάρχει και η σχέση γονέα - παιδιού*", δηλαδή το συμπέρασμα της συνεπαγωγής θα ίσχυε μόνο για την μία οντότητα *Y* (για το ένα παιδί) και όχι για οποιαδήποτε οντότητα *Y* (για όλα τα παιδιά της μητέρας).

Μια απλούστερη εκδοχή του παραπάνω παραδείγματος θα ήταν αν χρησιμοποιούνταν κατηγορήματα τάξης ένα για τις ιδιότητες μητέρα και γονέας, ώστε να μην υπάρχει ανάγκη ποσοτικοποίησης της μεταβλητής *Y*:

$$\forall X \; mother(X) \rightarrow parent(X)$$

Η πρόταση "*κανένας πατέρας δεν είναι μητέρα*", εκφράζει αρνητική γνώσης γενικής μορφής, συνεπώς θα πρέπει να χρησιμοποιηθεί καθολικός ποσοδείκτης και άρνηση. Θεωρώντας ότι υπάρχουν κατηγορήματα τάξης ένα για τις ιδιότητες *πατέρας* και *μητέρα*, η παραπάνω πρόταση διατυπώνεται ως εξής:

$$\forall X \neg (father(X) \wedge mother(X))$$

και ερμηνεύεται ως "*Για κάθε οντότητα X, δεν μπορεί να ισχύουν ταυτόχρονα οι ιδιότητες πατέρας και μητέρα*". Εναλλακτικά, η παραπάνω πρόταση θα μπορούσε να διατυπωθεί και με την χρήση υπαρξιακού ποσοδείκτη ως εξής:

$$\neg \exists X (father(X) \wedge mother(X))$$

η οποία ερμηνεύεται ως "*Δεν υπάρχει κάποια οντότητα X για την οποία να ισχύουν ταυτόχρονα οι ιδιότητες πατέρας και μητέρα*", η οποία είναι εντελώς ισοδύναμη με την προηγούμενη, όπως θα εξηγηθεί στην επόμενη ενότητα. Η λέξη "*κανείς*" δηλαδή μπορεί να αντιστοιχηθεί είτε στην ποσοτικοποίηση "*δεν υπάρχει κάποιος*" (¬∃), ή στην ποσοτικοποίηση "*όλοι δεν …*" (∀¬). Αυτή είναι και μία από τις λίγες περιπτώσεις στις οποίες ο καθολικός ποσοδείκτης συνοδεύει το συνδετικό της σύζευξης.

Τέλος, παρατίθεται ένα μεγαλύτερο παράδειγμα αναπαράστασης γνώσης σε κατηγορηματική λογική. Να μετατραπούν σε προτάσεις κατηγορηματικής λογικής οι ακόλουθες προτάσεις:

Υπάρχουν φοιτητές που παρακολουθούν όλα τα μαθήματα. (1)

Οι φοιτητές δεν παρακολουθούν τα βαρετά μαθήματα. (2)

Τα μαθήματα που διδάσκει ο Πέτρος παρακολουθούνται από όλους τους φοιτητές. (3)

Θα χρησιμοποιήσουμε τα ακόλουθα κατηγορήματα για να αποδώσουμε τις έννοιες και ιδιότητες που υπάρχουν στις προτάσεις:

- *student*(*X*): Ο *X* είναι φοιτητής
- *course*(*X*): Το *X* είναι μάθημα
- *boring*(*X*): Το *X* (μάθημα) είναι βαρετό
- *attends*(*X*,*Y*): Ο *X* (φοιτητής) παρακολουθεί το *Y* (μάθημα)
- *teaches*(*X*,*Y*): Ο *X* (καθηγητής) διδάσκει το *Y* (μάθημα)

Για την πρόταση (1), είναι φανερό ότι η λέξη "*υπάρχουν*" μας οδηγεί στην χρήση του υπαρξιακού ποσοδείκτη για την μεταβλητή που εμφανίζεται στο κατηγόρημα *student*, άρα το πρώτο σκέλος της πρότασης αναπαρίσταται ως: ∃*X* *student*(*X*). Στο δεύτερο σκέλος πρέπει να χρησιμοποιηθεί το κατηγόρημα *attends*(*X*,*Y*), όπου *X* ο φοιτητής και *Y* το μάθημα. Πλην όμως θα πρέπει να εξασφαλιστεί ότι η μεταβλητή *Y*

αναπαριστά μάθημα και αυτό μπορεί να γίνει με την χρήση του κατηγορήματος *course*(*Y*). Επειδή δε η πρόταση αναφέρεται σε "*όλα τα μαθήματα*", η μεταβλητή Υ πρέπει να είναι καθολικά ποσοτικοποιημένη. Από την προηγούμενη συζήτηση προέκυψε ότι ο καθολικός ποσοδείκτης συνδυάζεται συνήθως με το συνδετικό της συνεπαγωγής, συνεπώς το δεύτερο σκέλος της πρότασης θα πρέπει να συνδέει τις 2 ατομικές προτάσεις μέσω συνεπαγωγής:

$$\forall Y\ course(Y) \rightarrow attends(X, Y)$$

με την ερμηνεία: "*αν κάποιο Υ είναι μάθημα, τότε ο Χ το παρακολουθεί*". Τα δύο σκέλη της πρότασης έχουν την μεταβλητή Χ κοινή (καθώς μιλάμε για τους ίδιους φοιτητές), συνεπώς ο κύριος ποσοδείκτης της σύνθετης πρότασης είναι ο υπαρξιακός. Προηγουμένως αναφέρθηκε ότι ο υπαρξιακός ποσοδείκτης συνήθως συνδυάζεται με την σύζευξη, άρα το πρώτο σκέλος με το δεύτερο συνδέονται με το συνδετικό της σύζευξης. Η συνολική πρόταση (1) διαμορφώνεται λοιπόν ως εξής:

$$\exists X\,(student(X) \wedge \forall Y\,(course(Y) \rightarrow attends(X, Y))) \qquad (1\alpha)$$

Για την πρόταση (2) παρατηρούμε ότι όλες οι έννοιες είναι στον πληθυντικό αριθμό, συνεπώς θα χρησιμοποιηθούν μόνο καθολικοί ποσοδείκτες. Άρα το πρώτο σκέλος της πρότασης θα είναι $\forall X\ student(X)$ και λόγω του καθολικού ποσοδείκτη θα συνδέεται με συνεπαγωγή με το δεύτερο σκέλος. Στο δεύτερο σκέλος και πάλι θα υπάρχει καθολικός ποσοδείκτης (όλα τα μαθήματα) και συνεπαγωγή. Όπως πριν ο ατομικός τύπος *course*(Y) στο αριστερό μέρος της συνεπαγωγής θα εξασφαλίζει ότι οι οντότητες για τις οποίες ισχύει η πρόταση θα πρέπει να είναι μαθήματα. Επειδή θα πρέπει επιπλέον να είναι και "βαρετά" μαθήματα, θα υπάρχει σύζευξη (στο αριστερό μέρος) με το κατηγόρημα *boring*. Το συμπέρασμα της σύζευξης, λόγω της ύπαρξης του αρνητικού μορίου "*δεν*", πρέπει να είναι αρνητικό και να σχετίζεται με το κατηγόρημα *attends* που συσχετίζει φοιτητές με τα μαθήματα που παρακολουθούν. Η συνολική πρόταση (2) διαμορφώνεται ως εξής:

$$\forall X\ (student(X) \rightarrow \forall Y\ (\ (course(Y) \wedge boring(Y)) \rightarrow \neg attends(X, Y))) \qquad (2\alpha)$$

Για την πρόταση (3) σημειώνεται πως και αυτή αποτελείται από δύο σκέλη. Στο πρώτο σκέλος πρέπει να προσδιοριστούν όλα τα μαθήματα που διδάσκει ένας συγκεκριμένος καθηγητής και στο δεύτερο σκέλος αυτά τα μαθήματα συνδέονται με τους φοιτητές που τα παρακολουθούν. Άρα το πρώτο με το δεύτερο σκέλος πρέπει να έχουν κοινή μεταβλητή η οποία αντιπροσωπεύει τα μαθήματα. Αυτό υπονοεί την χρήση καθολικού ποσοδείκτη για την μεταβλητή αυτή και την σύνδεση πρώτου με δεύτερο μέρος της πρότασης με το συνδετικό της συνεπαγωγής. Στο πρώτο σκέλος προσδιορίζονται οντότητες *X* που είναι μαθήματα (*course*) και διδάσκονται (teaches) από έναν συγκεκριμένο καθηγητή: *course*(*X*) ∧ *teaches*(*petros*, *X*). Στο δεύτερο σκέλος θα πρέπει να προσδιοριστούν όλοι (∀) οι φοιτητές (*student*) που παρακολουθούν το μάθημα *X* (*attends*). Η χρήση του καθολικού ποσοδείκτη υποδεικνύει την χρήση του συνδετικού της συνεπαγωγής για την σύνδεση των δύο κατηγορημάτων: ∀*Z* (*student*(*Z*) → *attends*(*Z*, *X*)). Το *student*(*Z*) εξασφαλίζει ότι το *attends*(*Z*,*X*) θα πρέπει να οπωσδήποτε να είναι αληθές μόνο για οντότητες που είναι όντως φοιτητές. Η συνολική πρόταση (3) είναι διαμορφώνεται ως εξής:

$$\forall X\ ((course(X) \wedge teaches(petros, X)) \rightarrow \forall Z\ (student(Z) \rightarrow attends(Z, X))) \qquad (3\alpha)$$

Ισοδυναμίες και Κανονικές Μορφές

Όπως και στην προτασιακή λογική, έτσι και στην κατηγορηματική λογική υπάρχει ένα σύνολο ισοδυναμιών οι οποίες είναι ιδιαίτερα χρήσιμες για το μετασχηματισμό των τύπων της λογικής. Η κατηγορηματική λογική περιλαμβάνει όλες τις ισοδυναμίες της προτασιακής λογικής που παρουσιάστηκαν στον Πίνακα 3.5. Επιπλέον, στην κατηγορηματική λογική υπάρχουν οι ισοδυναμίες του Πίνακα 3.8 που αφορούν τους ποσοδείκτες.

Πίνακας 3.8: *Ισοδυναμίες της κατηγορηματικής λογικής*

	Ισοδυναμία	**Ονομασία - Σχόλια**
(8)	$\forall X(p(X)) \Leftrightarrow \neg\exists X\neg p(X)$	Νόμος de Morgan για τους

(9)	$\exists X(p(X)) \Leftrightarrow \neg\forall X\neg p(X)$	ποσοδείκτες
(10)	$\forall X(p(X)) \vee q \Leftrightarrow \forall X(p(X) \vee q)$	όπου το q δεν περιέχει ελεύθερες εμφανίσεις της μεταβλητής X
(11)	$\forall X(p(X)) \wedge q \Leftrightarrow \forall X(p(X) \wedge q)$	
(12)	$\exists X(p(X)) \vee q \Leftrightarrow \exists X(p(X) \vee q)$	
(13)	$\exists X(p(X)) \wedge q \Leftrightarrow \exists X(p(X) \wedge q)$	
(14)	$\forall X(p(X)) \Leftrightarrow \forall Y(p(Y))$	Μετονομασία μεταβλητών
(15)	$\exists X(p(X)) \Leftrightarrow \exists Y(p(Y))$	
(16)	$\forall X(p(X)) \wedge \forall X(q(X)) \Leftrightarrow \forall X(p(X) \wedge q(X))$	Επιμεριστική ιδιότητα του καθολικού ποσοδείκτη ως προς την σύζευξη
(17)	$\exists X(p(X)) \vee \exists X(q(X)) \Leftrightarrow \exists X(p(X) \vee q(X))$	Επιμεριστική ιδιότητα του υπαρξιακού ποσοδείκτη ως προς την διάζευξη

Οι ισοδυναμίες (14) και (15) δηλώνουν ότι η μετονομασία μεταβλητών σε ένα τύπο, στην εμβέλεια του αντίστοιχου ποσοδείκτη, είναι επιτρεπτή αρκεί να μην αλλάζει η σημασία του τύπου. Για παράδειγμα, στον τύπο $\forall X\exists Y\, \mu\eta\tau\acute{\varepsilon}\rho\alpha(Y,X)$ μπορούμε να μετονομάσουμε το X σε Z παίρνοντας τον τύπο $\forall Z\exists Y\, \mu\eta\tau\acute{\varepsilon}\rho\alpha(Y,Z)$, αλλά δεν είναι σωστό να μετονομαστεί το X σε Y καθώς προκύπτει ο τύπος $\forall Y\exists Y\, \mu\eta\tau\acute{\varepsilon}\rho\alpha(Y,Y)$ που έχει διαφορετική σημασία από τον αρχικό.

Ενδιαφέρον παρουσιάζουν και οι ισοδυναμίες (16) και (17). Η ισοδυναμία (16) ουσιαστικά δηλώνει ότι η σύζευξη καθολικά ποσοτικοποιημένων τύπων είναι ισοδύναμη με την καθολικά ποσοτικοποιημένη σύζευξή τους. Διαισθητικά, αν δύο εκφράσεις ισχύουν για κάθε στοιχείο του πεδίου, τότε για κάθε στοιχείο του πεδίου ισχύουν και οι δύο εκφράσεις.

Η ισοδυναμία (17) δηλώνει ότι η διάζευξη υπαρξιακά ποσοτικοποιημένων τύπων ισοδυναμεί με την υπαρξιακά ποσοτικοποιημένη διάζευξή τους. Διαισθητικά, αυτό γίνεται κατανοητό ως εξής: Αν υπάρχει τουλάχιστον ένα στοιχείο του πεδίου που καθιστά αληθή τον ένα τύπο ή υπάρχει τουλάχιστον ένα στοιχείο του πεδίου που καθιστά αληθή τον άλλο τύπο, τότε υπάρχει τουλάχιστον ένα στοιχείο του πεδίου που καθιστά αληθή ή τον ένα ή τον άλλο τύπο. Προφανώς το στοιχείο αυτό είναι τουλάχιστον αυτό που αληθεύει ή τον πρώτο τύπο είτε τον δεύτερο.

Η ισοδυναμία (16) δεν μπορεί να ισχύσει για τον υπαρξιακό ποσοδείκτη:

$$\exists X(p(X)) \wedge \exists X(q(X)) \not\Leftrightarrow \exists X(p(X) \wedge q(X))$$

Αυτό συμβαίνει γιατί το δεξιό μέρος της (μη-)ισοδυναμίας υπονοεί ότι οι δύο τύποι συναληθεύουν για το ίδιο X, ενώ στο αριστερό μέρος οι δύο σχέσεις μπορεί να συναληθεύουν για διαφορετικές τιμές του X.

Επίσης, η ισοδυναμία (17) δεν μπορεί να ισχύσει για τον καθολικό ποσοδείκτη:

$$\forall X(p(X)) \vee \forall X(q(X)) \not\Leftrightarrow \forall X(p(X) \vee q(X))$$

Αυτό συμβαίνει γιατί το δεξιό μέρος της (μη-)ισοδυναμίας μπορεί να αληθεύει όταν για όλα τα στοιχεία του πεδίου ισχύει ο ένας ή ο άλλος τύπος, ενώ για να αληθεύει το αριστερό μέρος πρέπει τουλάχιστον ο ένας από

τους δύο τύπους να αληθεύει για όλα τα στοιχεία του πεδίου. Αν λοιπόν, για παράδειγμα, για τα μισά στοιχεία του πεδίου αληθεύει ο ένας τύπος και για τα μισά ο άλλος, το δεξιό μέρος είναι αληθές, ενώ το αριστερό όχι.

Προσημασμένη συζευκτική κανονική μορφή

Το πλούσιο αλφάβητο της κατηγορηματικής λογικής επιτρέπει την ύπαρξη τύπων που αν και φαινομενικά διαφορετικοί, είναι στην ουσία λογικά ισοδύναμοι. Για παράδειγμα, οι ακόλουθοι τύποι αν και συντακτικά είναι ανόμοιοι είναι λογικά ισοδύναμοι:

$$\neg((\exists X)(p(X) \rightarrow q(X)))$$

$$(\forall X)(p(X) \wedge \neg q(X))$$

Όπως και στην προτασιακή λογική η δυνατότητα αναγωγής ενός τύπου σε μια περιορισμένη κανονική μορφή θα ήταν χρήσιμη για συγκρίσεις τέτοιων εκφράσεων, αλλά και για χρήση στις αποδεικτικές διαδικασίες. Μια τέτοια μορφή είναι η *προσημασμένη συζευκτική κανονική μορφή* (*prenex conjunctive normal form*), η μορφή της οποίας είναι

$$\forall X \exists Y\ (p(X) \wedge \neg q(X) \wedge (p(X) \vee \neg p(X) \vee q(X, Y)) \wedge \ldots \wedge (r(X,Y) \vee s(X)))$$

Τα βασικά δομικά στοιχεία της μορφής αυτής είναι τα *λεκτικά* στοιχεία (*literals*), τα οποία με τη σειρά τους συγκροτούν τις *φράσεις* (*clauses*), οι οποίες στη συνέχεια του κειμένου αναφέρονται ως προτάσεις. Ένα λεκτικό στοιχείο είναι ένας ατομικός τύπος ή η άρνηση ενός ατομικού τύπου. Μια φράση είναι μια πεπερασμένη διάζευξη (disjunction) κανενός ή περισσοτέρων λεκτικών στοιχείων. Για παράδειγμα, η έκφραση $p(X) \vee \neg p(X) \vee q(X, Y)$ είναι μια φράση, αφού είναι διάζευξη τριών λεκτικών στοιχείων. Η διάζευξη μηδέν στο πλήθος λεκτικών στοιχείων ονομάζεται κενή *φράση* ή *πρόταση* (*empty clause*) και αναπαρίσταται με το σύμβολο □. Ένας τύπος (formula) στη μορφή αυτή αποτελείται από μια σύζευξη φράσεων, η οποία είναι προσημασμένη από υπαρξιακούς και καθολικούς ποσοδείκτες στις μεταβλητές που εμφανίζονται στις φράσεις.

Οποιοσδήποτε τύπος της κατηγορηματικής λογικής μπορεί να αναχθεί σε έναν ισοδύναμο τύπο της προσημασμένης συζευκτικής κανονικής μορφής της λογικής. Η διαδικασία περιλαμβάνει τα ακόλουθα βήματα, σε κάθε ένα από τα οποία χρησιμοποιούνται οι αντίστοιχες ισοδυναμίες των Πινάκων 3.5 και 3.8:

1. Απαλοιφή των συνδετικών της ισοδυναμίας και συνεπαγωγής (ισοδυναμίες (6) και (7)).
2. Μετονομασία των μεταβλητών έτσι ώστε δύο μεταβλητές που ποσοτικοποιούνται από διαφορετικούς ποσοδείκτες να μην έχουν το ίδιο όνομα (ισοδυναμίες (14) και (15).
3. Μετατροπή των τύπων έτσι ώστε το συνδετικό της άρνησης να εφαρμόζεται μόνο σε ατομικούς τύπους (ισοδυναμίες (1), (2), (3), (8) και (9))
4. Μεταφορά των ποσοδεικτών με αναδρομική εφαρμογή των ισοδυναμιών (10)-(13).
5. Εφαρμογή των ισοδυναμιών επιμερισμού ως προς τη σύζευξη και διάζευξη έτσι ώστε ο τελικός τύπος να αποτελείται από συζεύξεις φράσεων (ισοδυναμίες (4) και (5)).

Στο παράδειγμα που ακολουθεί, ένας τύπος της κατηγορηματικής λογικής μετατρέπεται σε προσημασμένη συζευκτική μορφή:

$$\forall X\ (parent(X) \rightarrow \exists Y(hasChild(X,Y))) \wedge \neg(\exists Y(parent(Y) \wedge (programmer(Y) \vee doctor(Y)))$$

Το πρώτο βήμα αφορά στην απαλοιφή του συνδετικού της ισοδυναμίας που εμφανίζεται στην πρώτη σύζευξη (ισοδυναμία (6)):

$$\forall X\ (\neg parent(X) \vee \exists Y(hasChild(X,Y))) \wedge \neg(\exists Y(parent(Y) \wedge (programmer(Y) \vee doctor(Y)))$$

Επειδή η μεταβλητή Y εμφανίζεται ποσοτικοποιημένη από δύο διαφορετικούς ποσοδείκτες, η δεύτερή της εμφάνιση μετονομάζεται σε Z:

$$\forall X\ (\neg parent(X) \vee \exists Y(hasChild(X,Y))) \wedge \neg(\exists Z(parent(Z) \wedge (programmer(Z) \vee doctor(Z)))$$

Εφαρμογή των ισοδυναμιών DeMorgan έτσι ώστε να εφαρμόζεται η άρνηση μόνο σε ατομικούς τύπους:

$$\forall X\,(\neg parent(X) \vee \exists Y(hasChild(X,Y))) \wedge (\forall Z(\neg parent(Z) \vee (\neg programmer(Z) \wedge \neg doctor(Z)))$$

Εφαρμογή των ισοδυναμιών (10) και (12) για την κατάλληλη ομαδοποίηση των λεκτικών:

$$\forall X \exists Y \forall Z\,(\,(\neg parent(X) \vee hasChild(X,Y)) \wedge (\neg parent(Z) \vee (\neg programmer(Z) \wedge \neg doctor(Z)))\,)$$

Εφαρμογή της ισοδυναμίας (5) έτσι ώστε ο τελικός τύπος να αποτελείται από συζεύξεις προτάσεων:

$$\forall X \exists Y \forall Z\,((\neg parent(X) \vee hasChild(X,Y)) \wedge (\neg parent(Z) \vee \neg programmer(Z)) \wedge (\neg parent(Z) \vee \neg doctor(Z)))$$

Ο παραπάνω τύπος είναι σε προσημασμένη κανονική συζευκτική μορφή.

Κανονική μορφή κατά Skolem

Το ερώτημα που εγείρεται είναι αν είναι δυνατόν να υπάρξει κάποια κανονική μορφή στην οποία να εξαλείφονται πλήρως οι ποσοδείκτες. Μια τέτοια μορφή είναι η *κανονική μορφή κατά Skolem* που πήρε το όνομα της από τον Νορβηγό μαθηματικό Thoraf Skolem. Στην κανονική αυτή μορφή οι υπαρξιακά ποσοτικοποιημένες εμφανίσεις μεταβλητών αντικαθίστανται από σταθερές ή συναρτήσεις καθολικά ποσοτικοποιημένων μεταβλητών, μια διαδικασία που ονομάζεται *σκολεμοποίηση* (*skolemization*). Η μετατροπή ενός τύπου που είναι σε προσημασμένη συζευκτική μορφή σε κανονική μορφή κατά Skolem γίνεται με τον παρακάτω αλγόριθμο:

1. Έστω $\exists X_i$ η πρώτη από αριστερά υπαρξιακά ποσοτικοποιημένη μεταβλητή στον τύπο και $\forall X_1 ... \forall X_{i-1}$ οι καθολικά ποσοτικοποιημένες μεταβλητές του τύπου, μέσα στην εμβέλεια των οποίων βρίσκεται το $\exists X_i$, δηλαδή που βρίσκονται στα "αριστερά" του $\exists X_i$, τότε:
   a. Αν i=1, δηλαδή το πλήθος των καθολικών μεταβλητών ($X_1 ... X_{i-1}$) είναι μηδέν και δεν υπάρχουν καθολικά ποσοτικοποιημένες μεταβλητές στα αριστερά της X_i, τότε κάθε εμφάνιση της X_i στο τύπο αντικαθίσταται από μια νέα σταθερά Skolem sk_{Xi} (Skolem constant).
   b. Αν i>1, δηλαδή το πλήθος των καθολικών μεταβλητών είναι μεγαλύτερο του μηδενός, τότε κάθε εμφάνιση της μεταβλητής X_i στον τύπο αντικαθίσταται από μια νέα συνάρτηση Skolem (Skolem function) στις μεταβλητές $X_1 ... X_{i-1}$, $sk\_func_{Xi}(X_1,...,X_{i-1})$.
   c. Διέγραψε τον υπαρξιακό ποσοδείκτη $\exists X_i$ από τον τύπο.
2. Αν υπάρχουν άλλοι υπαρξιακοί ποσοδείκτες στον τύπο, τότε πήγαινε στο βήμα 1.
3. Διέγραψε όλους τους καθολικούς ποσοδείκτες.

Έστω για παράδειγμα, ο ακόλουθος τύπος σε προσημασμένη συζευκτική μορφή:

$$\forall X \exists Y \forall Z\,((\neg parent(X) \vee hasChild(X,Y)) \wedge (\neg parent(Z) \vee \neg programmer(Z)) \wedge (\neg parent(Z) \vee \neg doctor(Z)))$$

Στον τύπο υπάρχει μόνο ένας υπαρξιακός ποσοδείκτης ($\exists Y$) μέσα στην εμβέλεια του καθολικού ποσοδείκτη $\forall X$. Ο ποσοδείκτης αντικαθίσταται από τη συνάρτηση Skolem $sk\_function_Y(X)$, όπως φαίνεται παρακάτω:

$$\forall X \forall Z\,(\,(\neg parent(X) \vee hasChild(X,sk\_function_Y(X))) \wedge (\neg parent(Z) \vee \neg programmer(Z)) \wedge (\neg parent(Z) \vee \neg doctor(Z))\,)$$

Εφόσον δεν υπάρχουν άλλοι υπαρξιακοί ποσοδείκτες, διαγράφονται όλοι οι καθολικοί ποσοδείκτες από τον προηγούμενο τύπο:

$$(\,(\neg parent(X) \vee hasChild(X,sk\_function_Y(X))) \wedge (\neg parent(Z) \vee \neg programmer(Z)) \wedge (\neg parent(Z) \vee \neg doctor(Z))\,)$$

Όπως είναι προφανές, η συνάρτηση Skolem δεν εξαρτάται από τη μεταβλητή Z αλλά μόνο από τη X. Θα πρέπει να σημειωθεί ότι η μετατροπή ενός τύπου σε μορφή Skolem διατηρεί τη *μη-ικανοποιησιμότητα* (*unsatisfiability*), δηλαδή ο αρχικός τύπος είναι μη-ικανοποιήσιμος αν και μόνο αν η μορφή του κατά Skolem είναι μη-ικανοποιήσιμη και ανάστροφα. Αυτή η ιδιότητα είναι ιδιαίτερα σημαντική, καθώς ένα σύνολο προτάσεων S, που είναι μη-ικανοποιήσιμο, παραμένει μη ικανοποιήσιμο και στην Skolem μορφή του. Το

τελευταίο σημαίνει ότι αν χρησιμοποιηθεί μια αποδεικτική διαδικασία βασισμένη στην εις άτοπο απαγωγή, μπορούν να αποδειχθούν όλες οι προτάσεις που θα αποδεικνύονταν αν είχε επιλεγεί η κλασική μορφή της κατηγορηματικής λογικής ως μέθοδος αναπαράστασης.

Προτασιακή μορφή της κατηγορηματικής λογικής

Ο τελευταίος μετασχηματισμός αφορά στη μετατροπή ενός τύπου κανονικής μορφής σε ένα σύνολο προτάσεων - φράσεων, δηλαδή στην *προτασιακή (ή φραστική) μορφή* της λογικής (*clausal form*). Ο όρος *προτασιακή μορφή της λογικής* αποτελεί μετάφραση του *clausal form of predicate logic* και δεν θα πρέπει να συγχέεται με την προτασιακή λογική (propositional logic).

Ο μετασχηματισμός βασίζεται στον κανόνα της απαλοιφής σύζευξης. Έτσι, ένας τύπος της μορφής $p \wedge q \wedge r \wedge s \wedge t$ μετατρέπεται στο σύνολο των τύπων $\{p, q, r, s, t\}$, όπου καθένας από αυτούς αποτελείται αποκλειστικά από διαζεύξεις ατομικών προτάσεων ή αρνήσεις αυτών, δηλαδή *προτάσεις ή φράσεις* (*clauses*). Για παράδειγμα, ο τύπος:

$$((\neg parent(X) \vee hasChild(X,sk\_function_Y(X))) \wedge (\neg parent(Z) \vee \neg programmer(Z)) \wedge (\neg parent(Z) \vee \neg doctor(Z)))$$

μετατρέπεται στις προτάσεις-φράσεις:

$$\neg parent(X) \vee hasChild(X,sk\_function_Y(X))$$

$$\neg parent(Z) \vee \neg programmer(Z)$$

$$\neg parent(Z) \vee \neg doctor(Z)$$

Η διαδικασία μετατροπής ενός τύπου σε προτασιακή μορφή περιλαμβάνει επίσης και ένα βήμα μετονομασίας των μεταβλητών σε περίπτωση που δύο προτάσεις έχουν μεταβλητές με κοινά ονόματα, προκειμένου να μην συγχέονται οι μεταβλητές αυτές μεταξύ τους σε μία αποδεικτική διαδικασία. Για παράδειγμα, στις δύο τελευταίες προτάσεις παραπάνω υπάρχει κοινή μεταβλητή Z. Αν μετονομαστεί η Z σε W στην τρίτη πρόταση, τότε προκύπτει η πρόταση: $\neg parent(W) \vee \neg doctor(W)$.

Μορφή Kowalski

Μια εναλλακτική μορφή της προτασιακής λογικής με ιδιαίτερο ενδιαφέρον είναι η μορφή Kowalski. Σε αυτήν όλες οι προτάσεις εκφράζονται ως λογικές ισοδυναμίες της μορφής:

$$q_1, q_2, \ldots, q_n \rightarrow r_1, r_2, \ldots, r_m$$

Σε αυτήν την έκφραση οι ατομικοί τύποι r_i βρίσκονται σε διάζευξη, ενώ οι q_j σε σύζευξη. Τα r_i αποτελούν τα *συμπεράσματα* της πρότασης, ενώ τα q_j τις *υποθέσεις* της. Τόσο τα συμπεράσματα όσο και οι υποθέσεις είναι ατομικοί τύποι και όχι λεκτικά στοιχεία, δηλαδή δεν περιέχουν αρνήσεις ατομικών τύπων.

Η διαδικασία μετατροπής μιας πρότασης σε μορφή Kowalski είναι εξαιρετικά απλή. Για παράδειγμα, έστω η πρόταση:

$$p \vee \neg q \vee r \vee \neg s \vee t$$

Το πρώτο βήμα αφορά στη συγκέντρωση όλων των ατομικών τύπων σε άρνηση στο αριστερό μέρος της πρότασης, με εφαρμογή της αντιμεταθετικής ιδιότητας:

$$\neg q \vee \neg s \vee p \vee r \vee t$$

Εφαρμογή του νόμου de Morgan στο σύνολο των αρνητικών λεκτικών:

$$\neg(q \wedge s) \vee p \vee r \vee t$$

Εφαρμογή της ισοδυναμίας συνεπαγωγής - διάζευξης $(p \rightarrow q) \Leftrightarrow \neg p \vee q$:

$$q \wedge s \rightarrow p \vee r \vee t$$

Το τελευταίο βήμα αφορά στην αντικατάσταση των συμβόλων της σύζευξης και της διάζευξης με το σύμβολο ",".

$q, s \rightarrow p, r, t$

Για παράδειγμα, οι προτάσεις:

$\neg parent(X,Y) \vee \neg male(X) \vee father(X,Y)$

$\neg parent(X,Y) \vee father(X,Y) \vee mother(X,Y)$

μετατρέπονται στην μορφή Kowaski στις ακόλουθες προτάσεις:

$parent(X,Y), male(X) \rightarrow father(X,Y)$

$parent(X,Y) \rightarrow father(X,Y), mother(X,Y)$

Το πλεονέκτημα της μορφή Kowalski είναι ότι η γνώση εκφράζεται σε μια περισσότερο αναγνώσιμη μορφή, καθώς οι συνεπαγωγές είναι περισσότερο κατανοητές από τις διαζεύξεις και τις αρνήσεις.

Σε μια πρόταση Kowalski της μορφής:

$q_1, q_2, \ldots, q_n \rightarrow r_1, r_2, \ldots, r_m$

υπάρχουν οι ακόλουθες περιπτώσεις:

- Αν $m > 0$ και $n > 0$, τότε η πρόταση ερμηνεύεται σαν "ισχύει r_1 ή r_2…ή r_m εάν q_1 και q_2 …και q_n"
- Αν m=0, τότε οι υποθέσεις καταλήγουν σε αναληθή συμπέρασμα, που ουσιαστικά ισοδυναμεί με άρνηση της σύζευξης, δηλαδή ερμηνεύεται σαν "*Δεν ισχύει* q_1 *και* q_2 *και* $\ldots q_n$":

  $q_1, q_2, \ldots, q_n \rightarrow$

- Αν n=0, τότε αναπαριστάται μια πρόταση χωρίς υπόθεση, δηλαδή κάποιο από τα r_i ισχύει:

  $\rightarrow r_1, r_2, \ldots, r_m$

- Αν m=0 και n=0, τότε αναπαριστάται μια πρόταση χωρίς συμπέρασμα και υποθέσεις που δηλώνει πρόταση πάντα αναληθή και συμβολίζεται με την κενή πρόταση □.

Στο παράδειγμα της προηγούμενης υποενότητας, προέκυψαν οι ακόλουθες προτάσεις (clauses) στην προτασιακή μορφή της κατηγορηματικής λογικής:

$\neg parent(X) \vee hasChild(X, sk\_function_Y(X))$

$\neg parent(Z) \vee \neg programmer(Z)$

$\neg parent(W) \vee \neg doctor(W)$

Αυτές, μετατρέπονται στις ακόλουθες προτάσεις στην μορφή Kowalski:

$parent(X) \rightarrow hasChild(X, sk\_function_Y(X))$

$parent(Z) \wedge programmer(Z) \rightarrow$

$parent(W) \wedge doctor(W) \rightarrow$

Η προτασιακή μορφή της λογικής και η μορφή Kowalski είναι ιδιαίτερα χρήσιμη στην απόδειξη θεωρημάτων, όπως θα φανεί στη συνέχεια.

Προτάσεις Horn

Μια ειδική περίπτωση προτάσεων της μορφής Kowalski με μεγάλη σημασία σε πρακτικά συστήματα, όπως για παράδειγμα στο λογικό προγραμματισμό (Prolog), είναι οι *προτάσεις Horn* (*Horn clauses*). Στις προτάσεις Horn, επιτρέπεται μόνο ένας ατομικός τύπος στο συμπέρασμα, είναι δηλαδή της μορφής:

$q_1, q_2, \ldots, q_n \rightarrow r$

Η παραπάνω μορφή, όταν n>0, ονομάζεται συνήθως *κανόνας* και ερμηνεύεται ως εξής: "*εάν ισχύει η συνθήκη, τότε ισχύει και το συμπέρασμα*". Σε αντίθεση με την γενική μορφή Kowalski, το συμπέρασμα είναι ένα και συγκεκριμένο, άρα μπορεί να έχει πρακτική σημασία, ενώ στην γενική μορφή των προτάσεων

Kowalski το συμπέρασμα είναι μια διάζευξη ατομικών τύπων, συνεπώς υπάρχει ασάφεια σχετικά με το ποιος ή ποιοι από τους ατομικούς τύπους αληθεύουν.

Όταν n=0, η πρόταση Kowalski που προκύπτει είναι η:

$$\rightarrow r$$

η οποία ερμηνεύεται ως "*ισχύει το r*" και συνήθως ονομάζεται *γεγονός* (*fact*). Και στην περίπτωση των γεγονότων, η σημασία τους είναι πιο πρακτική από την γενική μορφή Kowalski, καθώς εκφράζει συγκεκριμένη γνώση που αληθεύει.

Οι προτάσεις στην μορφή Kowalski που προέκυψαν στο παράδειγμα της προηγούμενης υποενότητας είναι προτάσεις Horn, καθώς έχουν το πολύ έναν ατομικό τύπου στο συμπέρασμα.

Οι προτάσεις Horn αποτελούν την πλέον χρησιμοποιούμενη μορφή της λογικής, καθώς αποτελούν ένα αρκετά απλό αλλά παράλληλα αρκετά εκφραστικό τρόπο αναπαράστασης. Θα πρέπει όμως να σημειωθεί ότι δεν είναι δυνατό να μετατραπεί οποιαδήποτε έκφραση της κατηγορηματικής λογικής σε προτάσεις Horn και συνεπώς αποτελούν μια πιο περιορισμένη μορφή της λογικής.

Παραδείγματα εφαρμογής

Θα χρησιμοποιήσουμε τις προτάσεις (1α), (2α), (3α) οι οποίες παρουσιάστηκαν στο τέλος της προηγούμενης ενότητας.

Πρώτα θα μετατρέψουμε τις προτάσεις σε προσημασμένη συζευκτική κανονική μορφή. Όταν παραλείπονται βήματα σημαίνει ότι δεν έχουν εφαρμογή στην συγκεκριμένη περίπτωση.

// *βήμα 1* $\exists X\,(student(X) \wedge \forall Y\,(course(Y) \rightarrow attends(X, Y))) \Leftrightarrow$ (1α)

// *βήμα 4* $\exists X\,(student(X) \wedge \forall Y\,(\neg course(Y) \vee attends(X, Y))) \Leftrightarrow$

$\exists X \forall Y\,(student(X) \wedge (\neg course(Y) \vee attends(X, Y)))$ (1β)

// *βήμα 1* $\forall X\,(student(X) \rightarrow \forall Y\,(\,(course(Y) \wedge boring(Y)) \rightarrow \neg attends(X, Y))) \Leftrightarrow$ (2α)

// *βήμα 3* $\forall X\,(\neg student(X) \vee \forall Y\,(\,\neg(course(Y) \wedge boring(Y)) \vee \neg attends(X, Y))) \Leftrightarrow$

// *βήμα 4* $\forall X\,(\neg student(X) \vee \forall Y\,(\,\neg course(Y) \vee \neg boring(Y) \vee \neg attends(X, Y))) \Leftrightarrow$

$\forall X \forall Y\,(\neg student(X) \vee \neg course(Y) \vee \neg boring(Y) \vee \neg attends(X, Y))$ (2β)

// *βήμα 1* $\forall X\,((course(X) \wedge teaches(petros,X)) \rightarrow \forall Z\,(student(Z) \rightarrow attends(Z,X))) \Leftrightarrow$ (3α)

// *βήμα 3* $\forall X\,(\neg(course(X) \wedge teaches(petros,X)) \vee \forall Z\,(\neg student(Z) \vee attends(Z,X))) \Leftrightarrow$

// *βήμα 4* $\forall X\,(\neg course(X) \vee \neg teaches(petros,X) \vee \forall Z\,(\neg student(Z) \vee attends(Z,X))) \Leftrightarrow$

$\forall X \forall Z\,(\neg course(X) \vee \neg teaches(petros,X) \vee \neg student(Z) \vee attends(Z,X))$ (3β)

Οι προτάσεις (1β), (2β), (3β) βρίσκονται σε προσημασμένη συζευκτική κανονική μορφή. Μόνο η πρόταση (1β) θα μετατραπεί σε κανονική μορφή Skolem, καθώς είναι η μόνη που περιέχει υπαρξιακό ποσοδείκτη. Επειδή ο υπαρξιακός ποσοδείκτης δεν βρίσκεται στην εμβέλεια κάποιου καθολικού ποσοδείκτη θα αντικατασταθεί από σταθερά Skolem:

$$\forall Y\,(student(sk_X) \wedge (\neg course(Y) \vee attends(sk_X, Y))) \quad (1\gamma)$$

Για την μετατροπή των προτάσεων (1γ), (2β), (3γ) στην προτασιακή μορφή της κατηγορηματικής λογικής, διώχνουμε τους καθολικούς ποσοδείκτες και "διασπάμε" τις προτάσεις που έχουν σύζευξη σε ξεχωριστές προτάσεις (clauses):

$$student(sk_X) \quad (1\gamma.1)$$

$$\neg course(Y) \vee attends(sk_X, Y) \quad (1\gamma.2)$$

$$\neg student(X) \vee \neg course(Y) \vee \neg boring(Y) \vee \neg attends(X, Y) \quad (2\gamma)$$

$$\neg course(X) \vee \neg teaches(petros, X) \vee \neg student(Z) \vee attends(Z, X) \quad (3\gamma)$$

Τέλος, μετονομάζουμε τις μεταβλητές σε κάθε πρόταση:

$$student(sk_X) \quad (1\gamma.1)$$

$$\neg course(Y) \vee attends(sk_X, Y) \quad (1\gamma.2)$$

$$\neg student(X) \vee \neg course(W) \vee \neg boring(W) \vee \neg attends(X, W) \quad (2\gamma)$$

$$\neg course(R) \vee \neg teaches(petros, R) \vee \neg student(Z) \vee attends(Z, R) \quad (3\gamma)$$

Οι παραπάνω προτάσεις μετασχηματίζονται ως εξής στην μορφή Kowalski:

$$\rightarrow student(sk_X) \quad (1\delta.1)$$

$$course(Y) \rightarrow attends(sk_X, Y) \quad (1\delta.2)$$

$$student(X) \wedge course(W) \wedge boring(W) \wedge attends(X, W) \rightarrow \quad (2\delta)$$

$$course(R) \wedge teaches(petros, R) \wedge student(Z) \rightarrow attends(Z, R) \quad (3\delta)$$

Όλες οι παραπάνω προτάσεις είναι προτάσεις Horn, καθώς στα συμπεράσματα των κανόνων υπάρχει το πολύ ένας ατομικός τύπος.

Μηχανισμός Εξαγωγής Συμπερασμάτων

Ο βασικός μηχανισμός εξαγωγής συμπερασμάτων στην κατηγορηματική λογική είναι η απόδειξη. Υπάρχει ένα πλήθος κανόνων συμπερασμού που μπορεί να χρησιμοποιηθούν σε μια αποδεικτική διαδικασία για την εξαγωγή συμπερασμάτων από γνώση που είναι εκφρασμένη σε κατηγορηματική λογική. Αυτές συμπεριλαμβάνουν τους κανόνες συμπερασμού της προτασιακής λογικής που παρουσιάστηκαν στον Πίνακα 3.6, καθώς και δύο νέους κανόνες συμπερασμού που αφορούν προτάσεις που περιέχουν ποσοτικοποιημένες μεταβλητές, όπως φαίνεται στον Πίνακα 3.9.

Πίνακας 3.9: *Κανόνες συμπερασμού της κατηγορηματικής λογικής.*

	Κανόνας Συμπερασμού	**Ονομασία**
(7)	$\forall X p(X) \vdash p(a)\ \theta=\{X/a\}$	απαλοιφή καθολικού ποσοδείκτη (universal elimination)
(8)	$p(a) \vdash \exists X p(X)\ \theta=\{X/a\}$	εισαγωγή υπαρξιακού ποσοδείκτη (existential introduction)

Όλοι οι κανόνες βασίζονται στην διαδικασία της αντικατάστασης μεταβλητών και ενοποίησης ατομικών τύπων και όρων που παρουσιάζεται παρακάτω.

Αντικατάσταση και Ενοποίηση

Η ύπαρξη μεταβλητών κάνει απαραίτητη την εισαγωγή δύο εννοιών: την *αντικατάσταση* (*substitution*) και την *ενοποίηση* (*unification*). Η "*αντικατάσταση*" αφορά στην αντικατάσταση των μεταβλητών που εμφανίζονται σε έναν τύπο από κάποιους όρους. Μια αντικατάσταση αναπαριστάται με το σύνολο $\{X_i/t_i\}$ όπου X_i η μεταβλητή που θα αντικατασταθεί και t_i ο όρος με τον οποίο θα αντικατασταθεί. Για παράδειγμα, η αντικατάσταση $\{X/πέτρος\}$ στον ακόλουθο τύπο:

είναι(X, προγραμματιστής)

θα δώσει τον τύπο:

είναι(πέτρος, προγραμματιστής)

Ενοποίηση (*unification*) είναι η διαδικασία κατά την οποία δύο εκφράσεις γίνονται συντακτικά όμοιες με τη χρήση αντικαταστάσεων. Για παράδειγμα, οι ακόλουθες προτάσεις:

είναι(πέτρος, προγραμματιστής, X)

είναι(πέτρος, Y, prolog)

ενοποιούνται με την αντικατάσταση θ={*X/prolog*, *Y/προγραμματιστής*}.

Ο *ενοποιητής* (*unifier*) δύο εκφράσεων φ_1 και φ_2, είναι μια αντικατάσταση θ τέτοια ώστε η έκφραση $\varphi_1\theta$ να είναι συντακτικά όμοια με τη $\varphi_2\theta$. Αν υπάρχει μια τέτοια αντικατάσταση οι εκφράσεις φ_1 και φ_2 ονομάζονται *ενοποιήσιμες* (*unifiable*). Υπάρχουν πολλοί δυνατοί ενοποιητές ανάμεσα σε δύο ή περισσότερες εκφράσεις, όμως εκείνος που παρουσιάζει ενδιαφέρον είναι ο *γενικότερος ενοποιητής* (*mgu-most general unifier*), ο οποίος ενοποιεί τις εκφράσεις με τις λιγότερες δυνατές αντικαταστάσεις και είναι μοναδικός.

Η εύρεση του γενικότερου ενοποιητή ανάμεσα σε δύο εκφράσεις βρίσκεται εύκολα με τον ακόλουθο αναδρομικό αλγόριθμο:

1. Δύο σταθερές ενοποιούνται αν και μόνο αν είναι ίδιες.
2. Μια μεταβλητή ενοποιείται με οποιονδήποτε όρο, εισάγοντας μια νέα αντικατάσταση στο γενικότερο ενοποιητή.
3. Δύο συναρτησιακοί όροι ενοποιούνται αν έχουν το ίδιο συναρτησιακό σύμβολο, την ίδια τάξη (αριθμό ορισμάτων) και αν κάθε όρισμα του πρώτου μπορεί να ενοποιηθεί με το αντίστοιχο σε θέση όρισμα του δεύτερου όρου.
4. Δύο ατομικοί τύποι ενοποιούνται αν έχουν το ίδιο κατηγόρημα, την ίδια τάξη (αριθμό ορισμάτων) και αν κάθε όρισμα του πρώτου μπορεί να ενοποιηθεί με το αντίστοιχο σε θέση όρισμα του δεύτερου ατομικού τύπου.

Ο παραπάνω αλγόριθμος, αν και είναι αποδοτικός και χρησιμοποιείται σχεδόν σε όλα τα συστήματα λογικού προγραμματισμού που βασίζονται στην κατηγορηματική λογική πρώτης τάξης, όπως η Prolog, είναι μη ορθός. Η μη ορθότητά του έγκειται στις περιπτώσεις όπου η προς ενοποίηση μεταβλητή εμφανίζεται στον όρο με τον οποίο θα ενοποιηθεί. Για παράδειγμα, η ενοποίηση *X=προϊστάμενος*(*X*) θα δώσει *X=προϊστάμενος*(*προϊστάμενος*(*προϊστάμενος*(...))), δηλαδή δημιουργεί απειρία αντικαταστάσεων. Το πρόβλημα αυτό αναφέρεται σαν *έλεγχος εμφάνισης* (*occurs check*) και η αποφυγή του απαιτεί τη χρήση ενός αλγορίθμου με μεγάλο υπολογιστικό κόστος. Για τον λόγο αυτό συνήθως δεν υλοποιείται ή δεν χρησιμοποιείται στις υλοποιήσεις της γλώσσας Prolog και η ορθότητα των προγραμμάτων επαφίεται στον προγραμματιστή, με κέρδος την αποδοτικότητα.

Οι έννοιες της αντικατάστασης και της ενοποίησης είναι σημαντικές για την εφαρμογή των κανόνων συμπερασμού της κατηγορηματικής λογικής και την εξαγωγή νέας γνώσης, όπως παρουσιάζεται στη συνέχεια. Τέλος, η επέκταση της σύνταξης της λογικής με μεταβλητές και ποσοδείκτες αυξάνει σημαντικά την εκφραστική της ικανότητα, καθώς επιτρέπει την αναπαράσταση γενικής γνώσης.

Στον παρακάτω πίνακα (Πίνακας 3.10) παρουσιάζονται διάφορα παραδείγματα ενοποίησης ατομικών τύπων, επιτυχημένων και μη, καθώς και των αντίστοιχων γενικών ενοποιητών.

Πίνακας 3.10 *Παραδείγματα ενοποίησης ατομικών τύπων*

Ατομικός τύπος 1	**Ατομικός τύπος 2**	**Ενοποιητής**
πατέρας(*νίκος*, *X*)	*πατέρας*(*Y*, *θανάσης*)	{*Y/νίκος*, *X/θανάσης*}
επάγγελμα(*X*, *μισθός*(*υψηλός*), *ωράριο*(12))	*επάγγελμα*(*νίκος*, *Y*, *Z*)	{*X/νίκος*, *Y/μισθός*(*υψηλός*), *Z/ωράριο*(12)}

μηχανή(μέρος(*έμβολο*), *λειτουργεί*)	*μηχανή*(*X*, *βλάβη*)	**Δεν ενοποιείται**
υπάλληλος(*κώστας*, *διευθυντής*(*κώστας*))	*υπάλληλος*(*κώστας*, *νίκος*)	**Δεν ενοποιείται**
υπάλληλος(*κώστας*, *διευθυντής*(*κώστας*))	*υπάλληλος*(*X*, *διευθυντής*(*X*))	{*X*/*κώστας*}
υπάλληλος(*κώστας*, *Y*)	*υπάλληλος(X, διευθυντής(X))*	{*X*/*κώστας*, *Y*/*διευθυντής*(*κώστας*))}
υπάλληλος(*κώστας*, *διευθυντής*(*νίκος*))	*υπάλληλος*(*X*, *διευθυντής*(*X*))	**Δεν ενοποιείται**

Γενικευμένος τρόπος του θέτειν

Ο τρόπος του θέτειν μπορεί να χρησιμοποιηθεί και στην περίπτωση της κατηγορηματικής λογικής. Για παράδειγμα, έστω ότι η βάση γνώσης περιέχει τις προτάσεις:

$$\forall X\ (\text{πληροφορικός}(X) \rightarrow \text{προγραμματιστής}(X)) \quad (1)$$

$$\text{πληροφορικός}(\text{πέτρος}) \quad (2)$$

Σύμφωνα με τον κανόνα συμπερασμού (7) (Πίνακας 3.6), και την αντικατάσταση θ={*X*/*πέτρος*} είναι δυνατό να εξαχθεί το συμπέρασμα:

$$\text{πληροφορικός}(\text{πέτρος}) \rightarrow \text{προγραμματιστής}(\text{πέτρος}) \quad (3)$$

Με εφαρμογή του κανόνα "*τρόπος του θέτειν*" (*modus ponens*) στις προτάσεις (2) και (3), εξάγεται η πρόταση:

$$\text{προγραμματιστής}(\text{πέτρος}) \quad (4)$$

Οι προηγούμενες εφαρμογές των κανόνων συμπερασμού της απαλοιφής του καθολικού ποσοδείκτη και του "τρόπου του θέτειν" μπορεί να συνδυαστούν σε ένα μόνο κανόνα συμπερασμού που ονομάζεται "*γενικευμένος τρόπος του θέτειν*" (*generalized modus ponens*):

$$p_1', p_2', \ldots, p_n', p_1' \wedge p_2' \wedge \ldots \wedge p_n' \rightarrow q \vdash \theta(q)$$

όπου θ το σύνολο των αντικαταστάσεων οι οποίες κάνουν τα p'_i και p_i συντακτικά όμοια, όπως προκύπτει από μια διαδικασία ενοποίησης των τύπων. Με τη χρήση του *γενικευμένου τρόπου του θέτειν* η εξαγωγή του συγκεκριμένου συμπεράσματος γίνεται σε ένα βήμα:

$$\forall X\ (\text{πληροφορικός}(X) \rightarrow \text{προγραμματιστής}(X)),\ \text{πληροφορικός}(\text{πέτρος})$$

$$\vdash \text{προγραμματιστής}(\text{πέτρος}),\ \theta=\{X/\text{πέτρος}\}$$

Μια αποδεικτική διαδικασία που χρησιμοποιεί ως μοναδικό κανόνα συμπερασμού τον γενικευμένο τρόπο του θέτειν είναι ορθή, αλλά στη γενική περίπτωση, δεν είναι πλήρης. Μια τέτοια διαδικασία είναι πλήρης μόνο αν η βάση γνώσης αποτελείται από προτάσεις Horn. Το τελευταίο αποτελεί σημαντικό περιορισμό καθώς, όπως προαναφέρθηκε, δεν είναι δυνατό όλες οι προτάσεις της κατηγορηματικής λογικής να μετατραπούν σε προτάσεις Horn. Έτσι είναι αναγκαία η χρήση περισσότερων του ενός κανόνων συμπερασμού για την απόδειξη προτάσεων στην κατηγορηματική λογική, γεγονός που έκανε την αυτοματοποιημένη εύρεση απόδειξης μια πολύπλοκη διαδικασία. Το πρόβλημα λύθηκε από τον Robinson στη δεκαετία του 60, με την εισαγωγή της αρχής της ανάλυσης.

Η αρχή της ανάλυσης

Η σπουδαιότητα της αρχής της ανάλυσης εντοπίζεται στο γεγονός ότι είναι ο μοναδικός κανόνας που απαιτείται για την εξαγωγή όλων των ορθών συμπερασμάτων σε μια αποδεικτική διαδικασία που

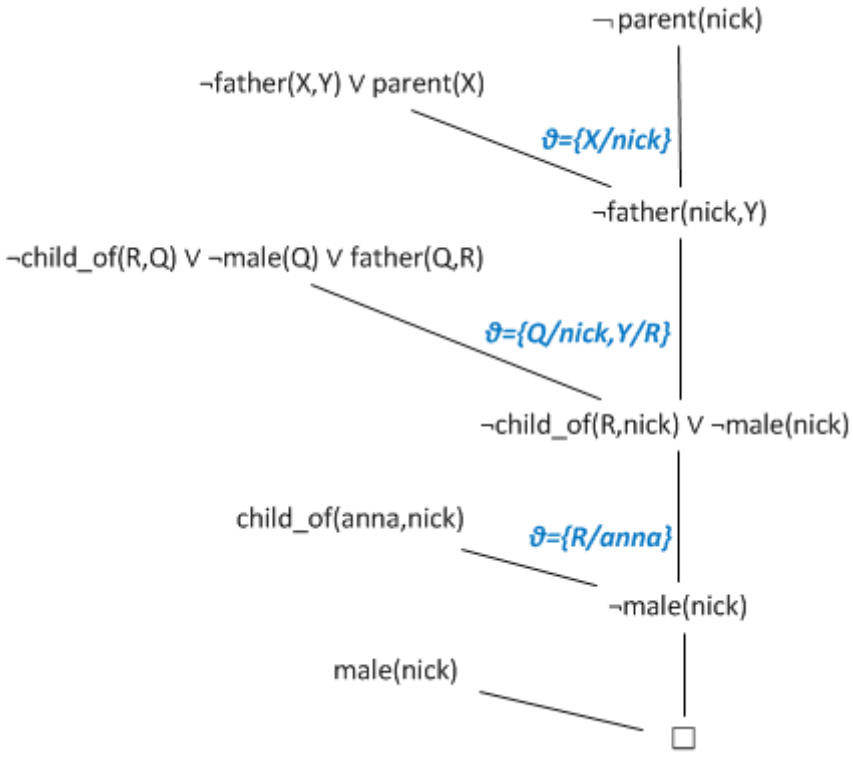
χρησιμοποιεί τη μέθοδο της "*εις άτοπο απαγωγής*" (*refutation*). Οπότε, η διαδικασία απόδειξης που προκύπτει είναι ορθή και πλήρης, όπως και η αντίστοιχη της προτασιακής λογικής.

Στην απλή περίπτωση, η *αρχή της ανάλυσης* (*resolution*) διατυπώνεται με προτάσεις που περιέχουν δύο λεκτικά στοιχεία (literals):

$$p \vee \neg q,\ z \vee q' \vdash \theta(p \vee z)$$

Τα λεκτικά στοιχεία q' και $\neg q$ ονομάζονται συμπληρωματικά ζεύγη και δεν είναι ανάγκη να είναι "απολύτως" όμοια, αλλά να ενοποιούνται με κατάλληλες αντικαταστάσεις (ενοποιητής θ). Οι αντικαταστάσεις μεταβλητών που προκύπτουν εφαρμόζονται στο αναλυθέν (resolvent) $p \vee z$. Όπως είναι φανερό, οι διαδικασίες αντικατάστασης και ενοποίησης αποτελούν βασικά εργαλεία στην εφαρμογή της συγκεκριμένης αποδεικτικής διαδικασίας.

Όπως και στην προτασιακή λογική, η διαδικασία απόδειξης περιλαμβάνει την εισαγωγή της άρνησης της προς απόδειξης πρότασης στο αρχικό σύνολο προτάσεων που αποτελεί τη γνώση και εφαρμογή του κανόνα της ανάλυσης μέχρι το σύστημα να εξαγάγει την κενή πρόταση (άτοπο), η οποία αναπαρίσταται με το σύμβολο □.



Σχήμα 3.1:*Απόδειξη κατηγορηματικής λογικής βασισμένη στην αρχή της ανάλυσης*

Για παράδειγμα, έστω ότι θέλουμε να διερευνήσουμε αν κάποιος άνθρωπος (nick) είναι γονέας (parent), βάσει του ακόλουθου συνόλου προτάσεων:

child_of(anna,nick)

male(nick)

$\neg father(X,Y) \vee parent(X)$

$\neg mother(W,Z) \vee parent(W)$

$\neg child\_of(R,Q) \vee \neg male(Q) \vee father(Q,R)$

Αρχικά εισάγεται η άρνηση της πρότασης *parent*(*nick*) στο σύνολο των προτάσεων και στη συνέχεια εφαρμόζεται διαδοχικά ο κανόνας της ανάλυσης. Όπως φαίνεται και στο Σχήμα 3.1, για τη συγκεκριμένη απόδειξη απαιτείται μόνο ένα υποσύνολο των προτάσεων της γνώσης.

Από το προηγούμενο παράδειγμα φαίνεται ότι, για να είναι δυνατή η εφαρμογή της αρχής της ανάλυσης, η γνώση του συστήματος πρέπει να είναι εκφρασμένη σε μορφή διαζεύξεων, δηλαδή στην προτασιακή μορφή της κατηγορηματικής λογικής. Όπως παρουσιάστηκε, η μορφή αυτή δεν είναι απόλυτα ισοδύναμη με τη "γενική" μορφή της λογικής, αλλά διατηρεί μόνο τη μη-αποδειξιμότητα του αρχικού συνόλου προτάσεων. Το τελευταίο δεν αποτελεί πρόβλημα, καθώς η αποδεικτική διαδικασία είναι η "*απαγωγή σε άτοπο*", άρα χρησιμοποιώντας την προτασιακή μορφή της λογικής μπορούμε να εξάγουμε το ίδιο ακριβώς σύνολο συμπερασμάτων.

Αρχή της ανάλυσης και μορφή Kowalski

Μια εναλλακτική διατύπωση του κανόνα της ανάλυσης προκύπτει με την χρήση της μορφής Kowalski και πιο συγκεκριμένα με την χρήση των προτάσεων Horn:

$$p_1,p_2,\ldots,p_n \rightarrow q,\ q',z_1,z_2,\ldots,z_n \rightarrow s \ \vdash \theta(p_1,p_2,\ldots,p_n,z_1,z_2,\ldots,z_n \rightarrow s)$$

όπου θ ο ενοποιητής των q και q'. Το πλεονέκτημα χρήσης της μορφής Kowalski είναι ότι η γνώση και όλη η διαδικασία είναι περισσότερο κατανοητές. Για παράδειγμα, το προηγούμενο πρόβλημα σε μορφή Kowalski γράφεται:

$\rightarrow child\_of(anna,nick)$

$\rightarrow male(nick)$

$father(X,Y) \rightarrow parent(X)$

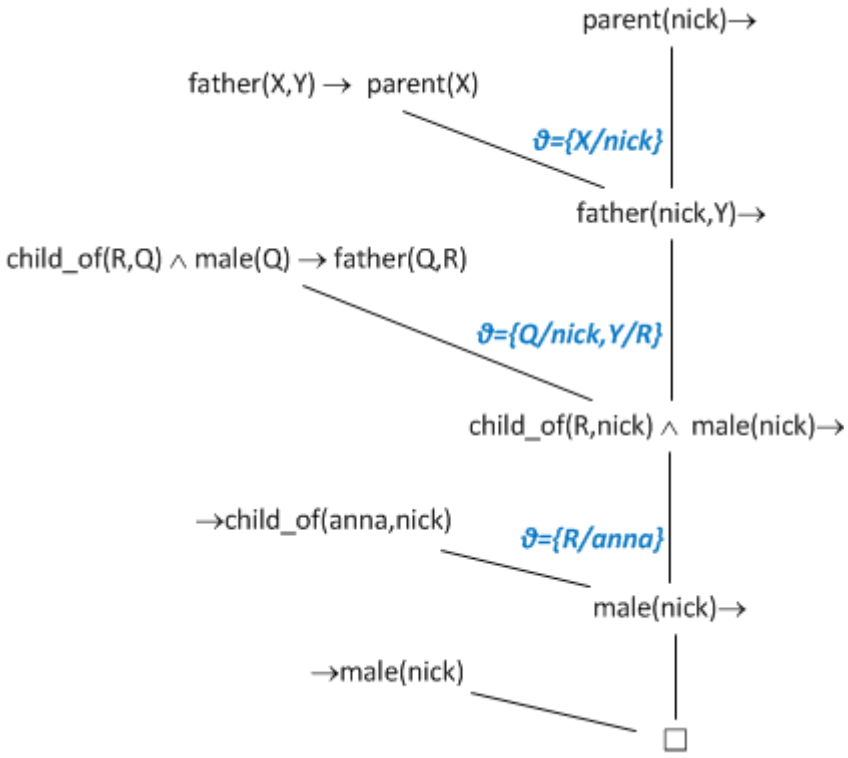
$mother(W,Z) \rightarrow parent(W)$

$child\_of(R,Q) \wedge male(Q) \rightarrow father(Q,R)$

Η άρνηση της προς απόδειξη πρότασης αναπαριστάται στη μορφή Kowalski ως:

$parent(nick) \rightarrow$

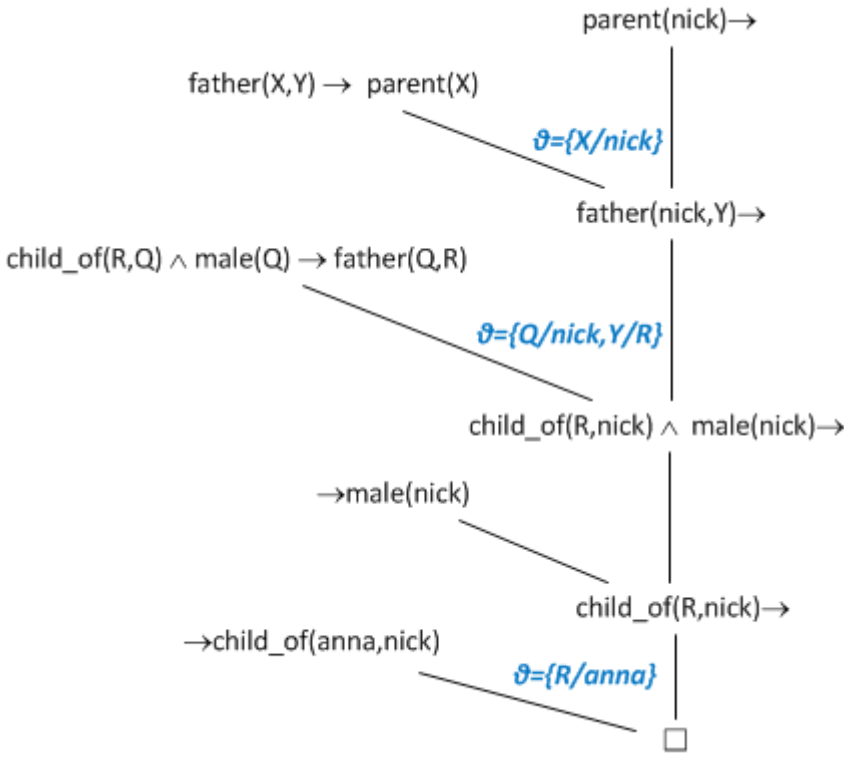
οπότε η πλήρης απόδειξη παίρνει τη μορφή που απεικονίζεται στο Σχήμα 3.2.



Σχήμα 3.2: *Απόδειξη με προτάσεις στη μορφή Kowalski*

Μια ενδιαφέρουσα παρατήρηση στις προηγούμενες αποδείξεις είναι ότι δεν είναι μοναδικές. Για παράδειγμα, η απόδειξη που φαίνεται στο Σχήμα 3.3, είναι το ίδιο έγκυρη με αυτές που παρουσιάστηκαν παραπάνω, αν και η σειρά των βημάτων που ακολουθήθηκε καθώς και οι ενδιάμεσες παραγόμενες προτάσεις είναι διαφορετικές. Η διαφορά ανάμεσα στις δύο αποδείξεις έγκειται στο γεγονός ότι τα συμπληρωματικά ζεύγη που επιλέχθηκαν από τη βάση γνώσης είναι διαφορετικά.

Τέλος, θα πρέπει να τονιστεί ότι η αρχή της ανάλυσης αποτέλεσε τη βάση για την υλοποίηση συστημάτων που βασίζονται στη λογική και τη δημιουργία μιας νέας σχολής γλωσσών προγραμματισμού, το λογικό προγραμματισμό. Ο πλέον διαδεδομένος αντιπρόσωπος αυτής της σχολής είναι η γλώσσα Prolog η οποία αποτελεί αντικείμενο μελέτης του παρόντος βιβλίου. Η Prolog χρησιμοποιεί για την αναπαράσταση της γνώσης προτάσεις Horn και για την εξαγωγή νέας γνώσης μια παραλλαγή της γραμμικής ανάλυσης, που ονομάζεται SLD-ανάλυση (Selection Linear Definite clause resolution). Η συγκεκριμένη αποδεικτική διαδικασία θα παρουσιαστεί σε επόμενα κεφάλαια.



Σχήμα 3.3: *Εναλλακτική απόδειξη σε μορφή Kowalski*

Παράδειγμα εφαρμογής

Στηριζόμενος στο ολοκληρωμένο παράδειγμα των προτάσεων (1), (2) και (3) της ενότητας 3.3, καθώς και στις αναπαραστάσεις των προτάσεων στην κατηγορηματική λογική και στις διάφορες κανονικές μορφές της, να αποδειχθεί ότι: "*Κανένα από τα μαθήματα που διδάσκει ο Πέτρος δεν είναι βαρετό*".

Καταρχήν θα πρέπει να αναπαρασταθεί και η αποδεικτέα πρόταση στην κατηγορηματική λογική. Οι έννοιες που συνδυάζει είναι *μαθήματα* (*course*), *διδάσκει* (*teaches*) και βαρετό (*boring*). Όπως είδαμε στην ενότητα 3.3, τα μαθήματα που διδάσκει ο Πέτρος αναπαρίστανται με την πρόταση *course(X)* ∧ *teaches(petros, X)*. Για όλα αυτά τα μαθήματα θα πρέπει να ισχύει ότι δεν είναι βαρετά, άρα θα χρησιμοποιηθεί καθολικός ποσοδείκτης και συνεπαγωγή:

$$\forall X\,((course(X) \wedge teaches(petros, X)) \rightarrow \neg boring(X)) \qquad (4\alpha)$$

Καθώς και η αποδεικτέα πρόταση, αλλά και οι αρχικές προτάσεις (1α), (2α) (3α) είναι σε μορφή που δύσκολα μπορεί να εφαρμοστεί αποδεικτική διαδικασία βασισμένη σε modus ponens σε συνδυασμό με τις υπόλοιπες αποδεικτικές διαδικασίες, θα χρησιμοποιήσουμε την αρχή της ανάλυσης σε συνδυασμό με την εις άτοπο απαγωγή. Ως πρώτο βήμα είναι να μετατρέψουμε την άρνηση της (4α) στην προτασιακή μορφή της λογικής. Αρχικά γίνεται η μετατροπή σε προσημασμένη κανονική συζευκτική μορφή:

$$\neg\forall X\,((course(X) \wedge teaches(petros, X)) \rightarrow \neg boring(X)) \Leftrightarrow \qquad \text{// βήμα 1}$$

$$\neg\forall X\,(\neg(course(X) \wedge teaches(petros, X)) \vee \neg boring(X)) \Leftrightarrow \qquad \text{// βήμα 2}$$

$$\neg\forall Q\,(\neg(course(Q) \wedge teaches(petros, Q)) \vee \neg boring(Q)) \Leftrightarrow \qquad \text{// βήμα 3}$$

$$\exists Q\,(course(Q) \wedge teaches(petros, Q) \wedge boring(Q)) \qquad (4\beta)$$

Η παραπάνω πρόταση, λόγω του υπαρξιακού ποσοδείκτη ο οποίος δεν είναι στην εμβέλεια κάποιου καθολικού ποσοδείκτη, μετασχηματίζεται στην ακόλουθη πρόταση στην μορφή Skolem, θεωρώντας ότι η μεταβλητή Q αντικαθίσταται από την σταθερά Skolem sk_Q:

$course(sk_Q) \wedge teaches(petros, sk_Q) \wedge boring(sk_Q)$ (4γ)

Στην προτασιακή μορφή της λογικής η παραπάνω πρόταση "διασπάται" στις ακόλουθες τρεις:

$course(sk_Q)$ (4γ.1)

$teaches(petros, sk_Q)$ (4γ.2)

$boring(sk_Q)$ (4γ.3)

Αντίστοιχα, στην μορφή Kowalski:

$\rightarrow course(sk_Q)$ (4δ.1)

$\rightarrow teaches(petros, sk_Q)$ (4δ.2)

$\rightarrow boring(sk_Q)$ (4δ.3)

Η απόδειξη μπορεί να γίνει είτε με την αρχή της ανάλυσης στην προτασιακή μορφή της λογικής, είτε στην μορφή Kowalski. Ακολουθεί αρχικά η απόδειξη στην προτασιακή μορφή της λογικής, ενώ στο Σχήμα 3.4 φαίνεται η απόδειξη στην μορφή Kowalski. Στην απόδειξη συμμετέχουν οι ακόλουθες προτάσεις:

$student(sk_X)$ (1γ.1)

$\neg course(Y) \vee attends(sk_X, Y)$ (1γ.2)

$\neg student(X) \vee \neg course(W) \vee \neg boring(W) \vee \neg attends(X, W)$ (2γ)

$\neg course(R) \vee \neg teaches(petros, R) \vee \neg student(Z) \vee attends(Z, R)$ (3γ)

$course(sk_Q)$ (4γ.1)

$teaches(petros, sk_Q)$ (4γ.2)

$boring(sk_Q)$ (4γ.3)

Η απόδειξη έχει ως εξής:

(4γ.3), (2γ) $\vdash \{W/sk_Q\}\ \neg student(X) \vee \neg course(sk_Q) \vee \neg attends(X, sk_Q)$ (5γ)

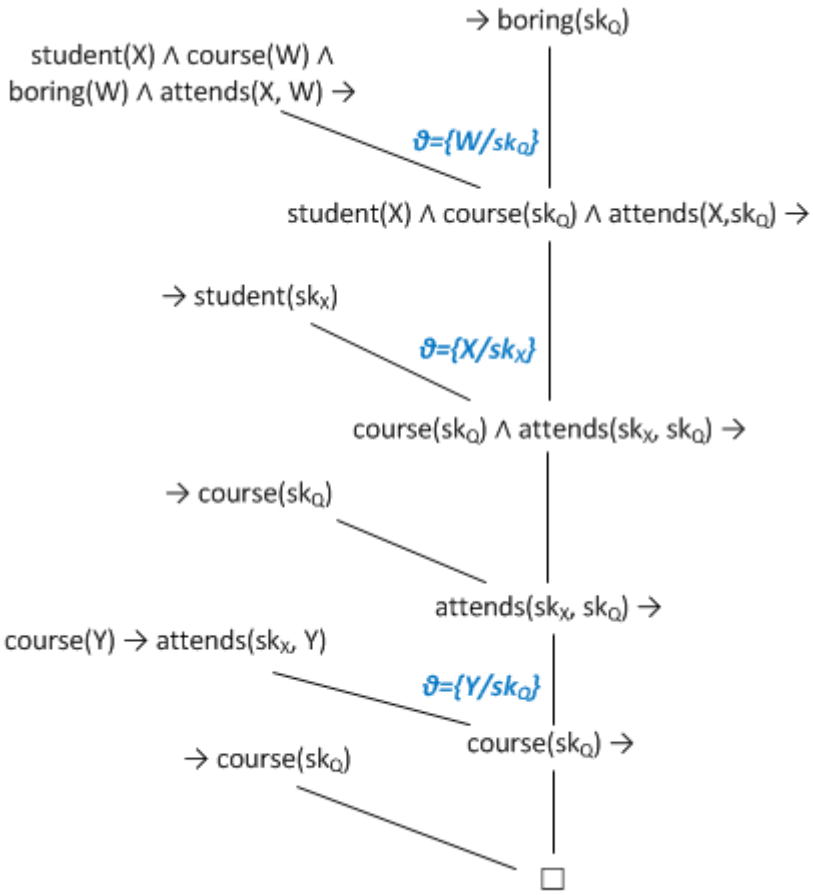
(5γ), (1γ.1) $\vdash \{X/sk_X\}\ \neg course(sk_Q) \vee \neg attends(sk_X, sk_Q)$ (6γ)

(6γ), (4γ.1) $\vdash \neg attends(sk_X, sk_Q)$ (7γ)

(7γ), (1γ.2) $\vdash \{Y/sk_Q\}\ \neg course(sk_Q)$ (8γ)

(8γ), (4γ.1) $\vdash \square$

Στην τελευταία γραμμή της απόδειξης καταλήγουμε σε άτοπο, άρα ισχύει η άρνηση της πρότασης (4γ), δηλαδή η πρόταση (4α).



Σχήμα 3.4: *Απόδειξη παραδείγματος σε μορφή Kowalski*

Βιβλιογραφία

Υπάρχει ένα πλήθος βιβλίων που αφορούν τη μαθηματική λογική, τόσο στην ελληνική, όπως τα (Τζουβάρας, 1992), (Μητακίδης, 1992) και (Τουρλάκης, 2011), όσο και στην ξένη βιβλιογραφία. Αναφέρονται ενδεικτικά τα (Robinson, 1979), (Mendelson, 1979), (Fitting, 1996), και (Hein, 2010), καθένα από τα οποία αποτελεί μια πολύ καλή εισαγωγή στη λογική. Η αρχή της ανάλυσης προτάθηκε από τον Robinson στην κλασική εργασία (Robinson, 1965). Επίσης εκτεταμένες εισαγωγές στη λογική υπάρχουν και σε βιβλία λογικού προγραμματισμού και της γλώσσας Prolog, όπως για παράδειγμα στα (Covington et al., 1988), (Clocksin & Mellish, 2003), συμπεριλαμβανομένου του πλέον σημαντικού (Kowalski, 1979), το οποίο παρουσιάζει εκτενώς την θεωρία πάνω στην οποία βασίστηκε όλη η λεγόμενη "υπολογιστική λογική" και ο "λογικός προγραμματισμός".

Επίσης, εισαγωγή στην λογική υπάρχει και σε βιβλία βάσεων δεδομένων, όπως τα (Ullman, 1988) και (Gray, 1984), καθώς το σχεσιακό μοντέλο δεδομένων είναι ισοδύναμο με την κατηγορηματική λογική πρώτης τάξης.

Τέλος, υπάρχουν εκτεταμένες παρουσιάσεις της λογικής και σε βιβλία που επικεντρώνονται κυρίως στη χρήση της στην Αναπαράσταση της Γνώσης και στην Συλλογιστική για την επίλυση προβλημάτων στην Τεχνητή Νοημοσύνη, όπως τα αγγλόφωνα (Russell & Norvig, 2009), (Luger, 2008), (Nilsson, 1998) και το ελληνικό (Βλαχάβας κ.ά, 2011).

Για το μάθημα της Prolog, γενικότερα, αλλά και για την Λογική και την σχέση της με τον Λογικό Προγραμματισμό, ειδικότερα, υπάρχουν διαφάνειες που αναπτύχθηκαν στο πλαίσιο του έργου Open Courses του Αριστοτελείου Πανεπιστημίου Θεσσαλονίκης (Βασιλειάδης, 2015).

Clocksin W. F. and Mellish Christopher S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

Covington, M. A. and Nute, D. and Vellino A. (1988). Prolog Programming in Depth. Scott Foresman & Co. ISBN 9780673186591.

Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving*. 2nd edition. Springer.

Gray, P. (1984). *Logic, Algebra and Databases*. John Wiley & Sons.

Hein, J. L. (2010). *Discrete Structures, Logic, and Computability*. 3rd edition. Jones and Bartlett.

Kowalski, R. (1979). *Logic for Problem Solving*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Luger G.F. (2008). *Artificial Intelligence: Structures and Strategies for Complex Problem-Solving*. 6th edition. Addison Wesley Longman.

Mendelson E. (1979). *Introduction to Mathematical Logic*. D. Van Nostrand Company.

Nilsson N.J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.

Robinson J.A. (1965). A machine-oriented logic based on the resolution principle. *Journal of ACM*. Vol. *12*. pp. 23-41.

Robinson J.A. (1979). *Logic Form and Function*. University Press.

Russell S. and Norvig P. (2009). *Artificial Intelligence: A Modern Approach*. 3rd edition. Prentice Hall.

Ullman J. D. (1988). *Principles of Database and Knowledge-Base Systems*. Volume I. Computer Science Press. ISBN 0-7167-8158-1.

Βασιλειάδης, Ν. (2015). *Υπολογιστική Λογική και Λογικός Προγραμματισμός*. Ανακτήθηκε την Πέμπτη, 02 Απριλίου 2015 από http://eclass.auth.gr/courses/OCRS163/

Βλαχάβας, Ι. και Κεφαλάς, Π. και Βασιλειάδης, Ν. και Κόκκορας, Φ. και Σακελλαρίου Η. (2011). Τεχνητή Νοημοσύνη. Γ' Έκδοση. ISBN: 978-960-8396-64-7. Εκδόσεις Πανεπιστημίου Μακεδονίας.

Μητακίδης Γ. (1992). *Από τη Λογική στο Λογικό Προγραμματισμό και την Prolog*. Εκδόσεις Καρδαμίτσα.

Τζουβάρας Α. (1992). *Στοιχεία Μαθηματικής Λογικής*. Εκδόσεις Ζήτη.

Τουρλάκης Γ. (2001). *Μαθηματική Λογική: Θεωρία και Πράξη*. Πανεπιστημιακές Εκδόσεις Κρήτης.

Ασκήσεις

3.1 Αποδείξτε με τη χρήση πινάκων αλήθειας ότι οι παρακάτω τύποι είναι ταυτολογίες:

(α) $(\neg P \rightarrow P) \rightarrow P$

(β) $(P \rightarrow Q) \wedge (\neg P \rightarrow Q) \rightarrow Q$

(γ) $\neg Q \wedge (P \rightarrow Q) \rightarrow \neg P$

(δ) $P \wedge \neg P \rightarrow Q$

3.2 Αποδείξτε με τη χρήση πινάκων αλήθειας ότι οι παρακάτω τύποι είναι αντιφάσεις:

(α) $P \wedge \neg Q \wedge (P \rightarrow Q)$

(β) $P \leftrightarrow \neg P$

3.3 Αποδείξτε με τη χρήση πινάκων αλήθειας την ορθότητα των κανόνων συμπερασμού του Πίνακα 3.6.

3.4 Αποδείξτε με τη χρήση πινάκων αλήθειας την ορθότητα των ισοδυναμιών του Πίνακα 3.5.

3.5 Με την χρήση των λογικών ισοδυναμιών του Πίνακα 3.5 να αποδειχθούν οι ταυτολογίες, αντιφάσεις και λογικές ισοδυναμίες των ασκήσεων 3.1, 3.2, 3.3.

3.6 Να διατυπωθούν σε κατηγορηματική λογική οι ακόλουθες προτάσεις:

a. Δεν είναι όλοι οι συγγραφείς βιβλίων διάσημοι.
b. Μερικά βιβλία είναι διάσημα.
c. Ένα βιβλίο είναι διάσημο αν και μόνο αν είναι καλογραμμένο.
d. Όλοι οι στιχουργοί είναι ποιητές.
e. Μερικοί στιχουργοί είναι ποιητές.
f. Κανένας ποιητής δεν είναι στιχουργός.
g. Όλοι αγαπούν κάποιον.
h. Υπάρχει κάποιος που τον αγαπούν όλοι.
i. Πίσω από κάθε επιτυχημένο άνδρα κρύβεται μία φιλόδοξη γυναίκα.
j. Κανένας άνθρωπος δεν είναι μεγαλύτερος από τον πατέρα του/της.

3.7 Προσπαθήστε να διατυπώσετε σε κατηγορηματική λογική τις ακόλουθες προτάσεις. Θα διαπιστώσετε ότι οι προτάσεις είναι αμφίσημες. Συνεπώς κωδικοποιήστε την κάθε πρόταση με τουλάχιστον δύο διαφορετικούς τρόπους στην κατηγορηματική λογική.

a. Ο Κώστας απεχθάνεται έναν μαθηματικό.
b. Ο Γιώργος συκοφαντεί τον Κώστα και τους γονείς του.
c. Ένας μαθητής είναι καλύτερος από έναν δάσκαλο.
d. Μόνο οι πλούσιοι Γερμανοί αγοράζουν σπίτια στο Μόναχο.
e. Στον Σωτήρη αρέσει ένα γρήγορο αμάξι.
f. Μερικά λάθη έγιναν από όλους.

3.8 Αποδείξτε με την χρήση των κανόνων συμπερασμού των πινάκων 3.6 και 3.9 τις ακόλουθες συνεπαγωγές στην κατηγορηματική λογική:

(α) $\forall X\,(\,p(X) \rightarrow q(X)\,) \models \neg q(a) \rightarrow \neg p(a)$

(β) $\forall X\,(\,p(X) \rightarrow q(X)\,),\ p(a) \models \exists Y\, q(Y)$

(γ) $\neg\forall X\, q(X) \models \exists X\, q(X)$

(δ) $\exists X\, \neg p(X,a) \models \exists Z\, \neg\forall Y\, p(Y,Z)$

(ε) $\exists X\, \exists Y\, \forall Z\, \forall W\, p(X,Y,Z,W) \models \forall Z\, \exists X\, \forall W\, \exists Y\, p(X,Y,Z,W)$

3.9 Διατυπώστε στην κατηγορηματική λογική τον ακόλουθο ισχυρισμό και αποδείξτε ότι είναι έγκυρος

Όλοι οι φιλόσοφοι που έχουν μελετήσει λογική ξέρουν τον Gödel. Συνεπώς, εάν όλοι οι φιλόσοφοι έχουν μελετήσει λογική, τότε όλοι ξέρουν τον Gödel.

3.10 Μετατρέψτε όλες τις προτάσεις των ασκήσεων 3.6, 3.7, 3.9, σε:

a. Προσημασμένη συζευκτική κανονική μορφή
b. Κανονική μορφή κατά Skolem
c. Προτασιακή μορφή της κατηγορηματικής λογικής
d. Μορφή Kowalski
e. Εξετάστε αν οι προτάσεις που προκύπτουν στην μορφή Kowalski είναι προτάσεις Horn.

3.11 Με την χρήση της αρχής της ανάλυσης εξετάστε αν ο ακόλουθος ισχυρισμός είναι έγκυρος:

$A \rightarrow B, \neg(B \rightarrow C) \rightarrow \neg A \models A \rightarrow C$

3.12 Μετατρέψτε τις ακόλουθες προτάσεις σε συζευκτική κανονική μορφή:

a. $A \vee (B \wedge C)$
b. $A \leftrightarrow (B \wedge C)$
c. $(\neg A \wedge (\neg B \rightarrow C)) \rightarrow D$
d. $\neg(A \rightarrow \neg C) \wedge (\neg B \rightarrow C)$

3.13 Χρησιμοποιώντας οποιονδήποτε από τους κανόνες συμπερασμού του Πίνακα 3.6, πραγματοποιήστε την ακόλουθη απόδειξη:

$\neg Q \vee P, \neg R \rightarrow \neg P, S, S \rightarrow \neg R, T \rightarrow P \vdash \neg T \wedge \neg Q$

Στη συνέχεια, πραγματοποιήστε την ίδια απόδειξη χρησιμοποιώντας αποκλειστικά την αρχή της ανάλυσης, αρχικά χωρίς την χρήση της απαγωγής σε άτοπο και στη συνέχεια με την χρήση της απαγωγής σε άτοπο.

3.14 Αποδείξτε με τη χρήση πινάκων αλήθειας και των ισοδυναμιών του Πίνακα 3.5 τις παρακάτω λογικές ισοδυναμίες:

a. $(A \vee B) \wedge (A \vee \neg B) \Leftrightarrow A$
b. $\neg((A \wedge B) \vee (\neg A \wedge \neg B)) \Leftrightarrow (\neg A \wedge B) \vee (\neg B \wedge A)$

ΚΕΦΑΛΑΙΟ 4: Σύνταξη Prolog Προγραμμάτων

Λέξεις Κλειδιά:
Φράσεις (Γεγονότα – Κανόνες – Ερωτήσεις), Λογικές μεταβλητές, Δομές Δεδομένων

Περίληψη
Το κεφάλαιο παρουσιάζει τη σύνταξη της γλώσσας Prolog. Δίνεται αρχικά η σχέση μεταξύ της σύνταξης της Κατηγορηματικής Λογικής Πρώτης Τάξης και της σύνταξης της γλώσσας, ενώ μέσω απλών παραδειγμάτων, παρουσιάζονται τα δομικά στοιχεία ενός Prolog προγράμματος, δηλαδή τα γεγονότα και οι κανόνες. Δεδομένων αυτών παρουσιάζεται ο τρόπος αλληλεπίδρασης του χρήστη με το πρόγραμμα, δηλαδή τα ερωτήματα Prolog. Εισάγεται η έννοια της Λογικής μεταβλητής και τονίζεται η ιδιαιτερότητα της ανάθεσης τιμής σε αυτή καθώς και η διαφορά μεταξύ των γλωσσών μοναδικής και καταστροφικής ανάθεσης. Τέλος παρουσιάζονται οι συναρτησιακοί ή σύνθετοι όροι, και τονίζεται ότι αυτοί αποτελούν τη βασική αλλά εξαιρετικά ευέλικτη δομή δεδομένων στην Prolog.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός να:

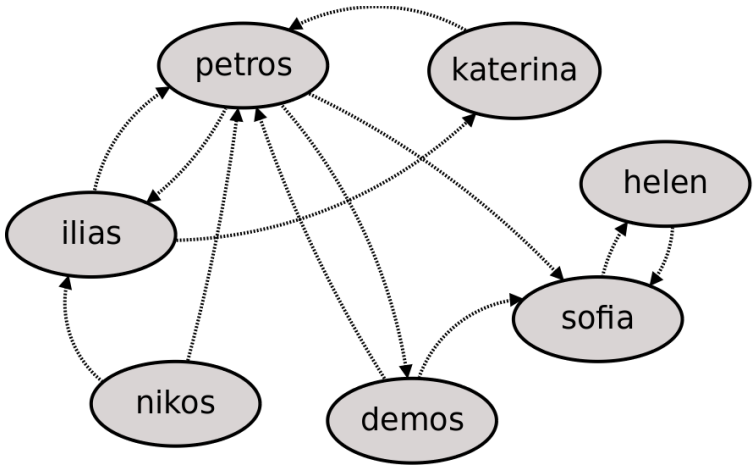
- γνωρίζει την βασική σύνταξη της γλώσσας Prolog, καθώς και την σχέση της με την κατηγορηματική λογική πρώτης τάξης,
- κατανοήσει ένα Prolog πρόγραμμα,
- περιγράψει ένα πρόβλημα χρησιμοποιώντας γεγονότα και κανόνες,
- εκτελεί ένα λογικό πρόγραμμα, μέσω κατάλληλων Prolog ερωτήσεων,
- κατανοεί τη φύση των λογικών μεταβλητών και την έννοια της μοναδικής ανάθεσης,
- κατανοεί την έννοια των συναρτησιακών όρων και της εκφραστικής τους δύναμης στην αναπαράσταση τύπων και δομών δεδομένων.

Μονοπάτι Μάθησης

Το κεφάλαιο αποτελεί θεμελιακό κεφάλαιο για την εισαγωγή στον λογικό προγραμματισμό, καθώς παρουσιάζει τη σύνταξη της γλώσσας Prolog, και αποτελεί προαπαιτούμενο για τα επόμενα κεφάλαια. Οι ενότητες 4.1 - 4.6 παρουσιάζουν την σύνταξη και την εκτέλεση Prolog προγραμμάτων, ενσωματωμένα κατηγορήματα αναγνώρισης τύπων των όρων, την επιλογή κατάλληλης αναπαράστασης και δύο παραδείγματα ανάπτυξης προγραμμάτων. Αυτές οι ενότητες μαζί με την ενότητα 4.10 περιέχουν ύλη η οποία πρέπει απαραίτητα να περιληφθεί στο βασικό μονοπάτι μάθησης. Οι ενότητες 4.7, 4.8 και 4.9 εξετάζουν κάποια περισσότερο προχωρημένα θέματα και μπορούν αν παραλειφθούν σε μια πρώτη ανάγνωση.

Παράδειγμα κίνητρο

Ένα κοινωνικό δίκτυο στον παγκόσμιο ιστό βασίζεται σε σχέσεις φιλίας (friendship) που δημιουργούνται μεταξύ χρηστών. Οι χρήστες έχουν αποθηκευμένα στοιχεία του προφίλ τους, όπως για παράδειγμα το φύλλο (male/female), η ηλικία, το επάγγελμα, κλπ. Οι χρήστες συνδέονται με σχέσεις φιλίας, οι οποίες είναι μονόδρομες, δηλαδή, ένας χρήστης μπορεί να "ακολουθεί" (follows) κάποιον άλλο χρήστη του δικτύου (Σχήμα 4.1).



Σχήμα 4.1: *Ένα κοινωνικό δίκτυο με χρήστες που συνδέονται με σχέσεις φιλίας.*

Στο κοινωνικό δίκτυο το ζητούμενο είναι να εξάγουμε χρήσιμη πληροφορία χρησιμοποιώντας γεγονότα που αναπαριστούν το δίκτυο και κανόνες που ορίζουν πιο πολύπλοκες σχέσεις, βασιζόμενοι σε απλούστερες σχέσεις. Μια τέτοια πολύπλοκη σχέση είναι για παράδειγμα είναι ότι δύο χρήστες είναι φίλοι (friends) όταν ο ένας ακολουθεί τον άλλο ή ότι ο χρήστης έχει φίλους του ιδίου φύλλου (friend_same_gender).

Η περιγραφή του παραπάνω προβλήματος χρησιμοποιώντας κατηγορηματική λογική πρώτης τάξης, απαιτεί καταρχήν την αποτύπωση της βασικής σχέσης follows μεταξύ δύο χρηστών. Η σχέση αυτή, καθώς συνδέει δύο οντότητες του πεδίου (domain), θα έχει δύο ορίσματα, με το πρώτο να αναπαριστά το χρήστη που "ακολουθεί" το χρήστη που αναπαριστάται στο δεύτερο όρισμα:

follows(ilias,petros) ←

follows(petros,ilias) ←

follows(nikos,petros) ←

...

Δύο χρήστες X και Y είναι "φίλοι" όταν ακολουθεί ο ένας τον άλλο, δηλαδή ισχύει ταυτόχρονα (σύζευξη) ο X να ακολουθεί τον Y και ο Y τον X. Αυτή η σχέση εκφρασμένη στη λογική διατυπώνεται ως:

friends(X,Y) ← *follows(X,Y)* ∧ *follows(Y,X)*

Στην παραπάνω συνεπαγωγή, απουσιάζουν οι ποσοδείκτες, όπως εξηγήθηκε στο Κεφάλαιο 3, δηλαδή έχουμε προτάσεις Horn στη μορφή Kowalski. (Στην Prolog, οι συνεπαγωγές γράφονται με τις "συνθήκες" της συνεπαγωγής στα δεξιά και το "συμπέρασμα" στα αριστερά του συμβόλου συνεπαγωγής, το οποίο έχει αντίστροφη φορά). Το φύλλο ενός χρήστη μπορεί να περιγραφεί με αντίστοιχα κατηγορήματα male/1 και female/1:

male(petros) ←

male(ilias) ←

...

female(sofia) ←

female(helen) ←

...

Η σχέση friend_same_gender είναι λίγο πιο πολύπλοκη, καθώς ορίζονται δύο εναλλακτικές περιπτώσεις μια για κάθε φύλλο:

friend_same_gender(X,Y) ← *male(X)* ∧ *friends(X,Y)* ∧ *male(Y).*

friend_same_gender(X,Y) ← *female(X)* ∧ *friends(X,Y)* ∧ *female(Y).*

Η πρώτη πρόταση, εκφράζει ότι ο Χ είναι φίλος του ιδίου φύλλου με τον Υ, αν ισχύει ότι ο Χ είναι άρρεν ΚΑΙ οι Χ και Υ φίλοι ΚΑΙ ο Υ είναι άρρεν. Η δεύτερη πρόταση εκφράζει την αντίστοιχη σχέση όταν οι χρήστες Χ και Υ είναι γένους θηλυκού.

4.1 Εμπειρικοί κανόνες σύνταξης της Prolog

Η σύνταξη της Prolog ακολουθεί εκείνη της **κατηγορηματικής λογικής προτάσεων Horn** ή φράσεων Horn (Horn Clauses) (στα επόμενα, θα χρησιμοποιούμε τον όρο φράση ή πρόταση χωρίς διάκριση μεταξύ τους). Τα σύμβολα της λογικής αντικαθίστανται από κατάλληλα σύμβολα για να διατυπώνονται εύκολα σαν πρόγραμμα Η/Υ, δηλαδή χρησιμοποιώντας τα σύμβολα των ASCII χαρακτήρων. Η μορφή που προκύπτει είναι πολύ κοντά στη μορφή Kowalski.

Έτσι, οι βασικοί κανόνες σύνταξης της γλώσσας είναι:

- το σύμβολο της σύζευξης (∧) αντικαθίσταται από το κόμμα (“,”),
- το σύμβολο της διάζευξης (∨) αντικαθίσταται από το από ερωτηματικό (“;”),
- το σύμβολο της συνεπαγωγής (←) αντικαθίσταται από τα σύμβολα άνω κάτω τελεία και παύλα (“:-”)
- οι σταθερές και τα ονόματα των συναρτησιακών συμβόλων και των κατηγορημάτων ξεκινούν με πεζό γράμμα.
- οι (λογικές) μεταβλητές ξεκινούν πάντα με ένα κεφαλαίο γράμμα,
- η κάθε λογική πρόταση τελειώνει με τελεία.

Βάσει των παραπάνω κανόνων, στο πρόβλημα του κοινωνικού δικτύου που παρουσιάστηκε παραπάνω, οι σχέση follows:

follows(ilias,petros) ←

εκφράζεται σε Prolog:

```
follows(ilias,petros).
```

Η απαλοιφή του συμβόλου της συνεπαγωγής, αποτελεί απλοποίηση της γραφής της πρότασης, καθώς το δεξιό μέλος της συνεπαγωγής στην λογική έκφραση θεωρείται ως αληθές. Αυτό συνεπάγεται ότι το αριστερό μέλος της συνεπαγωγής είναι αδιαμφισβήτητα αληθές. Τέτοιου είδους προτάσεις ονομάζονται **γεγονότα**. Προφανώς η αποτύπωση όλων των σχέσεων follows του κοινωνικού δικτύου απαιτεί τη συγγραφή όλων των αντίστοιχων γεγονότων:

```
follows(petros,ilias).
follows(nikos, petros).
...
```

Η σχέση friends, η οποία ορίστηκε ως:

friends(X,Y) ← *follows(X,Y)* ∧ *follows(Y,X)*

με τις αντικαταστάσεις των συμβόλων που περιγράφηκαν παραπάνω, γράφεται στην Prolog:

```
friends(X,Y) :- follows(X,Y), follows(Y,X).
```

όπου τα Χ και Υ είναι **μεταβλητές** καθώς ξεκινούν με κεφαλαίο γράμμα. Τέτοιου είδους προτάσεις, όπου το δεξί μέλος είναι σύζευξη ενός ή περισσότερων ατομικών τύπων, ονομάζονται **κανόνες**. Οι κανόνες εκφράζουν την αλήθεια μιας σχέσης υπό συνθήκη, δηλαδή στη συγκεκριμένη περίπτωση η σχέση friends είναι αληθής όταν είναι αληθής η σύζευξη που εμφανίζεται δεξιά του συμβόλου της συνεπαγωγής (σύμβολο **:-**).

4.2 Εκτέλεση ενός Prolog προγράμματος οδηγούμενη από ερωτήματα

Κατά συνέπεια, ένα Prolog πρόγραμμα αποτελείται από προτάσεις Horn (γεγονότα και κανόνες), που περιγράφουν σε ένα υψηλό επίπεδο αφαίρεσης το πρόβλημα (τον κόσμο του προβλήματος σύμφωνα με την TN). Δεδομένης μιας τέτοιας περιγραφής, ο χρήστης μπορεί να ρωτήσει για την αλήθεια μιας λογικής πρότασης, να διαπιστώσει δηλαδή αν μια πρόταση αποτελεί λογικό συμπέρασμα του προγράμματος. Η απόδειξη της πρότασης αποτελεί την "εκτέλεση" ενός Prolog προγράμματος. Για παράδειγμα, έστω το ακόλουθο πρόγραμμα που εκφράζει μερικές από τις σχέσεις που περιγράφουν το κοινωνικό δίκτυο του παραδείγματος:

```
follows(ilias,petros).
follows(petros,ilias).
follows(petros, demos).
follows(petros,sofia).
follows(nikos, petros).

friends(X,Y) :-
  follows(X,Y),
  follows(Y,X).
```

Η πλέον προφανής ερώτηση που μπορεί να γίνει στα παραπάνω είναι να ζητηθεί από το σύστημα να αποδείξει ότι ο χρήστης petros ακολουθεί τον χρήστη ilias:

```
?- follows(petros,ilias).
true
```

Η Prolog, εφόσον έχει διατυπωθεί το αντίστοιχο γεγονός, γνωρίζει την "αλήθεια" της ερώτησης και απαντά true, δηλαδή αληθές. Αντίστοιχα στην ερώτηση:

```
?-follows(ilias,helen).
false
```

η απάντηση είναι false. Στην συγκεκριμένη περίπτωση, ο μηχανισμός της γλώσσας απέτυχε να αποδείξει την αλήθεια της πρότασης, καθώς δεν περιλαμβάνονταν αντίστοιχο γεγονός στο πρόγραμμα. Η Prolog ακολουθεί την "υπόθεση του κλειστού κόσμου" (closed world assumption) που πολύ απλά λέει ότι "ό,τι δεν γνωρίζω είναι ψευδές" και απαντά ότι η προς απόδειξη πρόταση είναι ψευδής (false). Περισσότερα για την άρνηση και την αποτυχία θα εξετάσουμε στο Κεφάλαιο 9.

Όμως οι παραπάνω ερωτήσεις είναι ιδιαίτερα απλές και σε πρώτη ανάγνωση, όχι ιδιαίτερα χρήσιμες. Η εισαγωγή μεταβλητών σε μια ερώτηση κάνει την αλληλεπίδραση με το πρόγραμμα περισσότερο ενδιαφέρουσα:

```
?- follows(petros,X).
X = ilias.
```

Η εισαγωγή της μεταβλητής στην ερώτηση, διατυπώνει ουσιαστικά το ερώτημα "υπάρχει X, για το οποίο η λογική σχέση follows(petros,X) να είναι αληθής;" και η Prolog απαντά δίνοντας (ενοποιώντας) την μεταβλητή X με την σταθερά ilias. Ο μηχανισμός απόδοσης τιμών στις μεταβλητές ώστε η ερώτηση να γίνει συντακτικά όμοια με το γεγονός, ονομάζεται ενοποίηση, παρουσιάστηκε στο Κεφάλαιο 3 και θα εξεταστεί ειδικότερα στα πλαίσια της Prolog, στο Κεφάλαιο 5.

Μια γρήγορη ανάγνωση του προγράμματος, δείχνει ότι υπάρχουν και άλλα γεγονότα στο πρόγραμμα τα οποία θα μπορούσαν να αποδώσουν στο X τιμές ώστε η σχέση να αληθεύει. Αν ζητηθεί από την Prolog να παράξει όλες τις εναλλακτικές λύσεις (πατώντας το πλήκτρο ";") η συμπεριφορά θα είναι η ακόλουθη:

```
?- follows(petros,X).
X = ilias ;
X = demos ;
X = sofia.
```

Ο μηχανισμός μέσω του οποίου η Prolog επιστρέφει τις εναλλακτικές λύσεις λέγεται οπισθοδρόμηση (backtracking) και θα εξεταστεί σε επόμενο κεφάλαιο (Κεφάλαιο 5).

Μια ερώτηση μπορεί να είναι και σύνθετη, δηλαδή να είναι σύζευξη ερωτήσεων. Για παράδειγμα, η ακόλουθη ερώτηση διατυπώνει το ερώτημα αν η σύζευξη των δύο κατηγορημάτων είναι αληθής:

```
?- follows(petros,ilias),follows(ilias,petros).
true.
```

και εφόσον τα δύο γεγονότα υπάρχουν στο πρόγραμμα η Prolog απαντά με true. Μέχρι στιγμής εξετάστηκαν μόνο ερωτήσεις οι οποίες αφορούν γεγονότα, όμως οι ερωτήσεις μπορούν να αφορούν και σχέσεις που ορίζονται μέσω κανόνων. Για παράδειγμα, η ερώτηση:

```
?- friends(petros,ilias).
true.
```

αφορά τη σχέση friends που ορίζεται μέσω του κανόνα που δόθηκε παραπάνω. Η Prolog σε αυτήν την περίπτωση για να αποδείξει την αλήθεια της ερώτησης, εξετάζει τη σύζευξη του δεξιού μέλος της συνεπαγωγής και εφόσον την αποδείξει, απαντά ότι η ερώτηση είναι αληθής. Προφανώς οι ερωτήσεις και σε αυτήν την περίπτωση μπορούν να περιέχουν μεταβλητές:

```
?- friends(petros,X).
X = ilias .


?- friends(X,Y).
X = ilias,
Y = petros ;
X = petros,
Y = ilias ;
```

Το δεύτερο ερώτημα παρουσιάζει μεγαλύτερο ενδιαφέρον, καθώς η Prolog επιστρέφει δύο απαντήσεις φαινομενικά ίδιες μεταξύ τους. Εδώ απαιτείται προσοχή καθώς η σειρά των ορισμάτων έχει σημασία, έστω και αν η προσωπική μας ερμηνεία είναι ότι η σχέση είναι αναγκαστικά αμφίδρομη. Για την Prolog, τα γεγονότα friends(ilias, petros) και friends(petros, ilias) παράγουν δύο διακριτές λύσεις.

Ένα βίντεο το οποίο παρουσιάζει την ανάπτυξη του παραπάνω απλού προγράμματος και την διαδικασία εκτέλεσής του βρίσκεται εδώ.

4.3 Η Σύνταξη της Prolog

Η ενότητα αυτή παρουσιάζει το πλήρες συντακτικό της γλώσσας, παραθέτοντας τους κανόνες της σύνταξης με μεγαλύτερη λεπτομέρεια.

Γεγονότα, Κανόνες και Κατηγορήματα

Όπως παρουσιάστηκε στην προηγούμενη παράγραφο, ένα Prolog πρόγραμμα αποτελείται από προτάσεις Horn της μορφής:

$$H :- B_1, B_2, \ldots, B_n.$$

όπου τα H, B_1,... ,B_n είναι ατομικοί τύποι.

- Αν n = 0, τότε η πρόταση ονομάζεται **γεγονός** (fact) ή μοναδιαία πρόταση (unit clause).
- Αν n > 0, τότε η πρόταση ονομάζεται **κανόνας** (rule). Σε ένα κανόνα το αριστερό μέλος της συνεπαγωγής ονομάζεται **κεφαλή** (head) και το δεξιό μέλος **σώμα** (body) του κανόνα.

Οι ατομικοί τύποι έχουν τη μορφή $p(t_1, t_2, \ldots t_n)$, όπου το p ονομάζεται όνομα (σύμβολο) κατηγορήματος (predicate name), ενώ τα $t_1, \ldots t_n$ ονομάζονται ορίσματα (arguments), όπως ακριβώς και στην κατηγορηματική λογική πρώτης τάξης. Ο αριθμός των ορισμάτων ονομάζεται τάξη (arity) του κατηγορήματος. Ως σύμβολο κατηγορήματος μπορεί να χρησιμοποιείται οποιαδήποτε συμβολοσειρά ξεκινά με πεζό γράμμα και περιέχει γράμματα ([a-z/A-Z]), αριθμούς [0-9] και το σύμβολο της κάτω παύλας "_" (underscore), έχει δηλαδή όπως θα δούμε παρακάτω ίδιους κανόνες με τα άτομα.

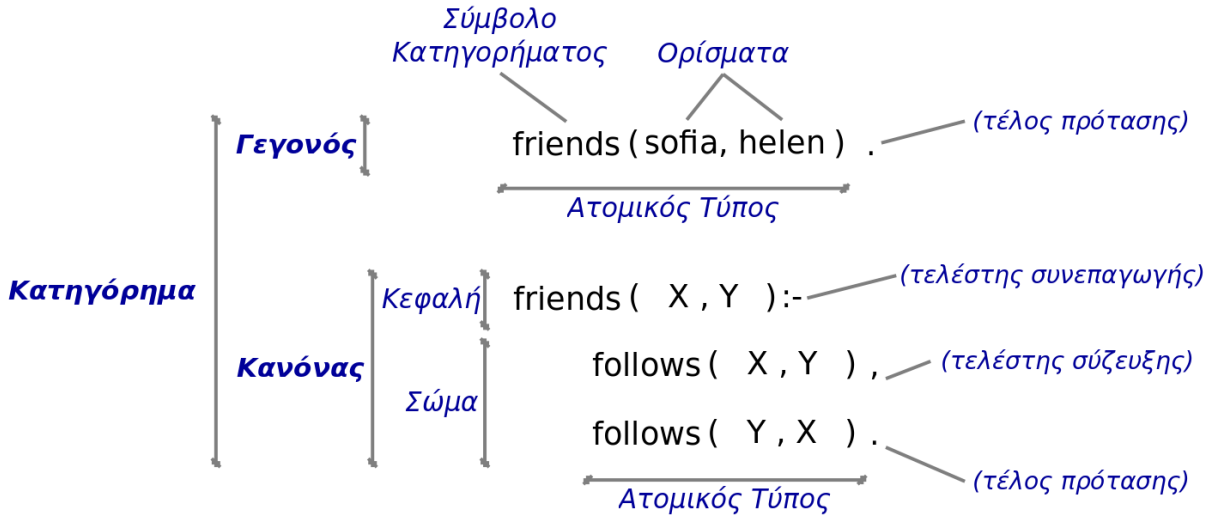
Ένα **κατηγόρημα (predicate)** (Σχήμα 4.2) ορίζεται ως συνδυασμός οποιουδήποτε αριθμού γεγονότων και κανόνων και χαρακτηρίζεται από το όνομά του και την τάξη του. Είναι σύνηθες να αναφερόμαστε σε ένα κατηγόρημα χρησιμοποιώντας το συμβολισμό **όνομα/τάξη (name/arity)**. Για παράδειγμα, η σχέση friends/2 θα μπορούσε να ορίζεται ως:

```
friends(sofia,helen).
friends(helen,sofia).
friends(X,Y) :-
  follows(X,Y),
  follows(Y,X).
```

αν ήταν επιθυμητό να δηλωθεί ρητά ότι οι χρήστες helen και sofia είναι φίλοι. Η σχέση friends_same_gender/2 της οποίας ο ορισμός σε κατηγορηματική λογική δίνεται παραπάνω, γράφεται στην Prolog σαν ένα κατηγόρημα με δύο κανόνες:

```
friend_same_gender(X,Y):-
  female(X),
  friends(X,Y),
  female(Y).

friend_same_gender(X,Y):-
  male(X),
  friends(X,Y),
  male(Y).
```

Τα ορίσματα των κατηγορημάτων είναι όροι. Ένας όρος μπορεί να είναι **σταθερά**, **μεταβλητή** ή **συναρτησιακός όρος**, όπως και στην κατηγορηματική λογική.



Σχήμα 4.2: *Η δομή των κατηγορημάτων στην Prolog.*

Σταθερές της Prolog

Ως σταθερές ορίζονται τα άτομα και οι αριθμοί. Οι αριθμοί έχουν την συνήθη μορφή (1, 2, 3.4 κλπ), όπως και στις άλλες γλώσσες προγραμματισμού. Τα άτομα, αντιπροσωπεύουν οντότητες του προβλήματος, και μπορεί να είναι:

- συμβολοσειρές γραμμάτων και ψηφίων που ξεκινούν από πεζό γράμμα και μπορεί να περιλαμβάνουν την κάτω παύλα (underscore), για παράδειγμα sunday_morning, theSun, more_Days12
- οποιαδήποτε συμβολοσειρά σε μονά εισαγωγικά, για παράδειγμα ‘Sunday Morning’, ‘SUN’, ‘s o s’
- οι συγκεκριμένες ακολουθίες χαρακτήρων [], {}, ; , !

- οποιαδήποτε συμβολοσειρά που περιλαμβάνει *αποκλειστικά* τα: +, -, *, /, \, ^, <, >, =, : , . ,?,@,#,$,&, για παράδειγμα ==>, \#=, >>>, --->, &=>

Μεταβλητές

Ως (λογικές) **μεταβλητές** ορίζονται οποιεσδήποτε συμβολοσειρές ξεκινούν με κεφαλαίο γράμμα και αποτελούνται από γράμματα και ψηφία. Για παράδειγμα τα ακόλουθα είναι μεταβλητές:

```
X, Father, FileName, File_name, File_Name
```

Οι μεταβλητές μπορούν να πάρουν **ως τιμή οποιονδήποτε όρο**, δηλαδή μια σταθερά (άτομο ή αριθμό), μια άλλη μεταβλητή ή ένα συναρτησιακό όρο. Υπάρχουν δύο σημαντικές διαφορές των λογικών μεταβλητών με τις μεταβλητές των κλασσικών διαδικαστικών γλωσσών.

Η πρώτη διαφορά αφορά στον **μηχανισμό ανάθεσης τιμής** σε μεταβλητή. Στην Prolog ο μοναδικός τρόπος για μία μεταβλητή να πάρει τιμή είναι μέσω του μηχανισμού ενοποίησης δύο όρων (Κεφάλαια 3 και 5), που αποτελεί μέρος της εκτέλεσης του λογικού προγράμματος. Περισσότερα για τον μηχανισμό ενοποίησης θα παρουσιαστούν στο Κεφάλαιο 5. Στην Prolog υπάρχει ένας ειδικός ενθεματικός τελεστής (infix operator) που συμβολίζεται με το ίσον (=) ο οποίος καλεί ρητά τον μηχανισμό ενοποίησης στα δύο ορίσματά του:

```
?- X = ilias.
X = ilias.
?- X = Y.
X = Y.
?- X = 12.
X = 12.
?- 12 = X.
X = 12.
```

Η τελευταίες δύο ερωτήσεις καταδεικνύουν μια ακόμη διαφορά του τελεστή ενοποίησης από την ανάθεση των άλλων γλωσσών. Δεν έχει σημασία τι εμφανίζεται αριστερά και δεξιά του τελεστή, απλά τα δύο ορίσματα γίνονται συντακτικά όμοια με κατάλληλες αναθέσεις τιμών. Ο αλγόριθμος ενοποίησης θα εξεταστεί αναλυτικά στο Κεφάλαιο 5.

Η δεύτερη και ίσως σημαντικότερη διαφορά είναι ότι η Prolog ανήκει στην κατηγορία των γλωσσών **μοναδικής ανάθεσης** (single assignment), δηλαδή, από την στιγμή (σημείο εκτέλεσης) στο οποίο μια μεταβλητή παίρνει τιμή, η τιμή αυτή δεν αλλάζει. Αντίθετα, στις γλώσσες καταστροφικής ανάθεσης (C/C++, Java), ο τελεστής ανάθεσης αντικαθιστά την προηγούμενη τιμή της μεταβλητής. Για παράδειγμα, στο ακόλουθο πρόγραμμα σε C, η τιμή της x είναι στο τέλος της εκτέλεσης 14:

```
int x;
x = 12;
x = 14;
```

Αντίθετα, η ακόλουθη Prolog ερώτηση αποτυγχάνει:

```
?- X = 12, X = 14.
false.
```

Για να γίνει κατανοητό το αποτέλεσμα, θα πρέπει η παραπάνω ερώτηση να αναγνωσθεί με δηλωτικό τρόπο: ποιο είναι το X το οποίο ενοποιείται με το 12 και ταυτόχρονα ενοποιείται με το 14. Η προφανής λογική απάντηση είναι ότι τέτοιο X δεν υπάρχει, άρα η Prolog απαντά αρνητικά. Η **εμβέλεια** των μεταβλητών (scope of variables) περιορίζεται στο γεγονός ή στον κανόνα ή στην ερώτηση όπου εμφανίζονται και όχι στο σύνολο των φράσεων που ορίζουν το κατηγόρημα. Άρα υπάρχει πλήρης ελευθερία να χρησιμοποιηθούν διαφορετικά ονόματα μεταβλητών σε κανόνες του ίδιου κατηγορήματος. Για παράδειγμα το κατηγόρημα friend_same_gender/2 που δόθηκε παραπάνω:

```
friend_same_gender(X,Y):-
  female(X),
  friends(X,Y),
  female(Y).
```

```
friend_same_gender(X,Y):-
  male(X),
  friends(X,Y),
  male(Y).
```

μπορεί εντελώς ισοδύναμα να γραφεί ως:

```
friend_same_gender(User,FemaleFriend):-
  female(User),
  friends(User,FemaleFriend),
  female(FemaleFriend).

friend_same_gender(User,MaleFriend):-
  male(User),
  friends(User,MaleFriend),
  male(MaleFriend).
```

Μια κάτω παύλα (_) συνιστά μια ειδική περίπτωση μεταβλητής, που ονομάζεται **ανώνυμη μεταβλητή**. Η ανώνυμη μεταβλητή δηλώνει ότι ο χρήστης δεν ενδιαφέρεται να "μάθει" την τιμή που θα αποδοθεί στο συγκεκριμένο όρισμα. Για παράδειγμα θέλουμε να διατυπώσουμε το ερώτημα "ακολουθεί κάποιον ο ilias", η Prolog ερώτηση είναι:

```
?- follows(ilias,_).
true .
```

ενώ το ερώτημα "ποιον ακολουθεί ο ilias" διατυπώνεται ως:

```
?- follows(ilias,X).
X = petros .
```

Η κύρια χρήση των ανώνυμων μεταβλητών είναι σε συνδυασμό με τον μηχανισμό εντοπισμού μεμονωμένων μεταβλητών (signleton variables), ο οποίος υπάρχει για να προφυλάσσει τον προγραμματιστή από λάθη στα ονόματα των μεταβλητών. Αν μια μεταβλητή σε έναν κανόνα εμφανίζεται μόνο μια φορά, τότε η τιμής της (προφανώς) δεν απαιτείται σε άλλα κατηγορήματα του κανόνα, άρα είναι αδιάφορη και θα πρέπει να δηλωθεί σαν ανώνυμη μεταβλητή. Όλες οι σύγχρονες υλοποιήσεις της Prolog, υποστηρίζουν ένα μηχανισμό ο οποίος εμφανίζει ως προειδοποίηση (warning) την ύπαρξη τέτοιων μεταβλητών σε μια πρόταση (γεγονός ή κανόνα).

Συναρτησιακοί ή Σύνθετοι όροι

Όπως εξετάστηκε στο Κεφάλαιο 3 της Κατηγορηματικής Λογικής, οι συναρτησιακοί (functional) ή **σύνθετοι όροι** (compound terms) έχουν τη μορφή $f(t_1, t_2, \ldots t_n)$, όπου το f ονομάζεται συναρτησιακό σύμβολο (functor) και τα $t_1..t_n$ ορίσματα. Το συναρτησιακό σύμβολο ακολουθεί ίδιους λεκτικούς κανόνες με τα άτομα, δηλαδή στη συνηθισμένη περίπτωση είναι οποιαδήποτε συμβολοσειρά γραμμάτων και ψηφίων που ξεκινά με πεζό γράμμα. Κάθε όρισμα με τη σειρά του μπορεί να είναι οποιοσδήποτε όρος (ακόμη και σύνθετος). Για παράδειγμα τα ακόλουθα είναι σύνθετοι όροι:

```
age(44)
age(ilias,44)
description(ilias,age(ilias,44),job(instructor))
```

Σε πρώτη ανάγνωση, γεννιέται το ερώτημα γιατί τα παραπάνω είναι σύνθετοι όροι, δηλαδή δεδομένα και όχι κατηγορήματα. Αν και έχουν την ίδια ακριβώς μορφή, υπάρχει μια σημαντικότατη διαφορά:

- ένα κατηγόρημα αποτιμάται από την Prolog σε αληθές ή ψευδές, είναι δηλαδή μια λογική πρόταση.
- ένας σύνθετος όρος δεν αποτιμάται, καθώς αποτελεί μια σύνθετη αναπαράσταση ενός αντικειμένου του κόσμου (μία δομή δεδομένων).

Αυτό που διακρίνει τα δύο είναι η θέση τους στο πρόγραμμα: τα κατηγορήματα τελειώνουν πάντα με τελεία, ενώ οι όροι εμφανίζονται ως ορίσματα. Για παράδειγμα θεωρείστε το ακόλουθο πρόγραμμα:

```
age(ilias,44).
description(ilias,age(ilias,44),job(instructor)).
```

Στην πρώτη πρόταση το age(ilias,44) είναι κατηγόρημα (τελειώνει με τελεία), άρα είναι δυνατό θα τεθεί η ερώτηση:

```
?- age(ilias,44).
true
```

και η Prolog να απαντήσει ανάλογα. Στη δεύτερη πρόταση, το age(ilias,44) είναι όρισμα και αναπαριστά δεδομένα. Στην περίπτωση αυτή, η ερώτηση η οποία μπορεί να γίνει, είναι:

```
?- description(ilias,age(ilias,44),job(instructor)).
true
```

και η Prolog θα αποφανθεί για την αλήθεια του κατηγορήματος description/3 και όχι του age/2. Το παραπάνω οδηγεί σε μια ακόμη σημαντική διαφορά της Prolog σε σχέση με τις άλλες γλώσσες προγραμματισμού, η οποία είναι ότι **δεν** γίνεται αποτίμηση των ορισμάτων ενός κατηγορήματος που "καλείται", ακόμα και αν είναι υπό την μορφή "συνάρτησης", όπως είναι οι σύνθετοι όροι. Έστω για παράδειγμα, η ερώτηση:

```
?- description(ilias,true,job(instructor)).
false.
```

Η απάντηση της Prolog είναι false, καθώς το δεύτερο όρισμα του γεγονότος description/3, age(ilias,44), δεν συνδέεται λογικά με το κατηγόρημα (γεγονός) age(ilias,44) που αποτιμάται σε true.

Οι σύνθετοι όροι αποτελούν τη βασική δομή δεδομένων που υποστηρίζεται από την Prolog και επιτρέπουν την οργάνωση σύνθετης πληροφορίας. Για παράδειγμα, έστω ότι για κάθε χρήστη του κοινωνικού δικτύου, θα θέλαμε να αναπαραστήσουμε πληροφορίες που αφορούν την εθνικότητα, την ημερομηνία γέννησης και το επάγγελμα του χρήστη. Μια πιθανή αναπαράσταση των παραπάνω θα ήταν να χρησιμοποιηθεί ένας όρος:

info(<Nationality>,birthday(<Day>,<Month>,<Year>),job(<Job>)

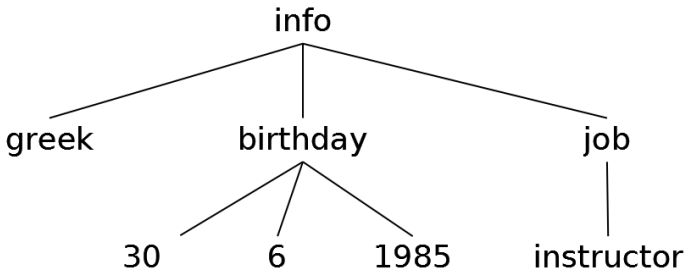
Αν ένας χρήστης έχει ελληνική υπηκοότητα, ημερομηνία γέννησης 30-6-1985 και εργάζεται σαν εκπαιδευτικός, ο όρος που θα περιέγραφε τα παραπάνω θα ήταν:

info(greek,birthday(30,6,1985),job(instructor))

και το γεγονός ότι η παραπάνω πληροφορία αφορά στον χρήστη iliass, θα υλοποιούνταν στην Prolog ως:

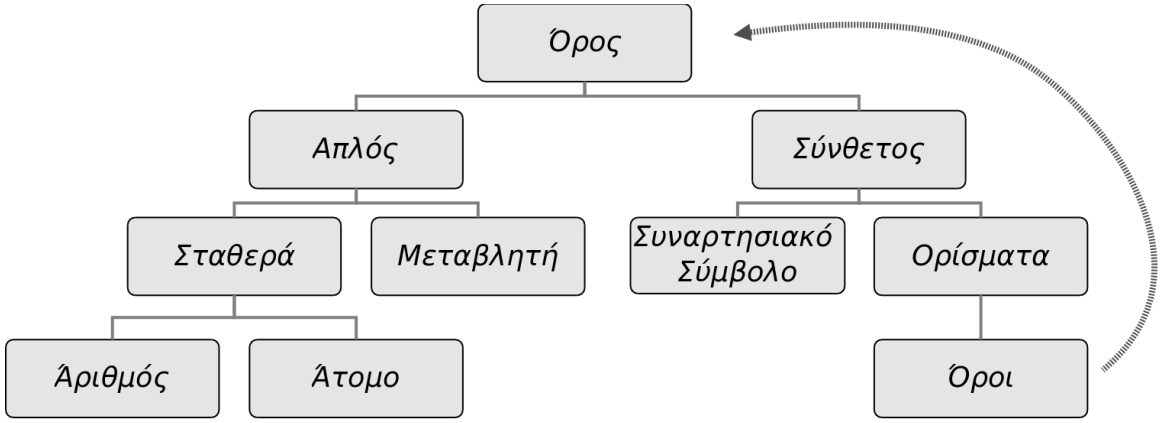
```
user(ilias,info(greek,birthday(30,6,1985),job(instructor))).
```

Οι σύνθετοι όροι αποτελούν ουσιαστικά πεπερασμένα δένδρα (finite trees), με ρίζα του δένδρου το συναρτησιακό σύμβολο και κλαδιά τα ορίσματά του. Για παράδειγμα ο όρος info/2 που περιγράφηκε παραπάνω αντιστοιχεί στο δένδρο που παρουσιάζεται στο Σχήμα 4.3.



Σχήμα 4.3: *Ο όρος info(greek,birthday(30,6,1985),job(instructor)) σαν δένδρο*

Το Σχήμα 4.4 παρουσιάζει διαγραμματικά τις περιπτώσεις όρων της Prolog που παρουσιάστηκαν στην προηγούμενη παράγραφο. Ιδιαίτερη σημασία έχει το γεγονός ότι τα ορίσματα ενός σύνθετου όρου είναι και αυτά όροι, κάτι που επιτρέπει την δημιουργία αναδρομικών δομών δεδομένων με απεριόριστο βάθος, όπως για παράδειγμα τις λίστες, που παρουσιάζονται στο Κεφάλαιο 7.



Σχήμα 4.4: *Οι όροι της Prolog.*

4.4 Η κρισιμότητα της σωστής αναπαράστασης

Η επιλογή της κατάλληλης αναπαράστασης των γεγονότων στο πρόβλημα παίζει σημαντικότατο ρόλο, καθώς καθορίζει σε μεγάλο βαθμό τι είδους ερωτήσεις μπορούν να τεθούν και την ευκολία με την οποία θα αποδειχθούν/απαντηθούν. Για παράδειγμα, έστω ότι οι χρήστες ανήκουν σε μία ή περισσότερες ομάδες (a,b,c) με το χρήστη ilias να ανήκει στις ομάδες a και b, και το χρήστη petros στις ομάδες a και c. Μια αναπαράσταση της παραπάνω πληροφορίας θα μπορούσε να αφορά γεγονότα με ένα όρισμα, το όνομα του χρήστη, και σαν όνομα κατηγορήματος το όνομα της ομάδας:

```
a(ilias).
a(petros).
b(ilias).
c(petros).
...
```

Η παραπάνω αναπαράσταση εξασφαλίζει την ευκολία απάντησης σε ερωτήματα διαπίστωσης αν κάποιος χρήστης είναι μέλος μιας συγκεκριμένης ομάδας, με ερωτήσεις της μορφής:

```
?- a(ilias).
true
?- c(petros).
true
```

ή ακόμη και ποιους χρήστες περιλαμβάνει μια συγκεκριμένη ομάδα:

```
?- a(User).
User = ilias;
User = petros;
```

Εντούτοις, είναι πολύπλοκο να ρωτήσουμε σε ποιες ομάδες ανήκει ένας συγκεκριμένος χρήστης. Η ερώτηση

```
?- X(ilias).
```

δεν είναι έγκυρη ερώτηση, καθώς η Prolog, όντας υποσύνολο της κατηγορηματικής λογικής πρώτης τάξης, δεν επιτρέπει προτάσεις και ερωτήσεις λογικής ανώτερης τάξης, δηλαδή ερωτήσεις που αφορούν στο "ποιο κατηγόρημα αληθεύει" αλλά μόνο αν κάποιο συγκεκριμένο κατηγόρημα είναι αληθές. Με άλλα λόγια, η χρήση των μεταβλητών στην Prolog επιτρέπεται μόνο στην θέση των ορισμάτων και όχι των ονομάτων των κατηγορημάτων. Αν θα έπρεπε να εξαχθεί αυτή η πληροφορία, η μόνο λύση θα ήταν να τεθούν διαδοχικά τα ερωτήματα

```
?- a(ilias).
true
?- b(ilias).
true
?- c(ilias).
false
```

Η κατάσταση δυσχεραίνεται περισσότερο γράφοντας το κατηγόρημα same_group(X,Y) το οποίο πετυχαίνει όταν οι χρήστης X και Y ανήκουν στην ίδια ομάδα, καθώς απαιτεί ένα κανόνα για κάθε μια από τις ομάδες:

```
same_group(X,Y):- a(X), a(Y).
same_group(X,Y):- b(X), b(Y).
same_group(X,Y):- c(X), c(Y).
```

Για τον παραπάνω σκοπό, μια καταλληλότερη αναπαράσταση αφορά στην θεώρηση ότι οι ομάδες αποτελούν και αυτές οντότητες του προβλήματος και ορισμό της σχέσης belongs_to/2, με το πρώτο όρισμα την ομάδα και δεύτερο όρισμα το χρήστη που ανήκει σε αυτή. Σύμφωνα με την νέα αναπαράσταση, η πληροφορία σχετικά με τις ομάδες αναπαρίσταται ως:

```
belongs_to(a,ilias).
belongs_to(a,petros).
belongs_to(b,ilias).
belongs_to(c,petros).
...
```

Το ερώτημα "σε ποιες ομάδες ανήκει ο χρήστης ilias" διατυπώνεται ως:

```
?- belongs_to(Group,ilias).
Group = a ;
Group = b.
```

και η σχέση same_group/2, περιγράφεται δηλωτικά ως "δύο χρήστες X και Y είναι στην ίδια ομάδα, ΑΝ ο χρήστης X ανήκει σε μια ομάδα Group ΚΑΙ ο χρήστης Y ανήκει στην ίδια ομάδα". Σε Prolog αυτό μπορεί να γραφεί ως:

```
same_group(X,Y):-
  belongs_to(Group,X),
  belongs_to(Group,Y).
```

όπου η χρήση της ίδιας μεταβλητής Group στους δύο όρους της σύζευξης εξασφαλίζει ότι ο χρήστης X και ο χρήστης Y ανήκουν στην ίδια ομάδα. Υπενθυμίζεται ότι εφόσον η Prolog είναι γλώσσα μοναδικής ανάθεσης, το κατηγόρημα θα επιτύχει μόνο αν τα δύο κατηγορήματα της σύζευξης αποδίδουν στη μεταβλητή Group την ίδια τιμή. Ο μηχανισμός με τον οποίο η γλώσσα θα εντοπίσει την κατάλληλη τιμή παρουσιάζεται στο Κεφάλαιο 5.

4.5 Παράδειγμα Εφαρμογής: Κοινωνικό δίκτυο

Το πρόβλημα του κοινωνικού δικτύου, επεκτείνεται στην παρούσα ενότητα. Μέχρι στιγμής, οι σχέσεις που περιγράφηκαν είναι:

- follows(X,Y), δηλαδή γεγονότα που περιγράφουν τη δομή του δικτύου (τον γράφο)
- male(X) και female(X), γεγονότα που περιγράφουν το φύλλο του χρήστη X,
- friends(X,Y), που εκφράζει τη σχέση φιλίας μεταξύ των X και Y, η οποία ισχύει όταν οι χρήστες X και Y ακολουθούν ο ένας τον άλλο,
- friends_same_gender(X,Y), που περιγράφει τη σχέση φιλίας ανάμεσα στους χρήστες X και Y του ιδίου φίλου,
- user(X,Inf) γεγονότα που συνδέουν χρήστη X με ένα σύνθετο όρο Inf ο οποίος περιέχει στοιχεία για τον χρήστη,
- user_group(G,X), όπου G η ομάδα στην οποία ανήκει ο χρήστης X, και
- same_group(X,Y), που πετυχαίνει όταν οι χρήστες X και Y ανήκουν στην ίδια ομάδα.

Ας υποθέσουμε ότι θέλουμε να εξάγουμε πληροφορία σχετικά με τους άρρενες φίλους ενός χρήστη. Το κατηγόρημα male_friends, θα έχει δύο ορίσματα, τον χρήστη X και τον άρρενα φίλο του (της) Y. Η δηλωτική περιγραφή του κατηγορήματος είναι "ο χρήστης X έχει άρρενα φίλο τον Y ΑΝ ο X και Y είναι φίλοι ΚΑΙ ο Y είναι άρρενας". Το αντίστοιχο κατηγόρημα σε Prolog, θα υλοποιούνταν ως:

```
male_friends(X,Y):-
  friends(X,Y),
  male(Y).
```

Τέλος, έστω ότι θα θέλαμε να προτείνουμε νέους φίλους σε έναν χρήστη. Ένας χρήστης Υ προτείνεται ως φίλος σε ένα χρήστη Χ, αν ο Υ είναι φίλος κάποιου φίλου Ζ του Χ. Μια πρώτη υλοποίηση του κατηγορήματος είναι η ακόλουθη:

```
recommend_common_friends(X,Y):-
  friends(X,Z),
  friends(Z,Y).
```

Θέτοντας την ακόλουθη ερώτηση στη Prolog:

```
?- recommend_common_friends(ilias,X).
X = ilias .
```

Η απάντηση δεν είναι η αναμενόμενη, καθώς δεν θα θέλαμε να προτείνεται ως νέος φίλος ο ίδιος χρήστης. Όμως το κατηγόρημα friends(X,Y), όπως παρουσιάστηκε σε προηγούμενη παράγραφο, επιτυγχάνει τόσο για την ερώτηση friends(ilias,petros) όσο και για την ερώτηση friends(petros,ilias) καθώς η σχέση είναι συμμετρική. Το προηγούμενο έχει ως αποτέλεσμα την παραπάνω απάντηση της Prolog, καθώς οι μεταβλητές Χ και Υ μπορούν να πάρουν οποιαδήποτε τιμή, άρα και την ίδια.

Για να λειτουργήσει σωστά το κατηγόρημα, θα πρέπει να δηλώσουμε ρητά, ότι οι μεταβλητές Χ και Υ πρέπει να έχουν διαφορετικές τιμές. Αυτό επιτυγχάνεται με τον τελεστή (ενσωματωμένο κατηγόρημα) \= που επιβάλει το αριστερό και δεξιό του όρισμα να μην είναι ενοποιήσιμα. Για παράδειγμα:

```
?- X = 12, Y = 12, X \= Y.
false.

?- X = 12, Y = 14, X \= Y.
X = 12,
Y = 14.
```

Η πρώτη ερώτηση αποτυγχάνει καθώς οι μεταβλητές Χ και Υ έχουν την ίδια τιμή, ενώ η δεύτερη ερώτηση πετυχαίνει. Με τη χρήση του τελεστή \= το κατηγόρημα γίνεται:

```
recommend_common_friends(X,Y):-
  friends(X,Z),
  friends(Z,Y),
  X \= Y.
```

εξασφαλίζοντας ότι δεν θα προτείνει τον ίδιο το χρήστη ως νέο φίλο.

Στα προηγούμενα,ορίστηκε το κατηγόρημα user/2 ως ένα σύνολο γεγονότων τα οποία περιγράφουν πληροφορίες σχετικές με τους χρήστες του κοινωνικού δικτύου. Έστω ότι στο πρόγραμμα περιλαμβάνονται τα ακόλουθα γεγονότα:

```
user(ilias,info(greek,birthday(30,6,1985),job(instructor))).
user(petros,info(greek,birthday(12,1,1980),job(director))).
user(demos,info(greek,birthday(14,1,1980),job(professor))).
user(nick,info(greek,birthday(1,12,1982),job(researcher))).
```

και αυτό που θα θέλαμε είναι να γραφεί ένα κατηγόρημα same_age/2, το οποίο επιτυγχάνει όταν τα ορίσματα του είναι χρήστες οι οποίοι έχουν την ίδια ηλικία (γεννήθηκαν την ίδια χρονιά). Εξετάζοντας τον σύνθετο όρο birthday/3, αυτό εξασφαλίζεται αν το τρίτο όρισμά του είναι το ίδιο για τους δύο χρήστες, το οποίο στην Prolog δηλώνεται απλά με την χρήση της ίδιας μεταβλητής. Έτσι, το κατηγόρημα same_birth_year/2, το οποίο επιτυγχάνει όταν οι δύο όροι birthday/3 των ορισμάτων του αφορούν την ίδια χρονιά, υλοποιείται σε Prolog, ως ένα γεγονός:

```
same_birth_year(birthday(_,_,Year),birthday(_,_,Year)).
```

Η χρήση των ανώνυμων μεταβλητών στα υπόλοιπα ορίσματα των όρων birthday/3, απλά δηλώνει ότι δεν ενδιαφερόμαστε για τις τιμές των αντίστοιχων ορισμάτων (ημερομηνία, μήνας). Δεδομένου του παραπάνω κατηγορήματος, η δηλωτική περιγραφή του same_age/2 είναι “Οι User1 και User2, έχουν την ίδια ηλικία ΑΝ είναι χρήστες του δικτύου και γεννήθηκαν την ίδια χρονιά”. Η υλοποίηση σε Prolog είναι:

```
same_age(User1,User2):-
  user(User1,info(_,BDay1,_)),
  user(User2,info(_,BDay2,_)),
  same_birth_year(BDay1,BDay2),
  User1 \= User2.
```

Ενδιαφέρον παρουσιάζει η χρήση των μεταβλητών BDay1 και BDay2, οι οποίες παίρνουν ως τιμή τους αντίστοιχους όρους birthday/3 των χρηστών. Αυτό είναι εφικτό καθώς μια μεταβλητή μπορεί να πάρει ως τιμή, οποιονδήποτε όρο.

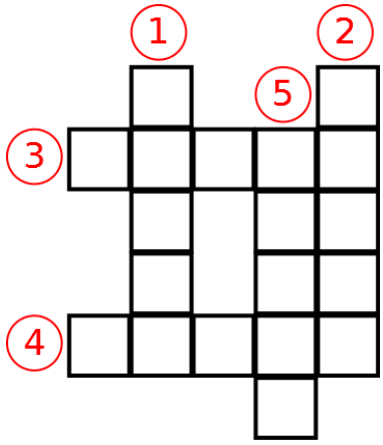
Και στην περίπτωση αυτή, εξασφαλίζουμε ότι η Prolog δεν θα επιστρέψει τον ίδιο χρήστη ως τιμή των μεταβλητών User1 και User2, χρησιμοποιώντας τον τελεστή \=. Η συμπεριφορά του κατηγορήματος, φαίνεται από την ακόλουθη ερώτηση:

```
?- same_age(U1,U2).
U1 = petros,
U2 = demos ;
U1 = demos,
U2 = petros ;
false.
```

Υπενθυμίζεται ότι οι λύσεις petros-demos και demos-petros είναι διαφορετικές για την Prolog.

4.6 Παράδειγμα εφαρμογής: Σταυρόλεξο

Στο σταυρόλεξο που φαίνεται στο Σχήμα 4.5, πρέπει να τοποθετηθούν πέντε συγκεκριμένες λέξεις των πέντε γραμμάτων η καθεμία. Οι διαθέσιμες λέξεις είναι jumps, jumbo, jocks, isles, babel και το ζητούμενο (όπως και σε κάθε σταυρόλεξο) είναι να βρεθεί σε ποιες από τις θέσεις που είναι σημειωμένες στο σχήμα αντιστοιχεί κάθε μια από τις προηγούμενες λέξεις.



Σχήμα 4.5: *Το σταυρόλεξο.*

Ο μόνος περιορισμός που προκύπτει στο πρόβλημα, είναι εκείνος που αφορά στα κοινά γράμματα των λέξεων. Για παράδειγμα, οι λέξεις στη θέση 1 και στη θέση 3 θα πρέπει να έχουν το δεύτερο τους γράμμα κοινό. Με παρόμοιο τρόπο, το τελευταίο γράμμα της λέξης στη θέση 1 είναι ίδιο με το δεύτερο γράμμα της λέξης στη θέση 4.

Εφόσον οι περιορισμοί (λογικές σχέσεις) αφορούν στα μεμονωμένα γράμματα των λέξεων, θα πρέπει να αναπαρασταθούν κατάλληλα οι λέξεις σαν ακολουθίες γραμμάτων. Εφόσον όλες οι λέξεις αποτελούνται από 5 γράμματα, για την αναπαράσταση τους μπορούμε να χρησιμοποιήσουμε ένα κατηγόρημα με όνομα word και 6 ορίσματα, με το πρώτο να είναι η λέξη και τα υπόλοιπα πέντε τα γράμματά της. Για παράδειγμα, η λέξη jumbo θα αναπαρασταθεί με το γεγονός word/6 ως:

```
word(jumbo,j,u,m,b,o).
```

Με παρόμοιο τρόπο αναπαρίστανται και οι υπόλοιπες λέξεις:

```
word(jocks,j,o,c,k,s).
word(isles,i,s,l,e,s).
word(jumps,j,u,m,p,s).
word(babel,b,a,b,e,l).
```

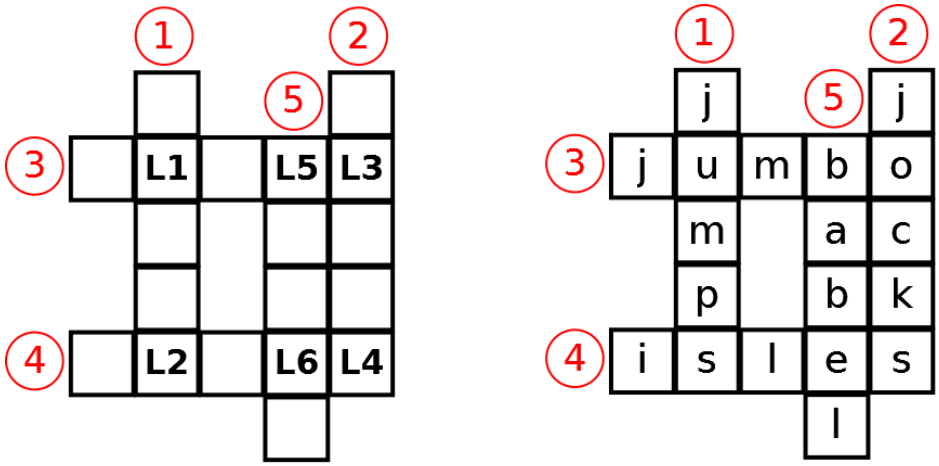
Δεδομένης της παραπάνω αναπαράστασης, είναι απλό να αναζητηθούν δύο λέξεις με κοινά γράμματα σε συγκεκριμένες θέσεις, αν απλά τοποθετηθεί στις αντίστοιχες θέσεις μια κοινή μεταβλητή. Έτσι, για παράδειγμα, αν αναζητούνται δύο λέξεις W1 και W3, με κοινό το δεύτερο τους γράμμα, η αντίστοιχη ερώτηση σε Prolog θα ήταν:

```
?- word(W1,_,L,_,_,_), word(W3,_,L,_,_,_).
W1 = W3, W3 = jumbo,
L = u ;
W1 = jumbo,
L = u,
W3 = jumps ;
....
```

Η παραπάνω ερώτηση επιστρέφει κατά την οπισθοδρόμηση όλες τις πιθανές λύσεις ακόμη και όμοιες λέξεις. Η ύπαρξη επιπλέον περιορισμών, θα μειώσει τις πιθανές λύσεις. Η προσέγγιση επεκτείνεται και στην περίπτωση που αναζητώ τρεις λέξεις W1, W3, και W4 όπου για τις δύο πρώτες ισχύει ο προηγούμενος περιορισμός και για την W4, ότι πρέπει να έχει το δεύτερό της γράμμα ίδιο με εκείνο τελευταίο της W3. Εφόσον πλέον υπάρχουν δύο διαφορετικές θέσεις οι οποίες πρέπει να είναι κοινές, θα απαιτηθεί να υπάρχουν δύο μεταβλητές L1 και L2. Άρα, στη περίπτωση αυτή η σύνθετη ερώτηση γίνεται:

```
?- word(W1,_,L1,_,_,_), word(W3,_,L1,_,_,L2), word(W4,_,L2,_,_,_).
W1 = W3, W3 = jumbo,
L1 = u,
L2 = o,
W4 = jocks ;
W1 = jumbo,
L1 = u,
W3 = jumps,
L2 = s,
W4 = isles ;
```

Κατά συνέπεια, αυτό που απαιτείται είναι να χαρακτηρίσουμε κάθε κοινή θέση δύο λέξεων στο σταυρόλεξο με μια διαφορετική μεταβλητή, όπως φαίνεται στο Σχήμα 4.6.



Σχήμα 4.6: *Οι χαρακτηρισμοί των κοινών θέσεων με μεταβλητές και η τελική λύση του σταυρόλεξου.*

Το κατηγόρημα το οποίο θα "λύνει" το σταυρόλεξο, ονομάζεται crossword και έχει πέντε ορίσματα, ένα για κάθε λέξη, με το πρώτο όρισμα να αντιστοιχεί στη λέξη της θέσης 1 στο σταυρόλεξο, το δεύτερο στη θέση 2 κοκ. Η υλοποίηση του κατηγορήματος είναι η ακόλουθη:

```
crossword(W1,W2,W3,W4,W5):-
```

```
    word(W1,_,L1,_,_,L2),
    word(W2,_,L3,_,_,L4),
    word(W3,_,L1,_,L5,L3),
    word(W4,_,L2,_,L6,L4),
    word(W5,L5,_,_,L6,_).
```

και η "εκτέλεσή" του:

```
?- crossword(W1,W2,W3,W4,W5).
W1 = jumps,
W2 = jocks,
W3 = jumbo,
W4 = isles,
W5 = babel ;
```

Η τιμές των μεταβλητών που επιστρέφονται αντιστοιχούν στη λύση του σταυρόλεξου που φαίνεται στο Σχήμα 4.6.

4.7 Μια πρώτη ματιά στους τελεστές

Συνήθως, η αναπαράσταση αριθμητικών εκφράσεων σε όλες τις γλώσσες προγραμματισμού ακολουθεί τη συνήθη μορφή, η οποία χρησιμοποιεί ενθεματικούς (infix) τελεστές, όπως είναι για παράδειγμα η μορφή της έκφρασης 5 + 3 κλπ. Όπως όμως αναφέρθηκε, η Prolog υποστηρίζει μόνο όρους της μορφής $f(t_1,..t_n)$, άρα οι παραπάνω εκφράσεις πρέπει να γραφούν με την μορφή όρων. Έτσι η αριθμητική έκφραση 5 + 3, στην Prolog αναπαριστάται ως:

```
+(5,3)
```

με την μορφή όρων. Σύμφωνα με τους λεκτικούς κανόνες που δόθηκαν στην αντίστοιχη ενότητα, το σύμβολο + είναι άτομο, άρα το +(5,3) είναι ένας έγκυρος Prolog όρος. Καθώς όμως γίνεται ιδιαίτερα πολύπλοκη η γραφή μεγαλύτερων παραστάσεων με τη μορφή αυτή, η Prolog υποστηρίζει ένα ευέλικτο μηχανισμό δήλωσης τελεστών και αυτόματης μετατροπής των αντίστοιχων εκφράσεων σε μορφή όρων. Ο μηχανισμός θα εξεταστεί στο Κεφάλαιο 13.

Το γεγονός ότι οι εκφράσεις με ενθεματικούς τελεστές, όπως η παραπάνω, μετατρέπονται σε όρους και σε συνδυασμό με το γεγονός ότι η Prolog δεν κάνει αποτίμηση των ορισμάτων ενός κατηγορήματος, οδηγεί συχνά σε προβλήματα τους προγραμματιστές που έρχονται σε πρώτη επαφή με τη γλώσσα. Για παράδειγμα, έστω ένα κατηγόρημα translate/2, που παρέχει τη λεκτική περιγραφή ενός αριθμού:

```
translate(10,ten).
```

Στην ερώτηση

```
?- translate(10,Name),
Name = ten
```

η απάντηση είναι προφανής. Όμως η ερώτηση

```
?- translate(5+5,Name),
false
```

αποτυγχάνει, καθώς το 5 + 5 είναι ο όρος +(5,5) και το 10 μια σταθερά, άρα για την Prolog δεν μπορούν να ενοποιηθούν. Ομοίως η ερώτηση

```
?- 10 = 5 + 5
false
```

αποτυγχάνει για τον ίδιο λόγο.

Η ύπαρξη των τελεστών, έχει ενδιαφέροντες συνέπειες κατά την ενοποίηση. Για παράδειγμα η ερώτηση δίνει τις παρακάτω ενοποιήσεις:

```
?- X + Y = 5 + 3
X = 5
Y = 3
```

επιτρέποντας έτσι την εύκολη επεξεργασία εκφράσεων και την ανάπτυξη εργαλείων όπως μεταγλωττιστών και μετα-διερμηνευτών (meta-interpreters). Τα θέματα αυτά θα εξεταστούν σε επόμενα κεφάλαια.

4.8 Ο τελεστής διάζευξης

Αν και η μορφή Kowalski δεν επιτρέπει την ύπαρξη διαζεύξεων στο σώμα ενός κανόνα, στην Prolog κάτι τέτοιο είναι εφικτό με την χρήση του τελεστή διάζευξης ";". Ο τελεστής, αλλάζει την δηλωτική ερμηνεία των κανόνων, που παρουσιάστηκε μέχρι τώρα. Έστω το κατηγόρημα:

```
a(X) :- b(X) ; c(X).
```

Η ερμηνεία του παραπάνω είναι ότι για το a(X) είναι αληθές, αν το b(X) είναι αληθές Η το c(X) είναι αληθές. Αν και φαίνεται η παραπάνω μορφή να δίνει σημαντική ευελιξία στην ανάπτυξη προγραμμάτων, η χρήση του τελεστή οδηγεί σε πιο δυσανάγνωστα προγράμματα και κώδικα δυσκολότερο στη συντήρηση και την αποσφαλμάτωση. Είναι προτιμότερο οι εναλλακτικές περιπτώσεις ενός κατηγορήματος να υλοποιούνται σαν διαφορετικοί κανόνες. Για παράδειγμα, το κατηγόρημα friends_same_gender/2 που απαρτίζεται από δύο κανόνες:

```
friend_same_gender(X,Y):-
  female(X),
  friends(X,Y),
  female(Y).

friend_same_gender(X,Y):-
  male(X),
  friends(X,Y),
  male(Y).
```

θα μπορούσε με την χρήση της διάζευξης (;) να γραφεί:

```
friend_same_gender(X,Y):-
  female(X),
  friends(X,Y),
  female(Y) ;
  male(X),
  friends(X,Y),
  male(Y).
```

Οι δύο μορφές είναι ισοδύναμες (παράγουν τις ίδιες λύσεις), όμως η πρώτη μορφή που αποτυπώνει κάθε εναλλακτική περίπτωση σαν ένα διαφορετικό κανόνα είναι σημαντικά πιο ευανάγνωστη και θα πρέπει να προτιμάται.

Η προτεραιότητα του τελεστή της διάζευξης είναι χαμηλότερη από εκείνη της σύζευξης, οπότε μια έκφραση της μορφής:

```
a(X),b(X);c(X),d(X)
```

είναι ισοδύναμη με:

```
(a(X),b(X)) ; (c(X),d(X)).
```

4.9 Ενσωματωμένα κατηγορήματα της Prolog που αφορούν όρους

Η Prolog έχει ένα αριθμό ενσωματωμένων κατηγορημάτων που στοχεύουν στην διαπίστωση του "τύπου" ενός όρου, δηλαδή επιτυγχάνουν αν το όρισμα τους είναι σταθερά, μεταβλητή ή σύνθετος όρος. Ενσωματωμένα κατηγορήματα που αφορούν τις σταθερές είναι:

- **atomic/1,** που επιτυγχάνει όταν το όρισμά του είναι άτομο.
- **number/1**, που επιτυγχάνει όταν το όρισμά του είναι αριθμός,
- **atom/1**, που επιτυγχάνει όταν το όρισμά του είναι άτομο,

Οι ακόλουθες ερωτήσεις δείχνουν τη συμπεριφορά των κατηγορημάτων:

```
?- number(8).
true.
?- number(3.5).
true.
?- atom(3).
false.
?- atom(iliass).
true.
?- atom(user).
true.
?- atom(==>).
true.
?- atom(==>).
true.
?- atom(+).
true
```

Μια συνέπεια της μοναδικής ανάθεσης τιμών είναι ότι μια μεταβλητή μπορεί να είναι σε δύο καταστάσεις: **ελεύθερη (free/unbound),** δηλαδή δεν της έχει ακόμη δοθεί τιμή, ή **δεσμευμένη (bound),** οπότε έχει πάρει τιμή μέσω του μηχανισμού ενοποίησης. Η Prolog διαθέτει ένα ειδικό κατηγόρημα **var/1** το οποίο πετυχαίνει όταν το όρισμά του είναι μια ελεύθερη μεταβλητή. Για παράδειγμα:

```
?- var(X).
true.
?- X = 12, var(X).
false
```

Η δεύτερη ερώτηση αποτυγχάνει γιατί κατά την στιγμή εκτέλεσης του var(X) ή μεταβλητή ήδη είχε πάρει τιμή (δεσμευμένη), άρα το κατηγόρημα αποτυγχάνει. Αν η σειρά των κλήσεων μεταβληθεί, τότε η ερώτηση θα επιτύχει:

```
?- var(X), X = 12.
true
```

γιατί κατά τη στιγμή της εκτέλεσης του var(X) η μεταβλητή X ήταν μη δεσμευμένη. Ενδιαφέρον παρουσιάζει η ακόλουθη ερώτηση

```
?- var(X), X = 12, number(X).
X = 12.
```

που επιτυγχάνει γιατί μετά την ενοποίηση X = 12, η μεταβλητή είναι ένας αριθμός (ή δεσμευμένη σε ένα αριθμό). Η παραπάνω ερώτηση, δείχνει και μια ακόμη πλευρά της Prolog, εκείνη της διαδικαστικής ερμηνείας των προγραμμάτων (Κεφάλαιο 5). Η πρώτη κλήση στην var(X) θα εκτελεστεί και θα πάρει τιμή πριν την ενοποίηση, και άρα θα επιτύχει, καθώς η εκτέλεση θα γίνει "από αριστερά προς τα δεξιά". Η κατανόηση της απάντησης στην ερώτηση απαιτεί γνώση του μηχανισμού εκτέλεσης που θα παρουσιαστεί στο Κεφάλαιο 5.

Τέλος, το κατηγόρημα compound/1 επιτυγχάνει όταν το όρισμά του είναι σύνθετος όρος:

```
?- compound(tree(12)).
true.
?- compound(tree).
false.
```

Υπάρχουν και άλλα κατηγορήματα στην συγκεκριμένη ομάδα, και οι διάφορες υλοποιήσεις της Prolog προσφέρουν μια πληθώρα τέτοιων κατηγορημάτων. Εδώ θα πρέπει να σημειωθεί ότι υπάρχει ISO προτυποποίηση της γλώσσας, η οποία απαιτεί να υπάρχουν στις υλοποιήσεις που συμμορφώνονται με αυτή συγκεκριμένος αριθμός ενσωματωμένων κατηγορημάτων.

4.10 Σημαντικές Παρατηρήσεις

Η κρισιμότητα της πειθαρχημένης ανάπτυξης

Όπως και σε όλες τις γλώσσες προγραμματισμού, υπάρχει μια σειρά καλών πρακτικών οι οποίες βοηθούν σημαντικά στην αποφυγή σφαλμάτων. Κάποιες αφορούν απλές οδηγίες μορφοποίησης κώδικα, που ακολουθούνται από την κοινότητα των προγραμματιστών:

- Στοίχιση με χρήση στηλοθέτη (tab) στους ατομικούς τύπους (subgoals) του σώματος ενός κανόνα.
- Ένας ατομικός τύπος του σώματος ενός κανόνα σε κάθε γραμμή.
- Καλό είναι να υπάρχει μια κενή γραμμή ανάμεσα στους κανόνες ενός κατηγορήματος.
- Τα κατηγορήματα θα πρέπει να έχουν ονόματα αντιπροσωπευτικά των σχέσεων που ορίζουν.
- Οι μεταβλητές θα πρέπει να έχουν κατάλληλα ονόματα που να αντιπροσωπεύουν το ρόλο τους στο κατηγόρημα.
- Εισαγωγή σχολίων στον κώδικα.

Οι παραπάνω κανόνες συμβάλλουν σημαντικά στην αναγνωσιμότητα των προγραμμάτων, άρα και στην εύκολη αποσφαλμάτωση/συντήρησή τους. Για παράδειγμα έστω το κατηγόρημα:

```
mfr(X,Y):-friends(X,Y),
male(Y).
```

Αν και είναι ορθό τόσο λογικά όσο και συντακτικά, είναι αρκετά δυσανάγνωστο σε σχέση με την ακόλουθη γραφή του που είναι ισοδύναμη:

```
male_friend(User,Friend):-
  friends(User,Friend),
  male(Friend).
```

Καθώς η πληκτρολόγηση τόσο περιγραφικών ονομάτων μεταβλητών μπορεί να οδηγήσει σε σφάλματα στην “ορθογραφία”, οι υλοποιήσεις της Prolog παρέχουν ένα μηχανισμό εντοπισμού τέτοιων λαθών (singleton variables) που περιγράφεται παρακάτω.

Το παραπάνω κατηγόρημα καταδεικνύει μια ακόμη πτυχή της κωδικοποίησης κατηγορημάτων που αφορά τη **σειρά των ορισμάτων**. Ο προγραμματιστής είναι ελεύθερος να επιλέξει όποια σειρά επιθυμεί στα ορίσματα, αρκεί η σειρά να διατηρηθεί σε όλες τις κλήσεις του κατηγορήματος που θα ακολουθήσουν. Για παράδειγμα το προηγούμενο κατηγόρημα θα μπορούσε κάλλιστα να κωδικοποιηθεί ως:

```
male_friend(Friend, User):-
  friends(User,Friend),
  male(Friend).
```

με το πρώτο όρισμα τον άρρενα φίλο ενός χρήστη που εμφανίζεται στο δεύτερο όρισμα, αρκεί η σύμβαση αυτή να διατηρούνταν και στο υπόλοιπο πρόγραμμα.

Τα σχόλια στην Prolog εισάγονται είτε με τον χαρακτήρα "%" και εκτείνονται μέχρι το τέλος της τρέχουσας γραμμής, είτε περικλείονται στους χαρακτήρες "/*" και "*/". Συνήθως στα σχόλια δίνεται και η περιγραφή του κατηγορήματος με την μορφή κατηγόρημα/τάξη και μια δηλωτική περιγραφή του κατηγορήματος. Για παράδειγμα:

```
%%% male_friends/2
%%% male_friends(User,Friend)
%%% Succeeds if Friend is a male friend of User.
male_friend(User,Friend):-
  friends(User,Friend),
  male(Friend).
```

Τέλος, κανόνες οι οποίοι απαρτίζουν τον ορισμό ενός κατηγορήματος, θα πρέπει να εμφανίζονται μαζί στο αρχείο του προγράμματος. Πολλές υλοποιήσεις της Prolog εμφανίζουν προειδοποιητικό μήνυμα ή ακόμη και σφάλμα αν αυτός ο κανόνας δεν τηρείται.

Ανώνυμες και μεμονωμένες μεταβλητές

Όπως αναφέρθηκε στην αντίστοιχη παράγραφο, οι περισσότερες υλοποιήσεις της Prolog έχουν ένα μηχανισμό προειδοποίησης μεμονωμένων μεταβλητών (signleton variables). Ο μηχανισμός προφυλάσσει τον προγραμματιστή από κοινά λάθη κατά την πληκτρολόγηση ονομάτων μεταβλητών. Έστω το ακόλουθο κατηγόρημα, που πετυχαίνει όταν δύο χρήστες ακολουθούν κάποιο τρίτο χρήστη που με τη σειρά του ακολουθεί κάποιον άλλο:

```
both_follow_a_follower(Person, Follower, Fellow):-
  follows(Folower,Person),
  follows(Fellow,Person),
  Follower \= Fellow,
  follows(Person,Someone).
```

Η Prolog κατά την φόρτωση του κατηγορήματος θα εμφανίσει τα ακόλουθα προειδοποιητικά μηνύματα (warnings):

```
...
Singleton variables: [Folower,Someone]
...
```

Η πρώτη περίπτωση μεμονωμένης μεταβλητής αποτελεί ξεκάθαρα ορθογραφικό λάθος του προγραμματιστή, ο οποίος είχε (προφανώς) την πρόθεση να αναφερθεί στη μεταβλητή Follower. Το προειδοποιητικό μήνυμα επιτρέπει στον προγραμματιστή να αντιληφθεί εύκολα το σφάλμα και μπορεί να το διορθώσει. Αν δεν υπήρχε ο μηχανισμός μεμονωμένων μεταβλητών, τέτοιου είδους σφάλματα θα κόστιζαν πολλές ώρες αποσφαλμάτωσης.

Η δεύτερη περίπτωση της μεταβλητής Someone δεν ανήκει στην ίδια κατηγορία με την πρώτη, καθώς η τιμή της δεν παίζει κανένα ρόλο στη λογική σχέση. Για να αποφύγουμε τα μηνύματα, η ορθή τακτική είναι να αντικατασταθεί η μεταβλητή Someone με την ανώνυμη μεταβλητή (_). Το διορθωμένο κατηγόρημα που δεν παράγει κανένα μήνυμα σφάλματος, είναι:

```
both_follow_a_follower(Person, Follower, Fellow):-
  follows(Follower,Person),
  follows(Fellow,Person),
  Follower \= Fellow,
  follows(Person,_).
```

Τέλος, ως ανώνυμη μεταβλητή είναι οποιαδήποτε μεταβλητή που ξεκινά με τον χαρακτήρα (“_”), για παράδειγμα η ανώνυμη μεταβλητή του κατηγορήματος θα μπορούσε να γραφεί _Someone. Η χρήση τέτοιων επώνυμων-ανώνυμων μεταβλητών απαιτεί προσοχή, όπως φαίνεται στις ακόλουθες ερωτήσεις:

```
?- _S = 2, _S = 4.
false.
?- _ = 2, _ = 4.
true.
```

Όπως φαίνεται παραπάνω, για την Prolog η μεταβλητή της _S πρέπει να είναι ίδια και στις δύο εμφανίσεις της. Είναι καλή πρακτική τα προγράμματα που αναπτύσσονται στην Prolog να μην έχουν κανένα προειδοποιητικό μήνυμα (όπως και σε όλες τις άλλες γλώσσες).

Βιβλιογραφία

Το θέμα της σύνταξης της γλώσσας, υπάρχει σε όλα τα “κλασικά” βιβλία αναφοράς για τη γλώσσα προγραμματισμού Prolog, όπως τα (Clocksin and Mellish, 2003), (Bratko, 2011), (Sterling and Shapiro, 1994) (O’Keefe, 1990) και (Covington et al., 1988).

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Clocksin, W. F. and Mellish C. S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

Covington, M. A. and Nute, D. and Vellino A. (1988). *Prolog Programming in Depth*. Scott Foresman & Co. ISBN 9780673186591.

O'Keefe, R. A. (1990), *The craft of Prolog*, Cambridge, Mass: MIT Press

Sterling, L. and Shapiro, E. Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass: MIT Press.

Άλυτες Ασκήσεις

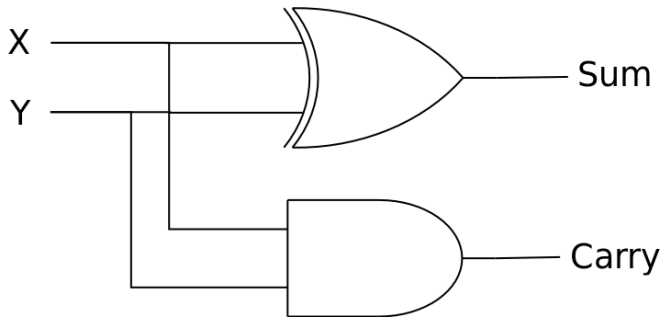
4.1. Να ορίσετε ξανά το κατηγόρημα friends_same_gender/2 του προβλήματος του κοινωνικού δικτύου, ώστε να αποτελείται από μόνο ένα κανόνα, δίνοντας ένα εναλλακτικό τρόπο ορισμού της σχέσεων male/1 και female/1 για τους χρήστες του δικτύου.

4.2. Οι πύλες είναι βασικά ηλεκτρονικά κυκλώματα με ένα αριθμό εισόδων και εξόδων, όπου η έξοδος ορίζεται από τις τιμές εισόδου. Στην Prolog ο πίνακας αληθείας μιας πύλης AND μπορεί να αναπαρασταθεί ως ακολούθως:

x	**y**	**x and y**	**Prolog**
0	0	0	and_gate(0,0,0).
0	1	0	and_gate(0,1,0).
1	0	0	and_gate(1,0,0).
1	1	1	and_gate(1,1,1).

α) Με βάση το παραπάνω παράδειγμα, να ορίσετε σε Prolog τις πύλες OR, XOR, και NOT (κατηγορήματα or_gate/2, xor_gate/2 και not_gate/1).

β) Ένα ψηφιακό κύκλωμα είναι ένα σύνολο από συνδεδεμένες πύλες. Για παράδειγμα το παρακάτω κύκλωμα (Σχήμα 4.7) είναι ένας ημιαθροιστής (half adder) που δημιουργείται από AND και XOR πύλες. Να γράψετε τον Prolog κώδικα που αντιστοιχεί στον παρακάτω ημιαθροιστή



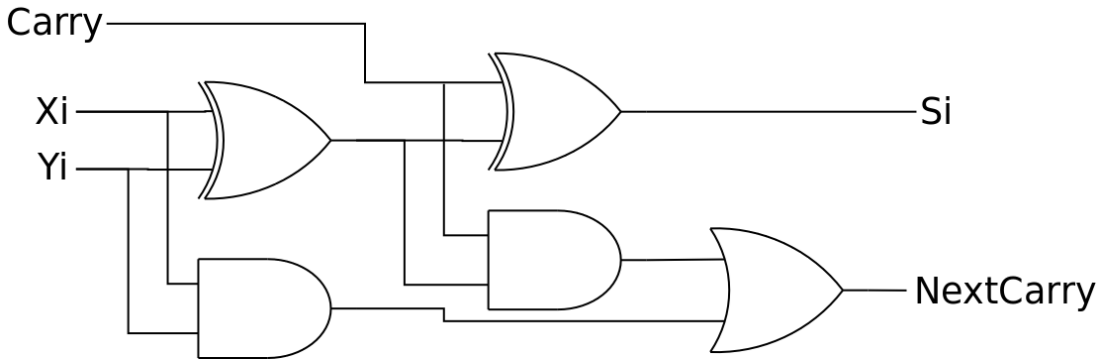
Σχήμα 4.7: *Κύκλωμα ημιαθροιστή*

γ) Δώστε μια ερώτηση της Prolog που να αντιστοιχεί στα παρακάτω ερωτήματα:

1. Ποιος είναι ο πίνακας αληθείας του ημι-αθροιστή?

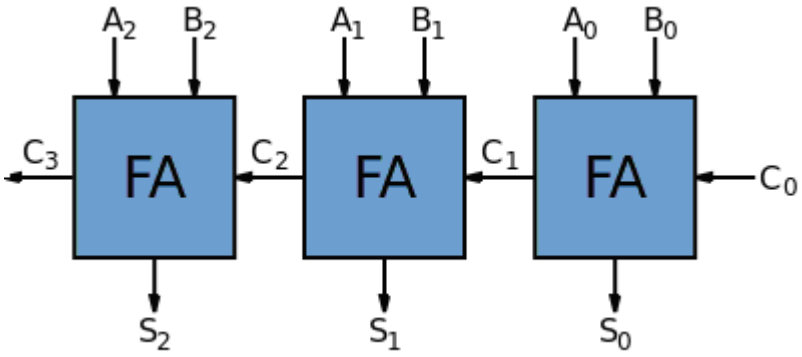
2. Ποιες τιμές δίνουν κρατούμενο 1 (C - Carry).

δ) Γράψτε ένα Prolog πρόγραμμα που να υλοποιεί τον ακόλουθο πλήρη αθροιστή (full-adder):



Σχήμα 4.8: *Κύκλωμα Πλήρη Αθροιστή*

γ) Να γραφεί ο ορισμός σε για έναν αθροιστή δύο τριψήφιων αριθμών (στο Σχήμα 4.9 κάθε FA είναι ένας πλήρης αθροιστής, όπως ορίστηκε παραπάνω):



Σχήμα 4.9: *Κύκλωμα Αθροιστή 3-bit.*

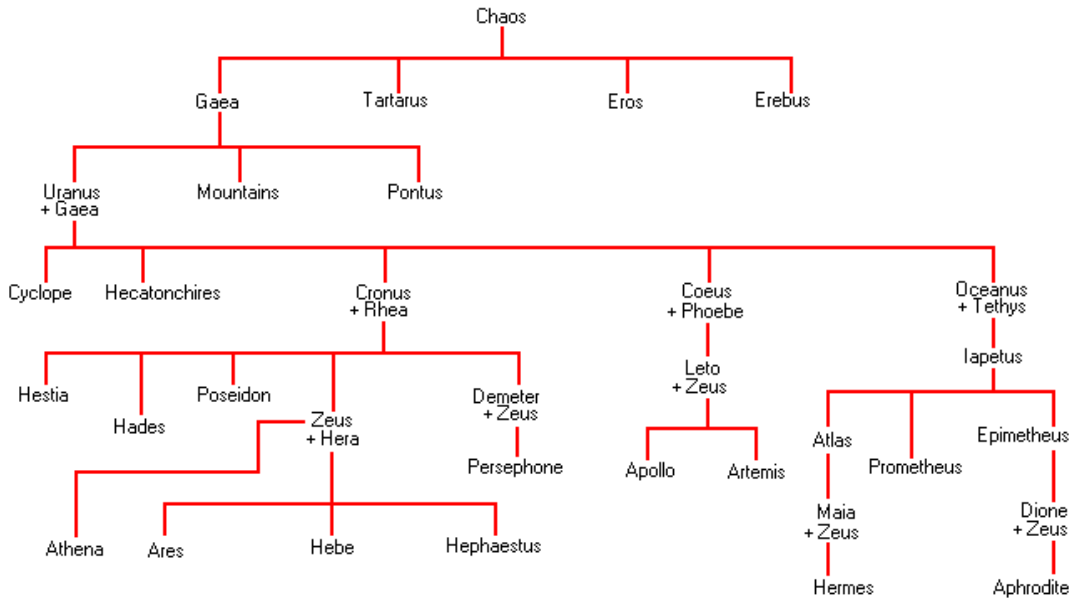
Γράψτε μία ερώτηση που να αθροίζει το A=101 και το B=001, με αρχικό κρατούμενο 0. Άρα τα ψηφία που πρέπει να αντιστοιχιστούν είναι A_0=1, A_1=0, A_2=1, B_0=1, B_1=0, B_2=0 και C_0=0. Θα πρέπει να περιμένετε τον άθροισμα να είναι S=110, δηλαδή S_0=0, S_1=1, S_2=1 και C_3=0.

δ) Να αλλαχθεί ο ορισμός του κατηγορήματος του αθροιστή δύο τριψήφιων αριθμών έτσι ώστε να δέχεται πέντε ορίσματα:

4. τους τρεις αριθμούς A,B και S με τη μορφή όρου number(Digit2,Digit1,Digit0),
5. τα δύο κρατούμενα C0 και C3

4.3 Η παρακάτω παράγραφος εκφράζει ένα ενδιαφέρον λογικό πρόβλημα: "*Παντρεύτηκα μια χήρα (ας την πούμε W) που έχει μια ενήλικη κόρη (ας την πούμε D). Ο πατέρας μου (F) που μας επισκεπτόταν συχνά, ερωτεύθηκε τη θετή μου κόρη και την παντρεύτηκε. Άρα, ο πατέρας μου έγινε πεθερός μου και η θετή μου κόρη έγινε η θετή μου μητέρα. Μερικούς μήνες αργότερα αποκτήσαμε ένα γιο (S1) ο οποίος έγινε ο θετός αδελφός του πατέρα μου, αλλά και ο θείος μου. Η σύζυγος του πατέρα μου, δηλαδή η θετή μου κόρη έκανε επίσης ένα γιο (S2)*". Γράψτε ένα πρόγραμμα Prolog που να λύνει το ερώτημα "*Είμαι ο παππούς μου;*"

4.4 Η Θεογονία του Ησίοδου αναφέρει το γενεαλογικό δένδρο του Σχήματος 4.10 στη μυθολογία.



Σχήμα 4.10: *Γενεαλογικό δένδρο της Θεογονίας*

α) Γράψτε γεγονότα που να αναπαριστούν τις σχέσεις γονιός, άρρεν και θήλυ (parent, male and female).

β) Γράψτε κανόνες που να αναπαριστούν τις σχέσεις: πατέρας (father/2), μητέρα (mother/2), προγονιός (grandparent/2), αδελφοί (brothers/2), αδελφές (sisters/2) και απόγονος (ancestor/2).

ΚΕΦΑΛΑΙΟ 5: Σημασιολογία Prolog Προγραμμάτων

Λέξεις Κλειδιά
Ερμηνεία λογικών προγραμμάτων, ενοποίηση, αποδεικτική διαδικασία Prolog, κλήσεις, οπισθοδρόμηση, παρακολούθηση προγραμμάτων Prolog

Περίληψη
Το κεφάλαιο ξεκινά με την παρουσίαση της διαδικασίας ενοποίησης, δηλαδή του κύριου μηχανισμού ανάθεσης τιμών σε μεταβλητή και περάσματος παραμέτρων, η οποία αποτελεί ένα από τα ισχυρότερα στοιχεία της γλώσσας. Μέσω παραδειγμάτων κλιμακούμενης δυσκολίας δίνεται η έννοια του πλέον γενικού ενοποιητή. Στη συνέχεια και έχοντας ορίσει τη διαδικασία ενοποίησης, περιγράφεται το πώς η αποδεικτική διαδικασία που βασίζεται στον κανόνα της αρχής της ανάλυσης υλοποιείται από τον μηχανισμό εκτέλεσης της Prolog, για την απάντηση ερωτημάτων, μέσω διαφόρων παραδειγμάτων εκτέλεσης που στοχεύουν στην κατανόηση των διαφόρων χαρακτηριστικών του μηχανισμού. Ο μηχανισμός εκτέλεσης της Prolog μπορεί να αναπαρασταθεί εποπτικά χρησιμοποιώντας το λεγόμενο δένδρο εκτέλεσης ή αναζήτησης, το οποίο παρουσιάζεται και αναλύεται εκτενώς, μέσω των αντίστοιχων παραδειγμάτων εκτέλεσης. Τέλος, μέσω των ίδιων παραδειγμάτων παρουσιάζεται ο μηχανισμός αποσφαλμάτωσης ή καλύτερα παρακολούθησης της εκτέλεσης της Prolog, το λεγόμενο "box model", και αναλύεται πώς μπορεί μέσω αυτού να εντοπιστούν σφάλματα σε λογικά προγράμματα.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός:

- Να κατανοεί πώς λειτουργεί ο μηχανισμός ενοποίησης της Prolog.
- Να κατανοεί πώς λειτουργεί ο μηχανισμός εκτέλεσης των λογικών προγραμμάτων της Prolog, πώς ενεργοποιείται και λειτουργεί ο μηχανισμός της οπισθοδρόμησης και πώς επιστρέφονται τα αποτελέσματα σε έναν ερώτημα.
- Να αναπαριστά γραφικά την εκτέλεση ενός λογικού προγράμματος της Prolog.
- Να παρακολουθεί βήμα προς βήμα την εκτέλεση ενός προγράμματος Prolog χρησιμοποιώντας την λειτουργία αποσφαλμάτωσης της Prolog.

Μονοπάτι Μάθησης

Το κεφάλαιο αποτελεί επίσης θεμελιακό κεφάλαιο (όπως και το προηγούμενο) για την κατανόηση του λογικού προγραμματισμού και πώς αυτός πραγματώνεται με την γλώσσα Prolog, καθώς παρουσιάζει τον μηχανισμό εκτέλεσης των λογικών προγραμμάτων και όλους τους συναφείς μηχανισμούς ενοποίησης και οπισθοδρόμησης, και αποτελεί προαπαιτούμενο για τα επόμενα κεφάλαια. Οι ενότητες 5.1 - 5.3 παρουσιάζουν τους προαναφερθέντες μηχανισμούς εκτέλεσης. Σε μια πρώτη ανάγνωση μπορεί κάποιος να παραλείψει τον αλγόριθμο εκτέλεσης και να αρκεστεί στην λεκτική περιγραφή του μηχανισμού εκτέλεσης. Η ενότητα 5.4 παρουσιάζει έναν εποπτικό τρόπο απεικόνισης της εκτέλεσης των Prolog προγραμμάτων με την χρήση των δένδρων αναζήτησης / εκτέλεσης. Αν και με μία επιφανειακή ματιά μπορεί κάποιος να θεωρήσει ότι αυτή η ενότητα μπορεί να παραληφθεί, εντούτοις προτείνεται η ανάγνωσή της καθώς η εποπτική απεικόνιση της εκτέλεσης βοηθάει πάρα πολύ στην κατανόηση της όλης διαδικασίας. Τέλος, η ενότητα 5.5 αφορά στις δυνατότητες αποσφαλμάτωσης της Prolog, κάτι που είναι εντελώς απαραίτητο τόσο στον αρχάριο, όσο και στον προχωρημένο προγραμματιστή. Η ενότητα 5.6 παρουσιάζει μια προχωρημένη περίπτωση στην διαδικασία ενοποίησης, το λεγόμενο occurs check, και σε μία πρώτη ανάγνωση μπορεί να παραληφθεί.

Παράδειγμα κίνητρο

Στο προηγούμενο κεφάλαιο είδαμε κάποια στοιχεία για την εκτέλεση των Prolog προγραμμάτων, όπως ότι η εκτέλεση ξεκινάει με ένα ερώτημα το οποίο ουσιαστικά αποτελεί στόχο απόδειξης (θεώρημα προς απόδειξη), ότι το ερώτημα μπορεί να έχει μεταβλητές, οι τιμές των οποίων αποτελούν μέρος της απάντησης, και ότι το ερώτημα μπορεί να περιέχει παραπάνω από ένα απλούστερα ερωτήματα. Όμως οι απαντήσεις σε αυτά τα ερωτήματα δόθηκαν διαισθητικά μόνο, χωρίς να γίνεται αναφορά στον αλγόριθμο ο οποίος είναι υπεύθυνος για τις απαντήσεις αυτές, στον μηχανισμό με τον οποίο οι μεταβλητές παίρνουν τιμή ή την σειρά με την οποία δίνονται οι απαντήσεις. Όλα τα παραπάνω παρουσιάζονται αναλυτικά με πάρα πολλά χρηστικά παραδείγματα αυξανόμενης πολυπλοκότητας.

5.1 Εισαγωγή - Ερμηνεία Λογικών Προγραμμάτων

Όπως ίσως έχει γίνει ήδη αντιληπτό από τα προηγούμενα κεφάλαια, και κυρίως από το Κεφάλαιο 2, τα (λογικά) προγράμματα στην Prolog μπορούν να γίνουν κατανοητά με δύο τρόπους, τον δηλωτικό (declarative) και τον διαδικαστικό (procedural). Για παράδειγμα, η πρόταση (κανόνας) `P :- Q, R.` μπορεί να ερμηνευτεί:

- είτε δηλωτικά "*Το* **`P`** *είναι αληθές εάν το* **`Q`** *είναι αληθές και το* **`R`** *είναι αληθές*", όπως επιτάσσει η αντιστοιχία της με την πρόταση της μορφής Kowalski $P, Q \rightarrow R$, όπως είδαμε στο Κεφάλαιο 3,
- είτε διαδικαστικά "*Για να αποδειχθεί ότι το* **`P`** *είναι αληθές, πρέπει πρώτα να αποδειχθεί ότι το* **`Q`** *είναι αληθές και στη συνέχεια ότι και το* **`R`** *είναι αληθές*", όπως επιτάσσει ο μηχανισμός εκτέλεσης της Prolog, ο οποίος παρουσιάστηκε πολύ σύντομα στο προηγούμενο κεφάλαιο και θα αναλυθεί στο τρέχον.

Η διαδικαστική ερμηνεία μιας λογικής πρότασης στην Prolog βρίσκεται νοηματικά πολύ κοντά στον ορισμό μιας διαδικασίας στον συμβατικό / διαδικαστικό προγραμματισμό: "*Για να εκτελέσω την διαδικασία* **`P`***, πρέπει να εκτελέσω τις διαδικασίες* **`Q`** *και* **`R`**", συνεπώς αποτελεί τον συνδετικό κρίκο μεταξύ του συμβατικού και του λογικού προγραμματισμού.

Ένα μεγάλο πλεονέκτημα αυτής της ισοδυναμίας είναι η δυνατότητα ελέγχου της ορθότητας (verification) ενός λογικού προγράμματος ευκολότερα από ότι ένα πρόγραμμα σε μία διαδικαστική γλώσσα προγραμματισμού. Οι συνήθεις τεχνικές ελέγχου ορθότητας ενός διαδικαστικού προγράμματος στηρίζονται στην αφηρημένη περιγραφή των προδιαγραφών (specifications) της συμπεριφοράς του προγράμματος με μία υψηλού επιπέδου δηλωτική γλώσσα, η οποία συνήθως είναι κάποια παραλλαγή της κατηγορηματικής λογικής πρώτης τάξης. Στη συνέχεια η λογική αυτή περιγραφή ελέγχεται ως προς διάφορες ιδιότητες με κάποιο λογισμικό απόδειξης θεωρημάτων (theorem prover). Στην Prolog όμως αυτή η μετατροπή δεν είναι απαραίτητη καθώς το ίδιο το πρόγραμμα είναι γραμμένο στην γλώσσα της λογικής, συνεπώς η εκτέλεσή του με τον μηχανισμό εκτέλεσης της Prolog συνιστά απόδειξη θεωρήματος! Συνεπώς στα λογικά προγράμματα η αφηρημένη περιγραφή και το πρόγραμμα ταυτίζονται, διευκολύνοντας έτσι την ανάπτυξη και τον έλεγχο των προγραμμάτων.

5.2 Διαδικασία Ενοποίησης

Πριν προχωρήσουμε στην παρουσίαση του μηχανισμού εκτέλεσης της Prolog, θα κάνουμε μια παρένθεση για να παρουσιάσουμε την διαδικασία ενοποίησης δύο ατομικών τύπων ή όρων στην Prolog, η οποία χρησιμοποιείται από τον μηχανισμό εκτέλεσης. Όπως είδαμε και στο Κεφάλαιο 3 της Λογικής, η ύπαρξη μεταβλητών είτε στην κλήση, είτε στις προτάσεις του λογικού προγράμματος, κάνει απαραίτητη την ύπαρξη μιας διαδικασίας η οποία θα εξετάζει (α) αν μία κλήση μπορεί να "ταιριάξει" (να γίνει όμοια) με ένα γεγονός ή την κεφαλή ενός κανόνα και (β) αν ταιριάζει, ποιες αντικαταστάσεις μεταβλητών με τιμές (όρους) είναι απαραίτητες. Η διαδικασία αυτή ονομάζεται *ενοποίηση* (*unification*). Για παράδειγμα, η κλήση:

```
?- follows(ilias,X).
```

μπορεί να ενοποιηθεί με το γεγονός:

```
follows(ilias,petros).
```

με την αντικατάσταση *{X/petros}*.

Ο ενοποιητής (unifier) δύο εκφράσεων φ_1 και φ_2, είναι μια αντικατάσταση θ τέτοια ώστε η έκφραση $\varphi_1\theta$ να είναι συντακτικά όμοια με τη $\varphi_2\theta$. Αν για παράδειγμα φ_1= *follows(ilias,X)* και φ_2=*follows(Y,petros)* ο ενοποιητής θ=*{X/petros, Y/ilias}*, ταυτίζει τους δύο όρους γιατί:

$\varphi_1\theta=\varphi_2\theta$: *follows(ilias,X) {X/petros,Y/ilias} = follows(Y, petros) {X/petros,Y/ilias}*

δηλαδή, follows(ilias,petros)=follows(ilias, petros).

Αν υπάρχει μια τέτοια αντικατάσταση οι εκφράσεις φ_1 και φ_2 ονομάζονται *ενοποιήσιμες* (*unifiable*). Υπάρχουν πολλοί δυνατοί ενοποιητές ανάμεσα σε δύο ή περισσότερες εκφράσεις, όμως εκείνος που παρουσιάζει ενδιαφέρον είναι ο γενικότερος ενοποιητής (mgu-most general unifier), ο οποίος ενοποιεί τις εκφράσεις με τις λιγότερες δυνατές αντικαταστάσεις και είναι μοναδικός.

Η εύρεση του *γενικότερου ενοποιητή* (*most general unifier*) ανάμεσα σε δύο εκφράσεις βρίσκεται με τον ακόλουθο αναδρομικό αλγόριθμο:

1. Δύο σταθερές ενοποιούνται αν και μόνο αν είναι ίδιες.
2. Μια μεταβλητή ενοποιείται με οποιονδήποτε όρο, εισάγοντας μια νέα αντικατάσταση στο γενικότερο ενοποιητή.
3. Δύο συναρτησιακοί όροι ενοποιούνται αν έχουν το ίδιο συναρτησιακό σύμβολο, την ίδια τάξη και αν κάθε όρισμα του πρώτου μπορεί να ενοποιηθεί με το αντίστοιχο σε θέση όρισμα του δεύτερου όρου.
4. Δύο ατομικοί τύποι ενοποιούνται αν έχουν το ίδιο κατηγόρημα, την ίδια τάξη και αν κάθε όρισμα του πρώτου μπορεί να ενοποιηθεί με το αντίστοιχο σε θέση όρισμα του δεύτερου ατομικού τύπου.

Στον Πίνακα 5.1 φαίνονται όλες οι περιπτώσεις ενοποίησης δύο όρων. Η περίπτωση της ενοποίησης δύο ατομικών τύπων είναι ολόιδια με την ενοποίηση δύο σύνθετων όρων, άρα αντιπροσωπεύεται από το κάτω-δεξιά κελί του πίνακα.

Πίνακας 5.1: *Περιπτώσεις ενοποίησης δύο όρων.*

Όροι	*<σταθερά>* c_1	*<μεταβλητή>* X_1	*<σύνθετος όρος>* $f(t_1,\ldots,t_\nu)$
<σταθερά> c_2	επιτυχές αν η c_1 είναι ίδια με τη c_2	η μεταβλητή X_1 παίρνει την τιμή c_2 $\{X_1=c_2\}$	αποτυγχάνει
<μεταβλητή> X_2	η μεταβλητή X_2 παίρνει την τιμή c_1 $\{X_2=c_1\}$	η μεταβλητή X_1 ενοποιείται με τη X_2 $\{X_1=X_2\}$	επιτυχές $\{X_2=f(t_1,\ldots,t_m)\}$
<σύνθετος όρος> $f(u_1,\ldots,u_\mu)$	αποτυγχάνει	επιτυχές $\{X_1=f(u_1,\ldots,u_\mu)\}$	επιτυχές αν $\nu=\mu$ και $u_1=t_1,\ldots,u_\mu=t_\nu$

Στη συνέχεια, στον Πίνακα 5.2, παρουσιάζουμε διάφορα παραδείγματα ενοποίησης δύο ατομικών τύπων ή σύνθετων όρων (δεν υπάρχει συντακτική διαφορά).

Πίνακας 5.2: *Παραδείγματα ενοποίησης ατομικών τύπων / σύνθετων όρων.*

1ος τύπος	**2ος τύπος**	**Αποτέλεσμα ενοποίησης /**

		επεξήγηση
follows(ilias, petros)	**follows(ilias, petros)**	**true**
follows(ilias, petros)	**follows(ilias, nikos)**	**false** τα δεύτερα ορίσματα είναι διαφορετικές σταθερές
follows(ilias, X)	**follows(ilias, petros)**	**{X=petros}**
follows(ilias, X)	**follows(demos, petros)**	**false** τα πρώτα ορίσματα είναι διαφορετικές σταθερές
follows(ilias, X)	**follows(Y, petros)**	**{X=petros, Y=ilias}**
follows(ilias, X)	**follows(X, petros)**	**false** η μεταβλητή X δεν μπορεί να ενοποιηθεί και με τις δύο σταθερές ταυτόχρονα
follows(ilias, X)	**friends(ilias, petros)**	**false** οι δύο τύποι έχουν διαφορετικό κατηγόρημα
friends(ilias, petros)	**friends(ilias, petros, demos)**	**false** οι δύο τύποι έχουν διαφορετική τάξη
follows(ilias, X)	**follows(ilias, Y)**	**{X=Y}**
follows(ilias, ilias)	**follows(X, X)**	**{X=ilias}**
follows(ilias, ilias)	**follows(X, Y)**	**{X=Y=ilias}**
follows(ilias, Y)	**follows(X, X)**	**{X=Y=ilias}**
follows(ilias, petros)	**follows(X, X)**	**false** η μεταβλητή X δεν μπορεί να ενοποιηθεί και με τις δύο σταθερές ταυτόχρονα
a(f(1,X), 3)	**a(f(Y,k), Z)**	**{Y=1,X=k,Z=3}**
a(3, X)	**a(Y, f(2, Y))**	**{Y=3,X=f(2,3)}** Ο γενικότερος ενοποιητής θα πρέπει να είναι όσο πιο συγκεκριμένος γίνεται
a(f(1, 2), 3)	**a(f(Y, 2, 3), Z)**	**false** Τα πρώτα ορίσματα είναι σύνθετοι όροι διαφορετικής

		τάξης.
`f(Z, g(1))`	`f(a, Z)`	**`false`** Η μεταβλητή Z δεν μπορεί να ενοποιηθεί ταυτόχρονα με την σταθερά a και τον σύνθετο όρο g(1)

Θα μπορούσε κάποιος να δοκιμάσει τα παραπάνω παραδείγματα απευθείας στην Prolog χρησιμοποιώντας το τελεστή ενοποίησης =/2 για να πάρει ακριβώς τις ίδιες απαντήσεις. Για παράδειγμα:

```
?- a(f(1,X),3)=a(f(Y,k),Z).
```

5.3 Μηχανισμός Εκτέλεσης της Prolog

Η *εκτέλεση* ενός προγράμματος στην Prolog ξεκινά με μια *ερώτηση* ή *πρόταση-στόχο* προς απόδειξη που υποβάλλει ο χρήστης. Η ερώτηση μπορεί να είναι (α) *απλή*, δηλαδή να περιέχει μία μόνο *κλήση*:

```
?- friends(petros,X).
```

ή (β) *σύνθετη*, δηλαδή να αποτελείται από σύζευξη πολλαπλών κλήσεων:

```
?- friends(petros,X), friends(ilias,X).
```

Η εκτέλεση φτάνει σε *λύση* όταν έχουν εξαντληθεί όλες οι κλήσεις της ερώτησης. Η *απάντηση* στην ερώτηση είναι το *αποτέλεσμα* του προγράμματος. Η απάντηση μπορεί να είναι απλά καταφατική όταν η ερώτηση δεν περιέχει μεταβλητές:

```
?- friends(ilias,petros).
true
```

Όταν η ερώτηση περιέχει *μεταβλητές*, τότε η απάντηση περιέχει μία *ανάθεση* τιμών στις μεταβλητές αυτές για την οποία αληθεύει η ερώτηση:

```
?- friends(ilias,X).
X = petros
```

Και στις δύο παραπάνω περιπτώσεις, όταν δεν μπορεί να αποδειχθεί η αλήθεια της ερώτησης, η απάντηση είναι *αρνητική*:

```
?- friends(nick,X).
false
?- friends(nick,petros).
false
```

Περιγραφή του Μηχανισμού Εκτέλεσης

Θα θεωρήσουμε αρχικά την περίπτωση όπου η Prolog προσπαθεί να αποδείξει την αλήθεια μιας απλής πρότασης, δηλαδή μιας πρότασης-στόχο με έναν μόνο υποστόχο (κλήση). Για την απόδειξή της ο μηχανισμός εκτέλεσης προσπαθεί να την "ταιριάξει" με ένα κατηγόρημα, δηλαδή με ένα γεγονός ή την κεφαλή ενός κανόνα. Η περίπτωση να ενοποιείται ο υποστόχος με ένα γεγονός είναι η απλούστερη: ένα γεγονός εκφράζει την αλήθεια μιας σχέσης χωρίς συνθήκες, άρα αν η ερώτηση ενοποιείται με ένα από τα γεγονότα του προγράμματος, τότε αποδεικνύεται η αλήθεια της χωρίς άλλη αναζήτηση. Στο παράδειγμα του κοινωνικού δικτύου, τα γεγονότα follows/2 εκφράζουν τη μονόδρομη σχέση φιλίας μεταξύ δύο χρηστών:

```
follows(ilias, petros).
follows(petros, ilias).
follows(petros, demos).
follows(petros, sofia).
```

Η κλήση ?- follows(ilias,X) ενοποιείται με το πρώτο γεγονός με την ανάθεση {X = petros}, άρα δεν απαιτεί τίποτε άλλο για να αποδειχθεί η αλήθεια της:

```
?-follows(ilias,X).
```

```
X = petros
```

Αν ο υποστόχος ενοποιείται με περισσότερες λογικές προτάσεις τότε επιλέγεται η πρώτη από αυτές, δηλαδή εκείνη που βρίσκεται υψηλότερα ή πρωθύστερα στον κώδικα του προγράμματος. Τα σημεία της εκτέλεσης όπου γίνονται τέτοιου είδους επιλογές (choice points) καταχωρούνται κατάλληλα από τον μηχανισμό εκτέλεσης. Επίσης καταχωρούνται και οι εναλλακτικές επιλογές, δηλαδή η ενοποίηση του υποστόχου με την δεύτερη, τρίτη πρόταση, κλπ., επειδή αυτές οι εναλλακτικές μπορεί να οδηγήσουν σε διαφορετικές αποδείξεις του ιδίου υποστόχου. Η εκτέλεση θα επιστρέψει σε αυτά τα σημεία σε περίπτωση που απαιτηθεί μια εναλλακτική απάντηση (οπισθοδρόμηση). Για παράδειγμα η ερώτηση ?-follows(petros,X), ενοποιείται με το δεύτερο, το τρίτο και το τέταρτο γεγονός και σύμφωνα με όσα αναφέρθηκαν ο μηχανισμός θα επιλέξει το πρώτο:

```
?- follows(petros,X).
X = ilias
```

Στο παραπάνω παράδειγμα εκτέλεσης, δεν ζητήθηκαν εναλλακτικές λύσεις. Αν όμως ο χρήστης πληκτρολογήσει τον χαρακτήρα ‘;’, τότε ενεργοποιείται ο *μηχανισμός οπισθοδρόμησης* για να δώσει την επόμενη (εναλλακτική) λύση. Ο μηχανισμός οπισθοδρόμησης επισκέπτεται τα σημεία επιλογής (τα οποία ονομάζονται για αυτό και *σημεία οπισθοδρόμησης*), και επιβάλει να γίνει ενοποίηση του υποστόχου με την επόμενη ενοποιήσιμη λογική πρόταση. Προφανώς, για να γίνει κάτι τέτοιο, όλες οι αναθέσεις τιμών οι οποίες έγιναν από το συγκεκριμένο σημείο επιλογής και μετά ακυρώνονται. Έτσι η ερώτηση:

```
?- follows(petros,X).
X = ilias ;
X = demos ;
X = sofia
```

επιστρέφει τις τρεις διαφορετικές απαντήσεις για τη μεταβλητή X, οι οποίες αντιστοιχούν στα τρία γεγονότα, με την σειρά που εμφανίζονται στον κώδικα του προγράμματος.

Στην περίπτωση των σύνθετων ερωτήσεων, ο μηχανισμός αποδεικνύει κάθε φορά έναν υποστόχο ξεκινώντας από εκείνον που βρίσκεται πρώτος από αριστερά. Αν ο υποστόχος είναι αληθής, τότε ο μηχανισμός προσπαθεί να αποδείξει τον επόμενο υποστόχο. Αν θεωρήσουμε ότι η σύνθετη ερώτηση είναι μια λίστα από υποστόχους προς απόδειξη, κάθε φορά ο μηχανισμός προσπαθεί να αποδείξει τον πρώτο υποστόχο στην λίστα. Αν ο πρώτος υποστόχος είναι αληθής, τότε αφαιρείται από την λίστα και η διαδικασία συνεχίζει με τον επόμενο. Για παράδειγμα, αν τεθεί η ερώτηση:

```
?- follows(ilias,X), follows(petros,Y).
```

η αρχική λίστα των υποστόχων είναι:

```
[follows(ilias,X), follows(petros,Y)]
```

Εφόσον αποδειχθεί (όπως παραπάνω) ότι ο πρώτος υποστόχος είναι αληθής για {X=petros}, η τρέχουσα λίστα στόχων γίνεται:

```
[follows(petros,Y)] {X/petros}
```

και έπειτα από την απόδειξη του follows(petros,Y) για {Y = ilias}:

```
[] {X/petros, Y/ilias}
```

Εφόσον η λίστα είναι κενή, δεν απομένει τίποτε προς απόδειξη, άρα η σύνθετη πρόταση είναι αληθής. Στα προηγούμενα σημειώσαμε δίπλα σε κάθε λίστα υποστόχων και τις αντικαταστάσεις των μεταβλητών της αρχικής ερώτησης. Αυτές οι αντικαταστάσεις επιστρέφονται σαν απάντηση στην ερώτηση:

```
?- follows(ilias,X), follows(petros,Y).
X = petros,
Y = ilias
```

Αλλά ο υποστόχος follows(petros,Y), έχει και άλλες εναλλακτικές λύσεις, δηλαδή μπορεί να ενοποιηθεί με το τρίτο και τέταρτο γεγονός. Έτσι, όπως και πριν, αν ζητηθούν εναλλακτικές λύσεις (‘;’), ο μηχανισμός οπισθοδρόμησης θα επιστρέψει στο σημείο:

```
[follows(petros,Y)] {X/petros}
```

και θα αποδείξει τον στόχο follows(petros,Y) για {Y = demos}:

```
[] {X/petros, Y/demos}
```

επιστρέφοντας την εναλλακτική απάντηση:

```
...
X = petros,
Y = demos
```

Ομοίως η τρίτη εναλλακτική παράγεται για {Y=sofia}:

```
X = petros,
Y = sofia
```

Η παραπάνω διαδικασία παρουσιάζεται σε αυτό το βίντεο.

Τέλος, η διαδικασία εκτέλεσης είναι παρόμοια και στην περίπτωση ενοποίησης της ερώτησης με ένα κανόνα. Για παράδειγμα ο κανόνας:

```
friends(X,Y):-
  follows(X,Y),
  follows(Y,X).
```

περιγράφει την σχέση friends/2. Στην ερώτηση:

```
?- friends(X,Y).
```

ο μηχανισμός εκτέλεσης θα δημιουργήσει την λίστα στόχων:

```
[friends(X,Y)]
```

της οποίας ο μοναδικός υποστόχος ενοποιείται με την κεφαλή του κανόνα. Στην περίπτωση αυτή, ο στόχος αντικαθίσταται από το σώμα του κανόνα, και η λίστα υποστόχων γίνεται:

```
[follows(X,Y), follows(Y,X)]
```

Η διαδικασία συνεχίζεται με τον τρόπο που περιγράφηκε παραπάνω, αποδεικνύοντας κάθε υποστόχο από αριστερά προς τα δεξιά. Έτσι ο πρώτος υποστόχος ενοποιείται με το γεγονός follows(ilias,petros) και οι αντικατάστασεις για τις μεταβλητές X και Y εφαρμόζονται στου υπόλοιπους υποστόχους:

```
[follows(petros,ilias)] {X/ilias, Y/petros}
```

Ο τελευταίος υποστόχος ενοποιείται με το δεύτερο γεγονός που δόθηκε παραπάνω, επιστρέφοντας τη λύση:

```
?- friends(X,Y).
X = petros,
Y = ilias.
```

Η παραπάνω διαδικασία απόδειξης, η οποία χρησιμοποιεί ένα κανόνα για την απάντηση στην ερώτηση του χρήστη, παρουσιάζεται σε αυτό το βίντεο.

Η περιγραφή που δόθηκε στην παράγραφο αποτελεί μια άτυπη περιγραφή του μηχανισμού εκτέλεσης. Στα επόμενα θα παρουσιαστεί ο μηχανισμός περισσότερο αυστηρά, αναλύοντας όλες τις λεπτομέρειες της εκτέλεσης.

Ο Αλγόριθμος του Μηχανισμού Εκτέλεσης

Ο μηχανισμός εκτέλεσης της Prolog είναι ένας αλγόριθμος που διαχειρίζεται τις κλήσεις μιας ερώτησης ως μία στοίβα (stack) και εκτελείται έως ότου αδειάσει η στοίβα, οπότε και τερματίζει με επιτυχία. Συγκεκριμένα, ο αλγόριθμος έχει ως εξής:

1. Έως ότου δεν υπάρχουν άλλες κλήσεις στην ερώτηση ?- a_1, a_2, …, a_n, επιλέγονται με τη σειρά, από αριστερά προς τα δεξιά, οι κλήσεις a_i της ερώτησης.
    a. Για κάθε κλήση a_i της ερώτησης, ο μηχανισμός εκτέλεσης αναλαμβάνει να βρει μια πρόταση του προγράμματος της οποίας η κεφαλή να *ενοποιείται* με την επιλεγμένη κλήση.
        i. Αν η κλήση a_i ενοποιείται με ένα από τα *γεγονότα* του προγράμματος, τότε αυτή ικανοποιείται και απομακρύνεται από την ερώτηση. Αυτό υποδηλώνει ότι η αλήθεια του ερωτήματος a_i αποδεικνύεται άμεσα.
        ii. Αν η κλήση a_i ενοποιείται με κάποιον *κανόνα*, τότε αυτή απομακρύνεται από την ερώτηση και την θέση της παίρνει το σώμα του κανόνα αυτού. Αυτό υποδηλώνει ότι

για την ικανοποίηση της αρχικής κλήσης είναι απαραίτητη η ικανοποίηση των κλήσεων του σώματος του κανόνα (προϋποθέσεων του κανόνα) που την αντικατέστησε.

iii. Αν υπάρχουν περισσότερες της μιας προτάσεις με τις οποίες μπορεί εν δυνάμει να ενοποιηθεί η κλήση a_i, τότε ενοποιείται με την πρώτη από αυτές. Το σημείο αυτό του προγράμματος σημειώνεται ως *σημείο οπισθοδρόμησης*, προστίθεται σε μία στοίβα σημείων οπισθοδρόμησης, και αντιπροσωπεύει πιθανές εναλλακτικές "απαντήσεις" στην κλήση.

b. Αν η κλήση a_i δεν μπορεί να ενοποιηθεί με καμία πρόταση (*σημείο αποτυχίας*), τότε ενεργοποιείται ο *μηχανισμός οπισθοδρόμησης* (*backtracking*)

i. Αν η στοίβα σημείων οπισθοδρόμησης δεν είναι κενή, τότε ο μηχανισμός *οπισθοδρόμησης* (α) επιστρέφει την εκτέλεση στο πιο πρόσφατο σημείο οπισθοδρόμησης α_j με $j<i$ (το πρώτο στοιχείο της στοίβας σημείων οπισθοδρόμησης), (β) ακυρώνει τα υπολογιστικά βήματα (ενοποιήσεις, αντικαταστάσεις - διαγραφές κλήσεων) μεταξύ του σημείου οπισθοδρόμησης και του σημείου αποτυχίας και (γ) αναζητά στις επόμενες προτάσεις, κάποια που να μπορεί να ενοποιηθεί με την κλήση α_j.

ii. Αν η στοίβα σημείων οπισθοδρόμησης είναι κενή, τότε τερματίζει η εκτέλεση του προγράμματος με αποτυχία (*fail*). Εκτύπωση false, έξοδος από το βρόχο και μετάβαση στο βήμα 3.

2. Όταν η στοίβα των κλήσεων μείνει κενή (□), η ερώτηση θεωρείται ότι απαντήθηκε επιτυχώς.

a. Αν υπάρχουν μεταβλητές στην αρχική κλήση, τότε επιστρέφεται οι τιμές των μεταβλητών οι οποίες προέκυψαν από τις ενοποιήσεις των κλήσεων στα βήματα 1.a.i και 1.a.ii, παραπάνω.

i. Αν η στοίβα των σημείων οπισθοδρόμησης δεν είναι κενή, τότε δίπλα από την απάντηση η Prolog περιμένει την είσοδο κάποιου χαρακτήρα από τον χρήστη.

1. Αν δοθεί ο χαρακτήρας '`;`', τότε ενεργοποιείται ο μηχανισμός οπισθοδρόμησης και ζητείται μία εναλλακτική απάντηση στο αρχικό ερώτημα, δηλαδή μία εναλλακτική ανάθεση τιμών στις μεταβλητές του ερωτήματος, για τις οποίες η αρχική ερώτηση επίσης αληθεύει. Αυτό προκαλεί την επιστροφή της εκτέλεσης στο βήμα 1.b.

2. Αν δοθεί ο χαρακτήρας enter, τότε μετάβαση στο βήμα 3.

b. Αν δεν υπάρχουν μεταβλητές στην αρχική κλήση, επιστρέφεται true και μετάβαση στο βήμα 3.

3. Η Prolog τερματίζει και επιστρέφει στο προτρεπτικό σήμα '`?-`'.

Στον παραπάνω αλγόριθμο, στο βήμα 2.a.i.1, η αναζήτηση για εναλλακτικές λύσεις από τον χρήστη μπορεί να γίνεται συνεχώς έως ότου η στοίβα των σημείων οπισθοδρόμησης μείνει κενή (σημείο 1.b.ii του αλγορίθμου), συνεπώς η εκτέλεση θα τερματιστεί με αποτυχία (false). Στην συνέχεια θα εξηγήσουμε την λειτουργία του παραπάνω αλγορίθμου μέσω παραδειγμάτων.

Παράδειγμα 1

Έστω το λογικό πρόγραμμα του κοινωνικού δικτύου που παρουσιάστηκε στην ενότητα 4.5. Ως πρώτο παράδειγμα, έστω το ακόλουθο ερώτημα, χωρίς μεταβλητές:

```
?- friend_same_gender(ilias,petros).          % Αρχική Κλήση
```

Η αρχική κλήση είναι απλή, συνεπώς η πλέον αριστερή κλήση είναι και η μοναδική. Στο βήμα 1.a η κλήση ενοποιείται με τον ακόλουθο κανόνα (1.a.ii), καθώς δεν υπάρχουν γεγονότα για το συγκεκριμένο κατηγόρημα:

```
friend_same_gender(X,Y):-
  female(X),
  friends(X,Y),
  female(Y).
```

Από την ενοποίηση προκύπτουν οι αναθέσεις τιμών {X=ilias, Y=petros}, συνεπώς στο βήμα 1.a.ii η αρχική κλήση αντικαθίσταται από το σώμα του παραπάνω κανόνα:

```
?- female(ilias), friends(ilias,petros), female(petros).        % Κλήση 1
```

Λόγω του σημείου 1.a.iii και της ύπαρξης δεύτερου κανόνα για το κατηγόρημα friend_same_gender/2, η αρχική κλήση τοποθετείται στην κορυφή της στοίβας των σημείων οπισθοδρόμησης, η οποία είναι αρχικά κενή. Η εκτέλεση επανέρχεται στην κορυφή του βρόχου (βήμα 1) και στο βήμα γίνεται προσπάθεια ενοποίησης της πιο αριστερής κλήσης ?-female(ilias), η οποία αποτυγχάνει καθώς δεν υπάρχει κάποια λογική πρόταση που να μπορεί να ενοποιηθεί. Έτσι στο βήμα 1.b ενεργοποιείται ο μηχανισμός οπισθοδρόμησης, ο οποίος ανακαλεί από την στοίβα το πρώτο (και μοναδικό) σημείο οπισθοδρόμησης στην αρχική κλήση (βήμα 1.b.i). Αυτό σημαίνει ότι ακυρώνεται η ενοποίηση της αρχικής κλήσης με τον πρώτο κανόνα του και η ανάθεση τιμών στις μεταβλητές X και Y.

Η εκτέλεση επανέρχεται στην κορυφή του βρόχου 1 με την αρχική κλήση, η οποία τώρα στο σημείο 1.a.ii ενοποιείται με τον δεύτερο κανόνα του friend_same_gender/2:

```
friend_same_gender(X,Y):-
  male(X),
  friends(X,Y),
  male(Y).
```

και λόγω των αντικαταστάσεων {X=ilias,Y=petros} παίρνουμε την νέα κλήση:

```
?- male(ilias), friends(ilias,petros), male(petros).        % Κλήση 2
```

Στη συνέχεια δοκιμάζεται και πάλι να ενοποιηθεί η πιο αριστερή κλήση ?-male(ilias), η οποία ενοποιείται επιτυχώς με αντίστοιχο γεγονός, συνεπώς στο σημείο 1.a.i απομακρύνεται η κλήση αυτή και πλέον η κλήση γίνεται:

```
?- friends(ilias,petros), male(petros).        % Κλήση 3
```

Η πιο αριστερή κλήση ?-friends(ilias,petros) ενοποιείται με τον μοναδικό κανόνα του κατηγορήματος friends/2:

```
friends(X,Y):-
  follows(X,Y),
  follows(Y,X).
```

Εδώ θα πρέπει να σημειωθεί ότι οι μεταβλητές X και Y του παραπάνω κανόνα είναι διαφορετικές από τις αντίστοιχες του προηγούμενου κανόνα friend_same_gender/2 της προηγούμενης ενοποίησης. Αυτό εσωτερικά στην Prolog εξασφαλίζεται με την αντιστοίχηση των μεταβλητών κάθε πρότασης σε μοναδικά "εσωτερικά" ονόματα, ώστε να μην προκύπτει σύγχυση. Στην παρουσίασή μας όμως, απλά θα μετονομάσουμε τις μεταβλητές σε X' και Y':

```
friends(X',Y'):-
  follows(X',Y'),
  follows(Y',X').
```

Έτσι, λόγω των αντικαταστάσεων {X'=ilias,Y'=petros} και του σημείου 1.a.ii του αλγορίθμου, η κλήση αντικαθίσταται από το σώμα του κανόνα:

```
?-follows(ilias,petros), follows(petros,ilias), male(petros).        % Κλήση 4
```

Εδώ παρατηρούμε ότι η σύνθετη κλήση πλέον περιέχει απλές κλήσεις που προέρχονται από διαφορετικές φάσεις της εκτέλεσης του αλγορίθμου, δηλαδή από σώματα διαφορετικών κανόνων. Αυτό δεν θα πρέπει να φαίνεται παράξενο, καθώς προκύπτει από την επαναληπτική εφαρμογή ενός απλού αλγόριθμου, ο οποίος για την απόδειξη κάποιας πρότασης απλά ανασκευάζει (rewrite) επαναληπτικά το πρώτο ερώτημα μιας σύνθετης κλήσης χρησιμοποιώντας τις προϋποθέσεις των αντίστοιχων συνεπαγωγικών κανόνων.

Στην συνέχεια, η πιο αριστερή κλήση ?-follows(ilias,petros) ενοποιείται με αντίστοιχο γεγονός και απομακρύνεται από την κλήση (βήμα 1.a.i):

```
?- follows(petros,ilias), male(petros).          % Κλήση 5
```

Λόγω ακριβώς της ίδιας αιτιολόγησης προκύπτει η νέα κλήση:

```
?- male(petros).                                 % Κλήση 6
```

και με νέα εφαρμογή του βήματος 1.a.i προκύπτει η κενή κλήση:

```
?- □.                                            % Κλήση 7
```

Συνεπώς, υπάρχει έξοδος από το βρόχο, τερματισμός με επιτυχία και εκτύπωση της λέξης true (βήμα 2.b).

Στο σημείο αυτό θα πρέπει να παρατηρήσουμε ότι ο μηχανισμός εκτέλεσης της Prolog είναι απολύτως συμβατός με την αποδεικτική διαδικασία της αρχής της ανάλυσης και της εις άτοπο απαγωγής της κατηγορηματικής λογικής, όπως παρουσιάστηκε στο Κεφάλαιο 3. Συγκεκριμένα, έστω η αρχική κλήση του παραδείγματος και ο πρώτος κανόνας του κατηγορήματος friend_same_gender/2, σε μορφή Kowalski:

friend_same_gender(*ilias*,*petros*) → *% Αρχική Κλήση*

female(*X*), *friends*(*X*,*Y*), *female*(*Y*) → *friend_same_gender*(*X*,*Y*) *% Κανόνας*

Όπως είχε εξηγηθεί στο Κεφάλαιο 3, η αρχική κλήση ή ερώτηση ή στόχος αποτελεί ουσιαστικά την άρνηση της προς απόδειξη πρότασης, καθώς αντιστοιχεί στο λεκτικό:

¬*friend_same_gender*(*ilias*,*petros*) *% Αρχική Κλήση*

Εφαρμόζοντας την εναλλακτική διατύπωση της αρχής της ανάλυσης για την μορφή Kowalski, οι παραπάνω προτάσεις (κλήση και κανόνας), συνδυάζονται (αναλύονται) ως εξής:

{X/ilias, Y/petros}(female(X), friends(X,Y), female(Y) →)

ή απλούστερα:

female(ilias), friends(ilias,petros), female(petros) →

το οποίο αντιστοιχεί στην κλήση 1 του παραδείγματος.

Ακόμα και αν δεν υιοθετηθεί η μορφή Kowalski, η αρχική κλήση και ο κανόνας αντιστοιχούν στις παρακάτω προτάσεις στην προτασιακή μορφή της λογικής:

¬*friend_same_gender*(*ilias*,*petros*) *% Αρχική Κλήση*

¬*female(X)* ∨ ¬*friends(X,Y)* ∨ ¬*female(Y)* ∨ *friend_same_gender(X,Y)* *% Κανόνας*

από όπου με εφαρμογή της γενικευμένης μορφής της αρχής της ανάλυσης παίρνουμε το συμπέρασμα:

{X/ilias, Y/petros}(¬female(X) ∨ ¬*friends(X,Y)* ∨ ¬*female(Y))*

ή απλούστερα:

¬*female(ilias)* ∨ ¬*friends(ilias,petros)* ∨ ¬*female(petros)* ⇔

¬*(female(ilias)* ∧ *friends(ilias,petros)* ∧ *female(petros))*

Η τελευταία πρόταση είναι πράγματι η άρνηση μιας σύνθετης πρότασης, δηλαδή μια σύνθετη κλήση.

Παράδειγμα 2

Έστω η ακόλουθη ερώτηση με μεταβλητή:

```
?- male_friends(petros,A).                       % Αρχική Κλήση
```

Η μοναδική απλή κλήση της ερώτησης ενοποιείται με τον μοναδικό κανόνα male_friends/2:

```
male_friends(X,Y):-
  friends(X,Y),
  male(Y).
```

με τις ακόλουθες αναθέσεις τιμών {X=petros,Y=A} και αντικαθίσταται από το σώμα του κανόνα (βήμα 1.a.ii):

```
?- friends(petros,A), male(A).                    % Κλήση 1
```

Η πιο αριστερή κλήση ?-friends(petros,A) ενοποιείται με τον μοναδικό κανόνα friends/2 (βλ. παραπάνω) και προκύπτει η ακόλουθη κλήση, λόγω του βήματος 1.a.ii, με τις αναθέσεις τιμών {X'=petros,Y'=A}:

```
?- follows(petros,A), follows(A,petros), male(A).  % Κλήση 2
```

Η πιο αριστερή κλήση ?-follows(petros,A) ενοποιείται με το γεγονός follows(petros,ilias) και με την ανάθεση {A=ilias}, λόγω του βήματος 1.a.i, απομακρύνεται από την κλήση 2:

```
?- follows(ilias,petros), male(ilias).            % Κλήση 3
```

Εδώ σημειώνουμε πως υπάρχουν και άλλα γεγονότα follows/2 με τα οποία μπορεί να ενοποιηθεί η κλήση ?-follows(petros,A), συνεπώς η κλήση 2 τοποθετείται στην κορυφή της στοίβας των σημείων οπισθοδρόμησης. Η πιο αριστερή κλήση ?-follows(ilias,petros) της σύνθετης κλήσης 3 ενοποιείται με το αντίστοιχο γεγονός και, λόγω του βήματος 1.a.i, απομακρύνεται:

```
?- male(ilias).                                   % Κλήση 4
```

Λόγω της ίδιας αιτιολόγησης προκύπτει η κενή κλήση:

```
?- □.                                             % Κλήση 5
```

Συνεπώς, υπάρχει έξοδος από το βρόχο, τερματισμός με επιτυχία και λόγω της ύπαρξης μεταβλητών στην αρχική ερώτηση, εκτυπώνεται η ανάθεση τιμής στις μεταβλητές αυτές που κατέστησαν επιτυχή την κατάληξη (βήμα 2.a):

```
A=ilias
```

Έστω ότι στο βήμα 2.a.i ο χρήστης ζητάει εναλλακτική λύση πατώντας τον χαρακτήρα ';' (σημείο 2.a.i.1). Τότε ενεργοποιείται ο μηχανισμός οπισθοδρόμησης και επιστρέφουμε στο βήμα 1.b.i, αφαιρώντας από την κορυφή της στοίβας σημείων οπισθοδρόμησης την (μοναδική) κλήση 2 και ακυρώνοντας όλες τις ενοποιήσεις και αναθέσεις τιμών που έχουν γίνει από την κλήση 2 και μετά. Η πρώτη από αριστερά κλήση ?-follows(petros,A) ενοποιείται πλέον με άλλο γεγονός follows(petros,demos) με την ανάθεση {A=demos}. Ακολουθώντας την ίδια ακριβώς πορεία των κλήσεων 3-5 παραπάνω, προκύπτει επιτυχία του ερωτήματος για την εναλλακτική τιμή της μεταβλητής A:

```
A=demos
```

Αν ζητηθεί εκ νέου εναλλακτική τιμή, ακολουθώντας την ίδια λογική, η εκτέλεση θα επιστρέψει στην κλήση 2, θα αναζητηθεί για την κλήση ?-follows(petros,A) άλλο γεγονός προς ενοποίηση, θα γίνει ενοποίηση με το γεγονός follows(petros,sofia) και με την ανάθεση {A=sofia} θα προκύψει η ακόλουθη κλήση:

```
?- follows(sofia,petros), male(sofia).            % Κλήση 9
```

Η πιο αριστερή κλήση ?-follows(sofia,petros) δεν μπορεί να ενοποιηθεί με κανένα γεγονός, συνεπώς λόγω του βήματος 1.b.ii, καθώς η στοίβα σημείων οπισθοδρόμησης είναι άδεια, τερματίζει με αποτυχία η εκτέλεση και τυπώνεται false. Προφανώς αυτό θα πρέπει να ερμηνευθεί ως "δεν υπάρχουν άλλες εναλλακτικές λύσεις" και όχι ως "απέτυχε η αρχική ερώτηση".

Παράδειγμα 3

Το επόμενο παράδειγμα, επιδεικνύει την περίπτωση στην οποία η απάντηση στην αρχική ερώτηση είναι αρνητική. Έστω ότι στο λογικό πρόγραμμα κοινωνικής δικτύωσης θέλουμε να διαπιστώσουμε αν υπάρχουν δύο φίλοι διαφορετικού φύλου:

```
?- female(A), friends(A,B), male(B).              % Αρχική κλήση
```

Η πιο αριστερή κλήση της αρχικής κλήσης μπορεί να ενοποιηθεί με τρία γεγονότα, συνεπώς αρχικά ενοποιείται με το πρώτο από αυτά ({A=sofia}), τοποθετείται στην στοίβα σημείων οπισθοδρόμησης, και προκύπτει η νέα κλήση:

```
?- friends(sofia,B), male(B).                     % Κλήση 1
```

Η πιο αριστερή κλήση ενοποιείται με τον μοναδικό κανόνα του κατηγορήματος friends/2, με τις αναθέσεις τιμών {X=sofia,Y=B}, και το σώμα του κανόνα αντικαθιστά την κλήση:

```
?- follows(sofia,B), follows(B,sofia), male(B).          % Κλήση 2
```

Η πιο αριστερή κλήση έχει μόνο ένα γεγονός με το οποίο μπορεί να ενοποιηθεί ({B=helen}):

```
?- follows(helen,sofia), male(helen).                    % Κλήση 3
```

Η πιο αριστερή κλήση ενοποιείται με το αντίστοιχο γεγονός:

```
?- male(helen).                                          % Κλήση 4
```

Η κλήση 4 όμως δεν μπορεί να ενοποιηθεί με κανένα γεγονός ή κανόνα, συνεπώς αποτυγχάνει και ενεργοποιεί τον μηχανισμό οπισθοδρόμησης, ο οποίος αφαιρεί από την στοίβα το μοναδικό στοιχείο, δηλαδή την αρχική κλήση. Έτσι ακυρώνονται όλες οι αναθέσεις τιμών από την αρχική κλήση και μετά, η κλήση female(A) ενοποιείται με το δεύτερο στη σειρά γεγονός ({Y=helen}) και η αρχική κλήση επανατοποθετείται στην στοίβα:

```
?- friends(helen,B), male(B).                            % Κλήση 5
```

Ακολουθώντας την ίδια πορεία με τις κλήσεις 2-4 παραπάνω, προκύπτει η παρακάτω ακολουθία κλήσεων, η οποία τελικά αποτυγχάνει:

```
?- follows(helen,B), follows(B,helen), male(B).          % Κλήση 6 {B=sofia}
?- follows(sofia,helen), male(sofia).                    % Κλήση 7
?- male(sofia).                                          % Κλήση 8
```

Έτσι ενεργοποιείται εκ νέου ο μηχανισμός οπισθοδρόμησης στην αρχική κλήση (αφού οι κλήσεις 5 και 6 δεν έχουν εναλλακτικές απαντήσεις), με νέα ανάθεση τιμής {A=katerina} στην κλήση ?-female(A). Αυτήν την φορά, η κλήση female(A) δεν έχει άλλες εναλλακτικές προτάσεις που θα μπορούσε να ενοποιηθεί, επομένως δεν μπαίνει ξανά στην στοίβα, η οποία πλέον είναι κενή. Στη συνέχεια, κατ' αναλογία με τις κλήσεις 5-7 παραπάνω, έχουμε την παρακάτω ακολουθία κλήσεων:

```
?- friends(katerina,B), male(B).                         % Κλήση 9
?- follows(katerina,B), follows(B,katerina), male(B).    % Κλήση 10 {B=petros}
?- follows(petros,katerina), male(petros).               % Κλήση 11
```

Η πιο αριστερή κλήση της κλήσης 11 δεν ενοποιείται με κάποιο γεγονός ή κανόνα, συνεπώς αποτυγχάνει. Καθώς δεν υπάρχει στην στοίβα κάποιο σημείο οπισθοδρόμησης, η αρχική κλήση τερματίζεται με αποτυχία (false).

Παράδειγμα 4

Το τελευταίο παράδειγμα αφορά στην καλύτερη κατανόηση της σειράς με την οποία επιστρέφονται οι εναλλακτικές λύσεις μέσω του μηχανισμού οπισθοδρόμησης. Έστω το ακόλουθο πολύ απλό λογικό πρόγραμμα το οποίο αποτελείται από δύο γεγονότα:

```
p(1).
p(2).
```

και το σύνθετο ερώτημα:

```
?- p(X), p(Y).                                           % Αρχική Κλήση
```

Η πρώτη από αριστερά κλήση p(X) ενοποιείται με το πρώτο γεγονός με την ανάθεση {X=1} και ταυτόχρονα η αρχική κλήση τοποθετείται στην στοίβα των σημείων οπισθοδρόμησης, καθώς υπάρχει και δεύτερο γεγονός με το οποίο μπορεί να ενοποιηθεί η p(X). Η νέα κλήση είναι η ακόλουθη:

```
?- p(Y).                                                 % Κλήση 1
```

Η μοναδική πλέον κλήση p(Y) επίσης ενοποιείται με το πρώτο γεγονός με την ανάθεση {Y=1} και ταυτόχρονα η κλήση 1 τοποθετείται στην κορυφή της στοίβας των σημείων οπισθοδρόμησης, πάνω από την αρχική κλήση, καθώς υπάρχει και δεύτερο γεγονός με το οποίο μπορεί να ενοποιηθεί η p(Y). Πλέον έχει μείνει η κενή κλήση, συνεπώς επιστρέφεται το αποτέλεσμα:

```
?- □.                                                    % Κλήση 2

X=1, Y=1
```

Αν ζητηθεί εναλλακτική λύση, ο μηχανισμός οπισθοδρόμησης θα αφαιρέσει από την στοίβα σημείων οπισθοδρόμησης την κορυφή της στοίβας, δηλαδή την κλήση 1, θα ακυρώσει την ανάθεση τιμής στην μεταβλητή Y (μόνο), και θα προσπαθήσει να ενοποιήσει την κλήση 1 με άλλη πρόταση, κάτι που πραγματοποιείται με την ενοποίηση με το δεύτερο γεγονός p(2) και την ανάθεση {Y=2}. Επιστρέφεται το αποτέλεσμα:

```
?- □.                                                    % Κλήση 3

X=1, Y=2
```

Καθώς δεν υπάρχει άλλο γεγονός του κατηγορήματος p/1, η κλήση 1 δεν τοποθετείται ξανά στην στοίβα. Έτσι η στοίβα έχει πλέον μόνο την αρχική κλήση. Αν ζητηθεί και άλλη εναλλακτική λύση, ο μηχανισμός οπισθοδρόμησης θα αφαιρέσει από την στοίβα σημείων οπισθοδρόμησης το στοιχείο στην κορυφή της στοίβας (που είναι και μοναδικό της στοιχείο), δηλαδή την αρχική κλήση, θα ακυρώσει την ανάθεση τιμής και στις δύο μεταβλητές X και Y, και θα προσπαθήσει να ενοποιήσει την πιο αριστερή κλήση p(X) της αρχικής κλήσης με άλλη πρόταση, κάτι που πραγματοποιείται με την ενοποίηση με το δεύτερο γεγονός p(2) και την ανάθεση {X=2}. Στη συνέχεια επαναλαμβάνεται εκ νέου η ακόλουθη κλήση, παρόμοια της κλήσης 1:

```
?- p(Y).                                                 % Κλήση 4
```

Σημειώνεται, πως η αρχική κλήση δεν τοποθετείται εκ νέου στην στοίβα, καθώς δεν υπάρχουν εναλλακτικές λύσεις για την p(X). Στη συνέχεια, η κλήση 4, παρόμοια με την κλήση 1, ενοποιείται με το πρώτο γεγονός ({Y=1}), τοποθετείται στην στοίβα σημείων οπισθοδρόμησης, και επιστρέφεται το αποτέλεσμα:

```
?- □.                                                    % Κλήση 5

X=2, Y=1
```

Αν ζητηθεί εκ νέου οπισθοδρόμηση, αφαιρείται η κλήση 4 από την στοίβα, ακυρώνεται η ανάθεση τιμής στην μεταβλητή Y, ενοποιείται η κλήση p(Y) με το δεύτερο γεγονός ({Y=2}) και επιστρέφεται το αποτέλεσμα:

```
?- □.                                                    % Κλήση 6

X=2, Y=2
```

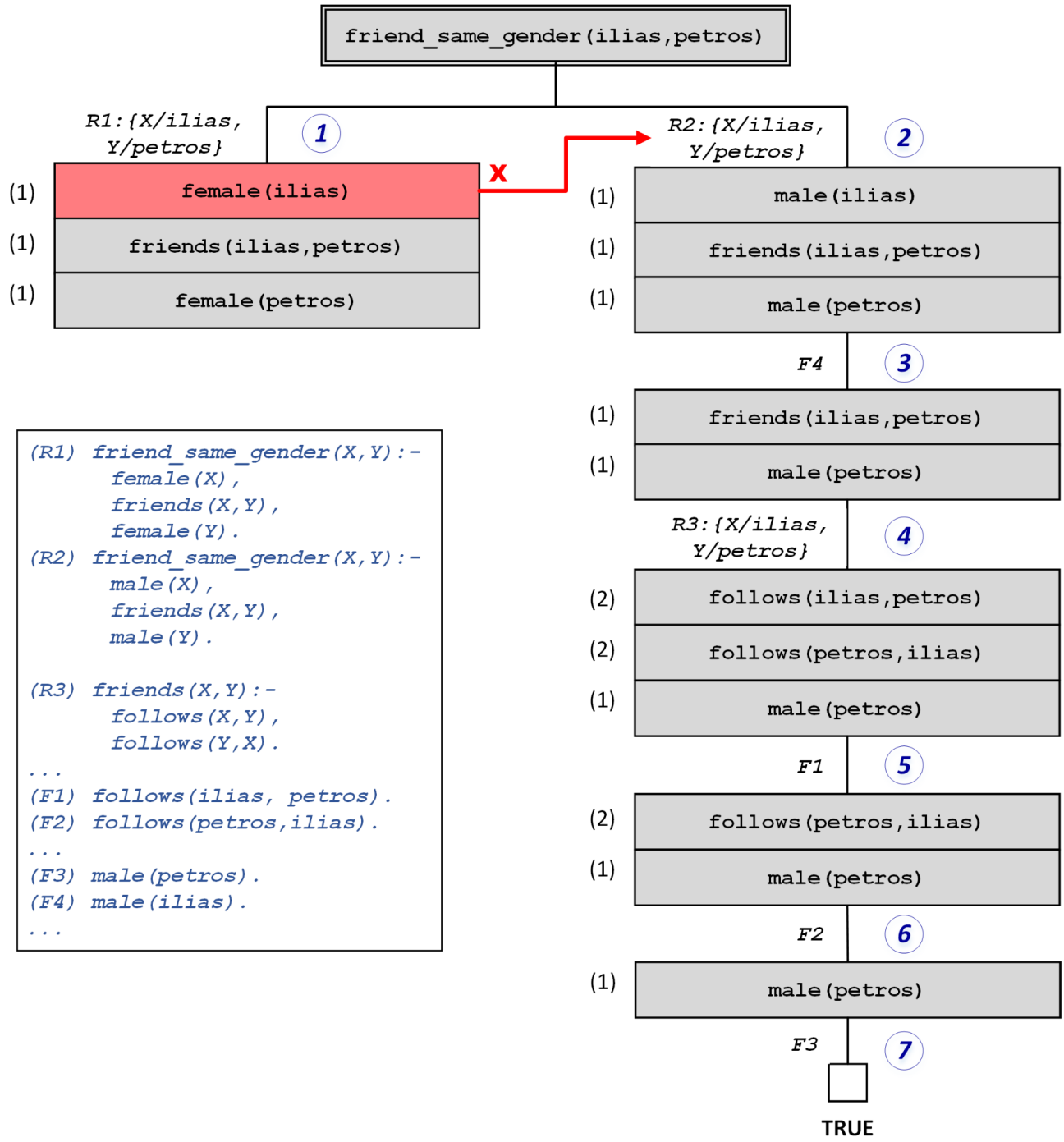
Η κλήση 4 δεν τοποθετείται εκ νέου στην στοίβα, καθώς δεν υπάρχουν εναλλακτικές λύσεις για την p(Y). Έτσι, αν ζητηθεί πάλι εναλλακτική λύση, καθώς η στοίβα σημείων οπισθοδρόμησης είναι πλέον κενή, θα επιστραφεί απάντηση false.

Αυτό λοιπόν που γίνεται φανερό από το παραπάνω παράδειγμα είναι ότι η οπισθοδρόμηση στην Prolog γίνεται πάντα στο πιο κοντινό "χρονικά" σημείο οπισθοδρόμησης, κάτι που οφείλεται στην χρήση στοίβας, καθώς στην στοίβα το πρώτο στοιχείο είναι και το πιο πρόσφατο. Για τον λόγο αυτό συνήθως ονομάζεται και "*χρονολογική οπισθοδρόμηση*" (*chronological backtracking*). Είναι φανερό πως αν αλλάξει κάποιος την δομή στην οποία συσσωρεύονται τα σημεία οπισθοδρόμησης ή την πολιτική πρόσβασης στην δομή αυτή, θα αλλάξει και ο μηχανισμός οπισθοδρόμησης.

5.4 Δένδρο Αναζήτησης / Εκτέλεσης της Prolog

Η διαδικασία της εκτέλεσης ερωτημάτων στην Prolog που παρουσιάστηκε στην προηγούμενη ενότητα μπορεί να αναπαρασταθεί οπτικά με ένα δένδρο εκτέλεσης ή υπολογισμού, το οποίο πολλές φορές αναφέρεται και ως δένδρο αναζήτησης Ουσιαστικά μπορούμε να θεωρήσουμε ότι η Prolog, προκειμένου να βρει μία απάντηση στο ερώτημα, εκτελεί αναζήτηση στον χώρο των καταστάσεων της εκτέλεσης, στον οποίο υπάρχουν τερματικές καταστάσεις, άλλες επιτυχείς (άρα αποτελούν απάντηση στο ερώτημα) και άλλες αποτυχημένες.

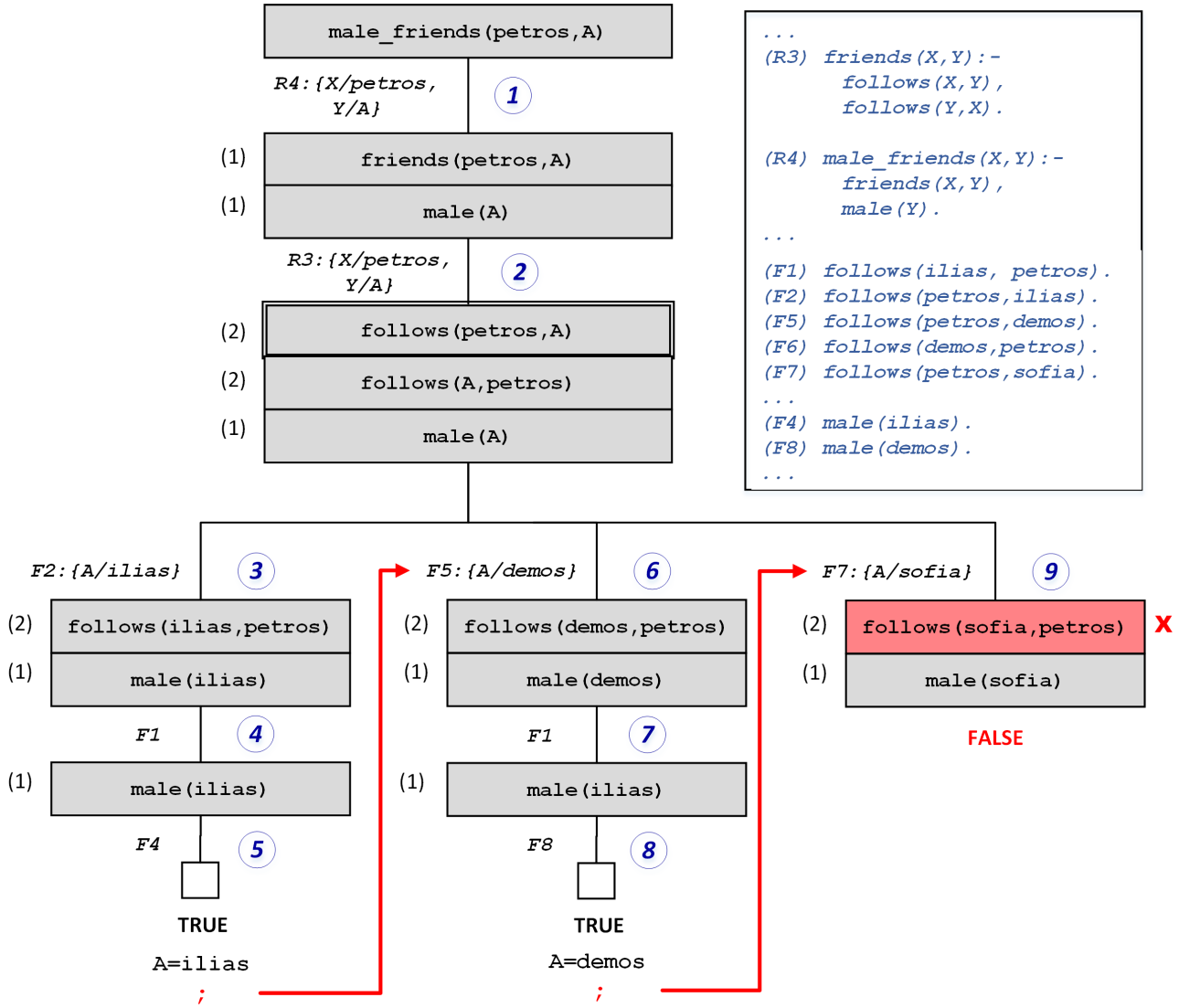
Το δένδρο εκτέλεσης αποτελείται από κόμβους και ακμές (Σχήμα 5.1). Κάθε κόμβος αντιπροσωπεύει μία (σύνθετη) κλήση. Ο κόμβος-ρίζα είναι η αρχική κλήση-ερώτηση που θέτει ο χρήστης στην Prolog. Κάθε άλλος κόμβος του δένδρου προέρχεται από τον μετασχηματισμό της κλήσης του κόμβου-γονέα μέσω του αλγορίθμου που παρουσιάστηκε στην προηγούμενη ενότητα (βήματα 1.a.i και 1.a.ii). Η ακμή που συνδέει δύο κόμβους (τον κόμβο-γονέα με τον κόμβο-παιδί) προέρχεται από την διαδικασία ενοποίησης της πρώτης από αριστερά κλήσης του κόμβου-γονέα και συνοδεύεται από τις αναθέσεις τιμών στις μεταβλητές της κλήσης και της λογικής πρότασης από την οποία προήλθε η ενοποίηση. Αν υπάρχουν εναλλακτικές προτάσεις με τις οποίες ενοποιείται η πιο αριστερή κλήση του κόμβου-γονέα (βήμα 1.a.iii), τότε από τον κόμβο γονέα ξεκινούν τόσες ακμές όσες είναι οι εναλλακτικές προτάσεις και ο κόμβος αυτός αποτελεί σημείο οπισθοδρόμησης και έχει περίμετρο με διπλή γραμμή. Οι τερματικοί κόμβοι στο δένδρο είναι είτε η κενή κλήση (βήμα 2), συνεπώς αντιπροσωπεύουν την επιτυχία απόδειξης του αρχικού ερωτήματος, είτε μη-κενές κλήσεις στις οποίες η πιο αριστερή κλήση δεν μπόρεσε να ενοποιηθεί επιτυχημένα με κάποια πρόταση (βήμα 1.b), συνεπώς αντιπροσωπεύουν την αποτυχία απόδειξης κάποιας ενδιάμεσης κλήσης και την ενεργοποίηση του μηχανισμού οπισθοδρόμησης, εφόσον υπάρχουν σημεία οπισθοδρόμησης στην αντίστοιχη στοίβα (1.b.i), είτε την τελική αποτυχία στην απόδειξη του αρχικού ερωτήματος (βήμα 1.b.ii), αν η στοίβα σημείων οπισθοδρόμησης είναι κενή. Σημειώνεται πως το τερματικό σημείο αποτυχίας της αρχικής κλήσης είναι ο δεξιότερος τερματικός κόμβος αποτυχίας στο δένδρο αναζήτησης.



Σχήμα 5.1: *Δένδρο αναζήτησης / εκτέλεσης του παραδείγματος 1.*

Στο Σχήμα 5.1 φαίνεται το δένδρο αναζήτησης / εκτέλεσης του παραδείγματος 1 της προηγούμενης ενότητας, καθώς και οι σχετικές προτάσεις του λογικού προγράμματος του κοινωνικού δικτύου. Στο σχήμα μπορούμε να παρατηρήσουμε ότι κάθε κόμβος αναπαριστά την σύνθετη κλήση ως στοίβα, όπου η πρώτη (κορυφαία) κλήση αντιστοιχεί στην πιο αριστερή κλήση, όπως παρουσιάστηκε στην προηγούμενη ενότητα. Κάθε κόμβος συνοδεύεται από έναν αριθμό σε κύκλο, ο οποίος αντιπροσωπεύει την σειρά με την οποία επισκέπτεται ο μηχανισμός εκτέλεσης της Prolog τον κόμβο και αντιστοιχεί στον αριθμό της κλήσης, όπως παρουσιάστηκε στην προηγούμενη ενότητα. Ο κόμβος-ρίζα ενοποιείται με δύο κανόνες και αποτελεί σημείο οπισθοδρόμησης, συνεπώς έχει διπλή γραμμή στην περίμετρο. Επίσης, αριστερά από κάθε απλή κλήση

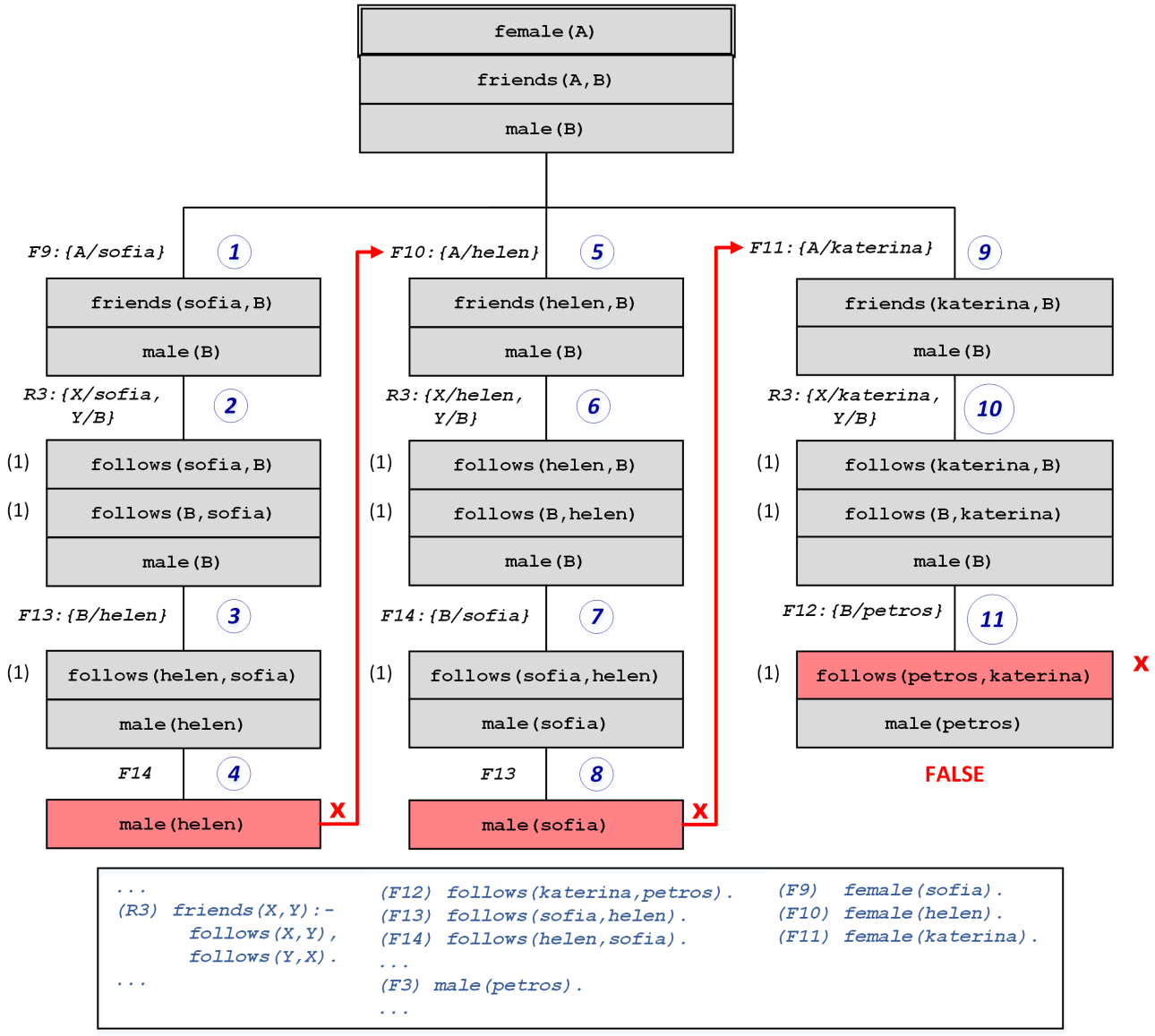
φαίνεται ένας αριθμός σε παρένθεση, ο οποίος εκφράζει το βάθος των αντικαταστάσεων από το οποίο προέκυψε η απλή κλήση. Για παράδειγμα, στο πρώτο επίπεδο του δένδρου κάτω από τον κόμβο-ρίζα (κόμβοι 1 και 2), όλες οι απλές κλήσεις προέκυψαν από την αντικατάσταση της αρχικής κλήσης με έναν από τους κανόνες του κατηγορήματος friend_same_gender/2, συνεπώς πρόκειται για το επίπεδο αντικαταστάσεων (1). Προσέξετε ότι οι δύο κόμβοι είναι εναλλακτικοί, συνεπώς οι αντικαταστάσεις βρίσκονται στο ίδιο βάθος. Παρακάτω στο δένδρο, στον κόμβο 4, οι δύο πρώτες κλήσεις προέκυψαν από την αντικατάσταση της κλήσης friends(ilias,petros) του κόμβου 3, με το σώμα του κανόνα R3, συνεπώς πρόκειται για το επίπεδο αντικαταστάσεων (2). Είναι φανερό ότι το επίπεδο αντικαταστάσεων αυξάνεται μόνο όταν έχει εφαρμογή το βήμα 1.a.ii του αλγορίθμου, δηλαδή όταν γίνεται ενοποίηση της πρώτης κλήσης με κανόνα. Επίσης, αν κάποια κλήση δεν έχει αριθμό δίπλα της, αυτό σημαίνει ότι πρόκειται για την αρχική κλήση (π.χ. βλ. Σχήμα 5.3). Τέλος, στο σχήμα φαίνεται με κόκκινο βέλος, και το σημείο στο οποίο ενεργοποιείται η οπισθοδρόμηση λόγω της αποτυχίας της κλήσης 1 στην αρχική κλήση και στην συνέχεια στην κλήση 2, που αποτελεί εναλλακτική της κλήσης 1.



Σχήμα 5.2: Δένδρο αναζήτησης / εκτέλεσης του παραδείγματος 2.

Στο Σχήμα 5.2 φαίνεται το αντίστοιχο δένδρο αναζήτησης / εκτέλεσης του παραδείγματος 2. Αυτό που ξεχωρίζει σε αυτό το δένδρο είναι η ύπαρξη μεταβλητής στην αρχική κλήση, συνεπώς στους επιτυχείς τερματικούς κόμβους του δέντρου πρέπει να επιστραφούν οι αναθέσεις τιμών στην μεταβλητές της αρχικής κλήσης. Αυτό γίνεται ακολουθώντας το μονοπάτι από τον τερματικό κόμβο ως τον κόμβο-ρίζα συλλέγοντας

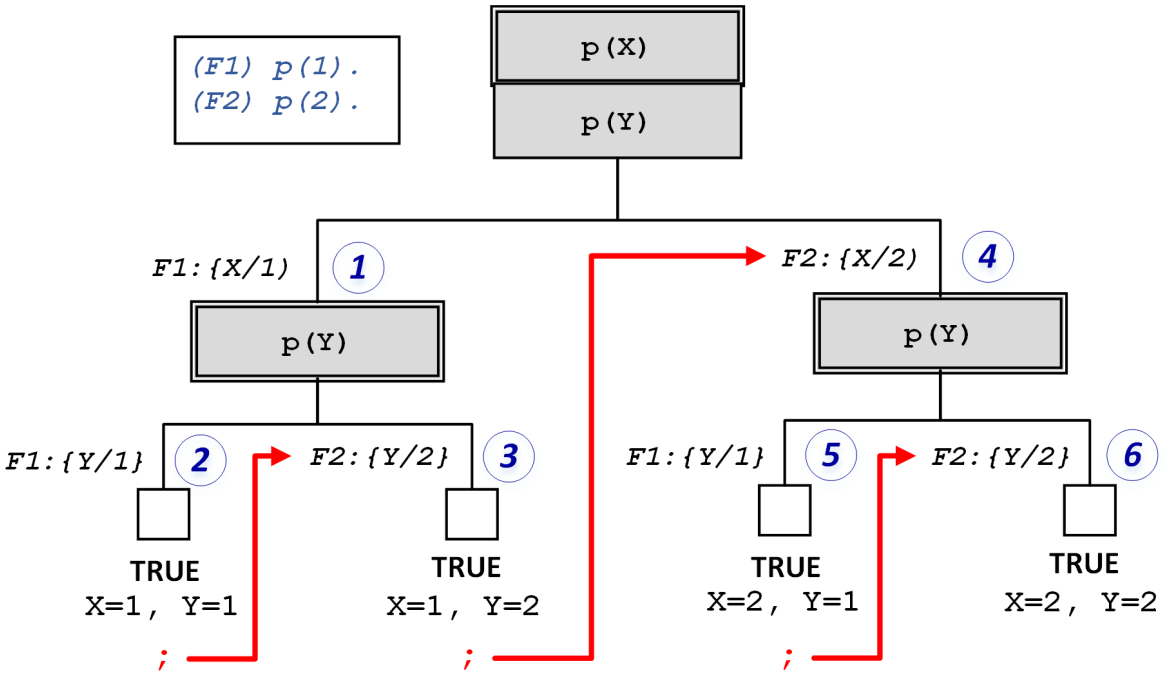
όλες τις αναθέσεις τιμών που αφορούν στις μεταβλητές της αρχικής κλήσης. Τέλος, επιστρέφεται ο πιο γενικός ενοποιητής για τις μεταβλητές αυτές. Στη συγκεκριμένη περίπτωση, η μεταβλητή A της αρχικής κλήσης ενοποιείται με την σταθερά ilias μεταξύ κλήσης 2 και 3. Η συγκεκριμένη ανάθεση επιστρέφεται στον επιτυχή τερματικό κόμβο 5. Στην περίπτωση της επιτυχούς κατάληξης και επιστροφής τιμής, και εφόσον υπάρχει σημείο οπισθοδρόμησης στην στοίβα, υπάρχει η δυνατότητα ο χρήστης να ζητήσει εναλλακτικές απαντήσεις στην αρχική ερώτηση με πάτημα του χαρακτήρα ‘;’. Αυτό αποτυπώνεται στο σχήμα, όπου φαίνεται με κόκκινο βέλος η ενεργοποίηση του μηχανισμού οπισθοδρόμησης στις εναλλακτικές λύσεις της κλήσης 2. Μετά την επιστροφή και της εναλλακτικής απάντησης A=demos (κόμβος 8), η εκ νέου ενεργοποίηση του μηχανισμού οπισθοδρόμησης οδηγεί στην κλήση 9, η οποία δεν μπορεί να ενοποιηθεί με καμία πρόταση και συνεπώς αποτυγχάνει. Καθώς όμως πρόκειται για την πιο δεξιά αποτυχημένη τερματική κλήση στο δένδρο, επιστρέφεται false, επειδή δεν υπάρχει άλλη εναλλακτική λύση για το μοναδικό σημείο οπισθοδρόμησης.



Σχήμα 5.3: *Δένδρο αναζήτησης / εκτέλεσης του παραδείγματος 3.*

Το σημείο της επιστροφής false φαίνεται πιο ξεκάθαρα στο Παράδειγμα 3 της προηγούμενης ενότητας. Στο Σχήμα 5.3 απεικονίζεται το δένδρο αναζήτησης / εκτέλεσης του παραδείγματος, όπου είναι φανερό πως υπάρχουν μόνο ανεπιτυχείς τερματικοί κόμβοι στο δένδρο, αλλά μόνο ο πιο δεξιός ανεπιτυχής κόμβος επιστρέφει false. Οι υπόλοιποι απλά ενεργοποιούν τον μηχανισμό οπισθοδρόμησης.

Γενικώς, για την παρακολούθηση της οπισθοδρόμησης στο δένδρο αναζήτησης ισχύει ότι σε περίπτωση πραγματικής ή τεχνητής αποτυχίας ενοποίησης, ο μηχανισμός εκτέλεσης επιστρέφει στον πρώτο κόμβο-πρόγονο του αποτυχημένου τερματικού κόμβου που συναντάει κινούμενος από κάτω-προς-τα-πάνω, στον οποίο υπάρχει εναλλακτική λύση, δηλαδή διακλάδωση στον δένδρο. Με τον τρόπο αυτό ο μηχανισμός εκτέλεσης της Prolog εκτελεί την διαδικασία της κατά βάθος αναζήτησης (depth-first search) στο δένδρο εκτέλεσης. Αυτό φαίνεται ξεκάθαρα στο Σχήμα 5.4 όπου απεικονίζεται το δένδρο αναζήτησης / εκτέλεσης του παραδείγματος 4. Για παράδειγμα, στον τερματικό κόμβο 2, όταν ζητείται εναλλακτική λύση, ο μηχανισμός οπισθοδρόμησης επιστρέφει στην πιο κοντινή "διχάλα" του δένδρου, κινούμενος από κάτω-προς-τα-πάνω (κλήση 1), δηλαδή στην πιο κοντινή χρονολογικά κλήση με εναλλακτικές λύσεις. Στη συνέχεια, στην τερματική κλήση 3, όπου έχουν εξαντληθεί πλέον οι εναλλακτικές λύσεις της κλήσης 1, η οπισθοδρόμηση οδηγεί στην αμέσως παραπάνω "διχάλα" του δένδρου, που είναι η αρχική κλήση.



Σχήμα 5.4: *Δένδρο αναζήτησης / εκτέλεσης του παραδείγματος 4.*

Τέλος, επισημαίνεται ότι το δένδρο αναζήτησης/εκτέλεσης επ' ουδενί δεν δημιουργείται εσωτερικά - στατικά στην Prolog, ούτε η αναζήτηση γίνεται σε μια στατική δενδρική δομή, όπως παρουσιάστηκε στην ενότητα αυτή για καθαρά εποπτικούς λόγους. Η αναζήτηση στην Prolog γίνεται δυναμικά, κατά την διάρκεια της εκτέλεσης και συνεπώς το υποτιθέμενο δένδρο αναζήτησης δημιουργείται δυναμικά.

5.5 Μηχανισμός Παρακολούθησης της Εκτέλεσης

Ο μηχανισμός εκτέλεσης της Prolog μπορεί να αναπαρασταθεί εκτός από το δένδρο εκτέλεσης / αναζήτησης και με το μοντέλο του κουτιού ελέγχου ροής εκτέλεσης διαδικασίας (procedure box flow control model) ή συντομότερα το "μοντέλο κουτιού" (box model), σύμφωνα με το οποίο η ροή ελέγχου για κάθε στόχο (ή ερώτημα ή κλήση) της Prolog περιλαμβάνει τέσσερις διακριτές φάσεις στον κύκλο ζωής της ή αλλιώς χαρακτηρίζεται από τέσσερα διαφορετικά συμβάντα (events):

1. **CALL**: Σηματοδοτεί την αρχική κλήση του κατηγορήματος. Αντιστοιχεί στο βήμα 1.a του αλγορίθμου που παρουσιάστηκε στην ενότητα 5.3.
2. **EXIT**: Σηματοδοτεί την επιτυχία στην κλήση ενός κατηγορήματος (επιστροφή). Αντιστοιχεί στα βήματα 1.a.i ή 1.a.ii του αλγορίθμου, ανάλογα αν υπήρξε επιτυχής ενοποίηση με γεγονός ή κανόνα, αντίστοιχα.

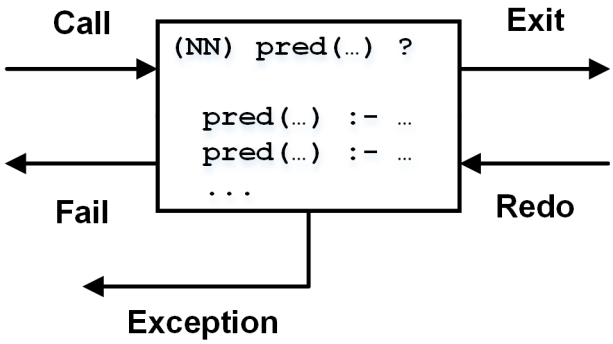
3. **REDO**: Σηματοδοτεί ότι η προηγούμενη κλήση και επιτυχία του κατηγορήματος έχει οδηγήσει την διαδικασία εκτέλεσης σε αποτυχία και οπισθοδρόμηση, και έτσι δοκιμάζεται η επόμενη εναλλακτική πρόταση για την ικανοποίηση της κλήσης. Το ίδιο μπορεί να συμβαίνει και όταν ζητάμε εναλλακτικές λύσεις με την χρήση του πλήκτρου '`;`'. Αντιστοιχεί στο βήμα 1.b.i του αλγορίθμου της ενότητας 5.3.

4. **FAIL**: Σηματοδοτεί ότι η συγκεκριμένη κλήση απέτυχε εντελώς, δηλαδή δεν υπάρχουν προτάσεις που να ικανοποιούν την κλήση ή ότι έχουν εξαντληθεί όλες οι εναλλακτικές λύσεις μέσω οπισθοδρόμησης. Αντιστοιχεί στο βήμα 1.b.ii του αλγορίθμου της ενότητας 5.3.

Συνήθως, σε όλες τις υλοποιήσεις της Prolog, υπάρχει και ένα πέμπτο συμβάν, το οποίο αφορά μόνο στην κλήση των ενσωματωμένων κατηγορημάτων (built-ins), τα οποία δεν μπορούν πάντα να εκτελεστούν με την λογική του μηχανισμού ενοποίησης:

5. **EXCEPTION**: Σηματοδοτεί ότι έχει γίνει κάποιο σφάλμα κατά την εκτέλεση της κλήσης ενός ενσωματωμένου κατηγορήματος, όπως για παράδειγμα χρησιμοποιήθηκε λάθος τύπου όρισμα σε κάποιο κατηγόρημα ή έγινε προσπάθεια εκτέλεσης αριθμητικής πράξης με ελεύθερη μεταβλητή, κ.λ.π. Δεν πρέπει να συγχέεται με την αποτυχία. Συνήθως, αν δεν έχει ληφθεί μέριμνα από τον προγραμματιστή μέσω ειδικών συμβάντων (events) χειρισμού τέτοιων λαθών (exception handling), η εκτέλεση του προγράμματος σταματά με μήνυμα λάθους.

Στο Σχήμα 5.5 φαίνεται το μοντέλο με δύο εισόδους, την αρχική κλήση (CALL) και την επανάκληση σε περίπτωση οπισθοδρόμησης (REDO), και δύο εξόδους επιτυχίας (EXIT) και αποτυχίας (FAIL). Η κατεύθυνση των βελών υποδηλώνει αν η εκτέλεση κινείται από αριστερά προς τα δεξιά, στην κανονική ροή εκτέλεσης, ή από δεξιά προς τα αριστερά, κατά την οπισθοδρόμηση.



Σχήμα 5.5: *Το box model των κλήσεων στην Prolog.*

Επιπρόσθετα, μέσα στο "κουτί" της κλήσης εμφανίζονται:

1. η κλήση αυτή καθ' αυτή στην κορυφή του ορθογωνίου, συνοδευόμενη από την ένδειξη '`?`' και από τον αριθμό **(NN)**, ο οποίος εκφράζει το βάθος των αντικαταστάσεων από το οποίο προέκυψε η κλήση (βλέπε προηγούμενη ενότητα), καθώς και

2. οι πιθανές εναλλακτικές προτάσεις με τις οποίες μπορεί να ενοποιηθεί η κλήση.

Σημειώνεται ότι το παραπάνω εποπτικό μοντέλο χρησιμοποιείται με παραλλαγές σε διάφορες υλοποιήσεις της γλώσσας Prolog και ενδέχεται κάποια από τα συστατικά του στοιχεία να εμφανίζονται διαφοροποιημένα ή καθόλου. Παρόλα αυτά η λογική των τεσσάρων (ή πέντε) συμβάντων συσχετιζόμενων με μία κλήση κατά την διάρκεια της εκτέλεσης ενός προγράμματος Prolog υπάρχει σε όλες τις υλοποιήσεις και με βάση αυτό το μοντέλο λειτουργεί η παρακολούθηση της εκτέλεσης των προγραμμάτων της Prolog.

Στη συνέχεια θα παρουσιάσουμε την παρακολούθηση της εκτέλεσης των παραδειγμάτων εκτέλεσης των προηγούμενων ενοτήτων. Για να ενεργοποιηθεί η παρακολούθηση της εκτέλεσης των προγραμμάτων πρέπει να εκτελεστεί το ενσωματωμένο κατηγόρημα trace:

```
?- trace.
true
```

Στη συνέχεια, όποια κλήση δοθεί αντί να εκτελεστεί άμεσα από την Prolog και να επιστραφεί το αποτέλεσμα, η εκτέλεση θα πραγματοποιηθεί βηματικά και η Prolog θα τυπώνει την κλήση και το συμβάν που σχετίζεται με αυτήν, ανάλογα με την πορεία εξέλιξης της εκτέλεσης. Ο χρήστης θα έχει την δυνατότητα να πληκτρολογήσει κάποιον χαρακτήρα που αφορά στον χειρισμό της διαδικασίας παρακολούθησης της Prolog, ο οποίος πολλές φορές ονομάζεται και αποσφαλματωτής (debugger) καθώς χρησιμεύει κυρίως κατά την διαδικασία αποσφαλμάτωσης των προγραμμάτων της Prolog. Η πιο συνηθισμένη ενέργεια κατά την διάρκεια της παρακολούθησης είναι απλά η συνέχιση της βηματικής παρακολούθησης (creep / ερπυσμός) η οποία υποδηλώνεται με το πάτημα του πλήκτρου '**c**' ή του πλήκτρου enter. Στη συνέχεια θεωρούμε ότι ο χρήστης πατάει συνεχώς το πλήκτρο enter. Στο τέλος της ενότητας θα δοθούν και εναλλακτικές δυνατότητες.

Για το παράδειγμα 1 της ενότητας 5.3, η παρακολούθηση έχει ως εξής:

```
[trace] ?- friend_same_gender(ilias,petros).                    % Αρχική κλήση
  Call: (0) friend_same_gender(ilias, petros) ?creep            % Αρχική κλήση
  Call: (1) female(ilias) ?creep                                % Κλήση 1
  Fail: (1) female(ilias) ?creep                                % Κλήση 1
  Redo: (0) friend_same_gender(ilias, petros) ?creep            % Αρχική κλήση
  Call: (1) male(ilias) ?creep                                  % Κλήση 2
  Exit: (1) male(ilias) ?creep                                  % Κλήση 2
  Call: (1) friends(ilias, petros) ?creep                       % Κλήση 3
  Call: (2) follows(ilias, petros) ?creep                       % Κλήση 4
  Exit: (2) follows(ilias, petros) ?creep                       % Κλήση 4
  Call: (2) follows(petros, ilias) ?creep                       % Κλήση 5
  Exit: (2) follows(petros, ilias) ?creep                       % Κλήση 5
  Exit: (1) friends(ilias, petros) ?creep                       % Κλήση 3
  Call: (1) male(petros) ?creep                                 % Κλήση 6
  Exit: (1) male(petros) ?creep                                 % Κλήση 6
  Exit: (0) friend_same_gender(ilias, petros) ?creep            % Αρχική κλήση
true                                                            % Κλήση 7
```

Διευκρινίζεται ότι ο πραγματικός αριθμός που εμφανίζεται δίπλα σε κάθε κλήση μπορεί να διαφέρει αν εκτελεστεί το παραπάνω πρόγραμμα σε κάποια υλοποίηση της Prolog, π.χ. στην SWI-Prolog. Η διαφορά οφείλεται σε εσωτερικές διαδικασίες αρχικοποίησης της υλοποίησης. Όπως και να έχει το σχετικό βάθος των κλήσεων μεταξύ τους παραμένει το ίδιο. Επίσης, δίπλα σε κάθε κλήση υπό την μορφή σχολίου εμφανίζουμε και τον αριθμό κλήσης όπως είχε οριστεί και παρουσιαστεί στις προηγούμενες ενότητες. Για το παράδειγμα 1 παρατηρούμε ότι υπάρχει μόνο ένα συμβάν FAIL στην κλήση 1, κάτι που απεικονίζεται με τον κόκκινου χρώματος κόμβο 1 στο Σχήμα 5.1, καθώς και μόνο ένα συμβάν REDO στην αρχική κλήση, κάτι που απεικονίζεται στην διακλάδωση κάτω από τον κόμβο-ρίζα στο Σχήμα 5.1. Όλα τα άλλα συμβάντα είναι CALL ή EXIT.

Στην συνέχεια θα δούμε την παρακολούθηση του παραδείγματος 2 της ενότητας 5.3, το οποίο διαφέρει από το προηγούμενο ως προς το γεγονός ότι έχει ελεύθερη μεταβλητή στην αρχική κλήση και συνεπώς δίνει την δυνατότητα στον χρήστη να ζητήσει εναλλακτική λύση, δηλαδή να ενεργοποιήσει τον μηχανισμό οπισθοδρόμησης.

Όπως φαίνεται παρακάτω, στην παρακολούθηση οι μεταβλητές απεικονίζονται με την εσωτερική τους αναπαράσταση στην Prolog, η οποία συνήθως βασίζεται σε αριθμούς και διαφέρει από υλοποίηση σε υλοποίηση της Prolog. Επίσης, βλέπουμε ότι η παρακολούθηση δεν ακολουθεί πάντα το ιδανικό δένδρο εκτέλεσης που παρουσιάστηκε στην προηγούμενη ενότητα. Για παράδειγμα, μετά την επιστροφή της πρώτης λύσης A=ilias και την ενεργοποίηση του μηχανισμού οπισθοδρόμησης από τον χρήστη με το πάτημα του πλήκτρου ';', η παρακολούθηση επιστρέφει την εκτέλεση στον κλήση 3 (συμβάν REDO), ενώ στο αντίστοιχο δένδρο εκτέλεσης στο Σχήμα 5.2 η επιστροφή γίνεται απευθείας στην κλήση 2, καθώς η κλήση 3 δεν έχει εναλλακτικές λύσεις. Βέβαια, αυτό δεν μπορεί πάντα να το γνωρίζει εκ των προτέρων ο μηχανισμός εκτέλεσης, καθώς δεν χτίζει το δένδρο στατικά αλλά δυναμικά, συνεπώς δεν είναι δυνατόν να γνωρίζει εκ των προτέρων ότι μία κλήση δεν μπορεί να έχει εναλλακτικές λύσεις.

```
[trace] ?- male_friends(petros,A).                              % Αρχική κλήση
  Call: (0) male_friends(petros, _G3640) ?creep                 % Αρχική κλήση
```

```
  Call: (1) friends(petros, _G3640) ?creep                    % Κλήση 1
  Call: (2) follows(petros, _G3640) ?creep                    % Κλήση 2
  Exit: (2) follows(petros, ilias) ?creep                     % Κλήση 2
  Call: (2) follows(ilias, petros) ?creep                     % Κλήση 3
  Exit: (2) follows(ilias, petros) ?creep                     % Κλήση 3
  Exit: (1) friends(petros, ilias) ?creep                     % Κλήση 1
  Call: (1) male(ilias) ?creep                                % Κλήση 4
  Exit: (1) male(ilias) ?creep                                % Κλήση 4
  Exit: (0) male_friends(petros, ilias) ?creep                % Αρχική Κλήση
A = ilias ;                                                   % Κλήση 5
  Redo: (2) follows(ilias, petros) ?creep                     % Κλήση 3
  Fail: (2) follows(ilias, petros) ?creep                     % Κλήση 3
  Redo: (2) follows(petros, _G3640) ?creep                    % Κλήση 2
  Exit: (2) follows(petros, demos) ?creep                     % Κλήση 2
  Call: (2) follows(demos, petros) ?creep                     % Κλήση 6
  Exit: (2) follows(demos, petros) ?creep                     % Κλήση 6
  Exit: (1) friends(petros, demos) ?creep                     % Κλήση 1
  Call: (1) male(demos) ?creep                                % Κλήση 7
  Exit: (1) male(demos) ?creep                                % Κλήση 7
  Exit: (0) male_friends(petros, demos) ?creep                % Αρχική Κλήση
A = demos ;                                                   % Κλήση 8
  Redo: (2) follows(petros, _G3640) ?creep                    % Κλήση 2
  Exit: (2) follows(petros, sofia) ?creep                     % Κλήση 2
  Call: (2) follows(sofia, petros) ?creep                     % Κλήση 9
  Fail: (2) follows(sofia, petros) ?creep                     % Κλήση 9
  Fail: (1) friends(petros, _G3640) ?creep                    % Κλήση 1
  Fail: (0) male_friends(petros, _G3640) ?creep               % Αρχική Κλήση
false
```

Όπως βλέπουμε παραπάνω, η πλήρης παρακολούθηση ενός προγράμματος, ιδιαίτερα αν είμαστε σίγουροι για την "σωστή" εκτέλεση ενός μεγάλου μέρους από αυτό, μπορεί να αποβεί χρονοβόρα διαδικασία. Στην συνέχεια θα παρουσιάσουμε μερικούς τρόπους χειρισμού της παρακολούθησης που οδηγούν σε πιο γρήγορη και στοχευμένη παρακολούθηση προκειμένου να εντοπιστεί γρήγορα κάποιο σφάλμα.

Έστω το παράδειγμα 3 της ενότητας 5.3, στο οποίο δεν μας ενδιαφέρει η αναλυτική παρακολούθηση της "μεσαίας" κλήσης friends(A,B), αλλά απλά μας ενδιαφέρει να δούμε το αποτέλεσμα της εκτέλεσης της συγκεκριμένης κλήσης. Στην περίπτωση αυτή όταν η παρακολούθηση φτάσει στο συμβάν CALL της κλήσης, αντί να πατήσουμε enter ή 'c' (δηλαδή αντί να κάνουμε creep), πατάμε το πλήκτρο 's' και κάνουμε skip (παράλειψη) της βηματικής παρακολούθησης της κλήσης.

```
?- female(A), friends(A,B), male(B).                          % Αρχική κλήση
  Call: (0) female(_G6565) ?creep                             % Αρχική κλήση
  Exit: (0) female(sofia) ?creep                              % Αρχική κλήση
  Call: (0) friends(sofia, _G6568) ?skip                      % Κλήση 1
  Exit: (0) friends(sofia, helen) ?creep                      % Κλήση 1
  Call: (0) male(helen) ?creep                                % Κλήση 4
  Fail: (0) male(helen) ?creep                                % Κλήση 4
  Redo: (0) female(_G6565) ?                                  % Αρχική κλήση
  ...
```

Η παράλειψη φαίνεται από την αναγραφή της ενέργειας skip δίπλα από την αντίστοιχη κλήση, καθώς και από το γεγονός ότι η επόμενη κλήση στην παρακολούθηση είναι η κλήση 4, δηλαδή παραλείπεται η παρακολούθηση των κλήσεων 2 και 3 οι οποίες δημιουργούνται από την ενοποίηση της κλήσης 1 με κανόνα.

Μία άλλη χρήσιμη ενέργεια για την συντόμευση του χρόνου της παρακολούθησης της εκτέλεσης είναι ο ορισμός κάποιων *σημείων κατασκόπευσης* (*spy points*) τα οποία μας ενδιαφέρει να παρακολουθήσουμε λεπτομερώς, ενώ θα θέλαμε να προσπεράσουμε γρήγορα την παρακολούθηση του υπόλοιπου προγράμματος. Ο ορισμός αυτών των σημείων γίνεται με την βοήθεια του ενσωματωμένου κατηγορήματος spy/1, του οποίου το όρισμα είναι το κατηγόρημα που θέλουμε να κατασκοπεύσουμε, στην μορφή predicate/arity. Στο ακόλουθο

παράδειγμα παρακολούθησης (από το πρόγραμμα του παραδείγματος 3 της ενότητας 5.3), έστω ότι μας ενδιαφέρει να παρακολουθήσουμε μόνο την εκτέλεση του κατηγορήματος male/1 της σύνθετης αρχικής κλήσης. Αφού οριστεί το σημείο κατασκόπευσης, στην συνέχεια όταν ενεργοποιηθεί η παρακολούθηση πατάμε το πλήκτρο 'l' (leap, δηλαδή "άλμα"), ώστε να μεταβεί η παρακολούθηση κατευθείαν στο κατηγόρημα male/1, προσπερνώντας όλες τις υπόλοιπες κλήσεις. Δίπλα από τα συμβάντα της κλήσης του male/1 υπάρχει αστερίσκος ο οποίος υποδηλώνει ότι αποτελεί σημείο κατασκόπευσης.

```
?- spy(male/1).
% Spy point on male/1
true

?- female(A), friends(A,B), male(B).                    % Αρχική κλήση
  Call: (8) female(_G2773) ?leap                         % Αρχική κλήση
* Call: (8) male(helen) ?creep                           % Κλήση 4
* Fail: (8) male(helen) ?leap                            % Κλήση 4
* Call: (8) male(sofia) ?creep                           % Κλήση 4
* Fail: (8) male(sofia) ?leap                            % Κλήση 4
false
```

Στο πίνακα που ακολουθεί (Πίνακας 5.3), εμφανίζονται συγκεντρωτικά κάποιες χρήσιμες ενέργειες του αποσφαλματωτητή.

Πίνακας 5.3: *Χρήσιμες Ενέργειες του Αποσφαλματωτή*

Ενέργεια	**Περιγραφή λειτουργίας**
'c'- creep (ερπυσμός) ή πλήκτρο Enter	συνέχιση της βηματικής παρακολούθησης
's' - skip (παράλειψη)	παράλειψη της βηματικής παρακολούθησης της κλήσης
'a' - abort (εγκατάλειψη)	τερματισμός εκτέλεσης του τρέχοντος προγράμματος.
'b' - break (διακοπή)	προσωρινή διακοπή της εκτέλεσης του προγράμματος προκειμένου να μπορέσει να εκτελέσει ερωτήματα ο χρήστης. Επιστροφή στην κανονική εκτέλεση με το πάτημα του συνδυασμού πλήκτρων **CTRL-d**.
'n' - no debug	συνέχιση της εκτέλεσης του προγράμματος μέχρι τέλους χωρίς παρακολούθηση.
'h' - help	εκτύπωση βοήθειας για όλες τις διαθέσιμες εντολές του αποσφαλματωτή.

Στο βίντεο που υπάρχει εδώ, μπορείτε να δείτε την διαδικασία αποσφαλμάτωσης ενός Prolog προγράμματος με την χρήση του αποσφαλματωτή trace της Prolog.

5.6 Προηγμένα Θέματα: Έλεγχος Εμφάνισης

Ο αλγόριθμος ενοποίησης που παρουσιάστηκε στην ενότητα 5.2 δεν λαμβάνει υπόψη του μιαν ακραία αλλά πιθανή περίπτωση: όταν μία μεταβλητή ενοποιείται με με έναν σύνθετο όρο, η μεταβλητή να υπάρχει κάπου μέσα στον σύνθετο όρο. Για παράδειγμα, έστω η ενοποίηση:

```
?- X = f(X).
```

Σύμφωνα με τον αλγόριθμο ενοποίησης που παρουσιάστηκε, η μεταβλητή X ενοποιείται με τον όρο f(X) με την αντικατάσταση ή ανάθεση τιμής {X=f(X)}. Όμως το X μέσα στον όρο f(X) μπορεί επίσης να

αντικατασταθεί με την τιμή του X που υποδεικνύει η προηγούμενη αντικατάσταση, συνεπώς έχουμε μία νέα αντικατάσταση {X=f(f(X))}. Προφανώς αυτές οι αντικαταστάσεις μπορούν να συνεχιστούν επ' άπειρον συνεπώς η διαδικασία της ενοποίησης πέφτει σε ατέρμονα βρόχο.

Συνεπώς, προκειμένου να είναι ορθός ο αλγόριθμος ενοποίησης, θα πρέπει να προστεθεί η παραπάνω περίπτωση ελέγχου, η οποία ονομάζεται **έλεγχος εμφάνισης** (occurs check), στο βήμα 2 του αλγορίθμου, όπως φαίνεται παρακάτω:

2. Μια μεταβλητή ενοποιείται με οποιονδήποτε όρο, εισάγοντας μια νέα αντικατάσταση στο γενικότερο ενοποιητή.
    a. Σε περίπτωση που η μεταβλητή υπάρχει μέσα στον όρο με τον οποίο πάει να ενοποιηθεί, τότε η ενοποίηση αποτυγχάνει (**occurs check**).

Επισημαίνεται ότι το βήμα 2.a δεν υπάρχει σε όλες τις υλοποιήσεις της γλώσσας Prolog, καθώς καθυστερεί σημαντικά τον χρόνο της ενοποίησης, ιδιαίτερα αν αναλογιστούμε ότι ο παραπάνω αλγόριθμος εκτελείται συνεχώς, όπως είδαμε στην ενότητα 5.3. Ως αποτέλεσμα της μη-ύπαρξης του occurs check, το αποτέλεσμα της ενοποίησης είτε δεν είναι λογικά ορθό, είτε καταλήγει σε ατέρμονα βρόχο. Σε πολλές υλοποιήσεις της Prolog υπάρχει τερματισμός της διαδικασίας και επιστροφή μιας απάντησης που παραπέμπει σε "άπειρο" όρο:

```
X=f(f(f(...)))
```

Στην SWI-Prolog υπάρχει τρόπος να ελεγχθεί η εφαρογή ή όχι του μηχανισμού occurs check από τις ρυθμίσεις του περιβάλλοντος του διερμηνέα:

```
?- set_prolog_flag(occurs_check,false).
true
?- X=f(X).
X = f(X).
?- set_prolog_flag(occurs_check,true).
true
?- X=f(X).
false
```

Στην περίπτωση της μη-εφαρμογής του μηχανισμού occurs check πετυχαίνει η ενοποίηση, χωρίς ουσιαστικά να προχωρήσει η ενοποίηση αναδρομικά (άρα δεν πρόκειται για τον γενικότερο ενοποιητή), ενώ στην περίπτωση της εφαρμογής του μηχανισμού occurs check η ενοποίηση αποτυγχάνει.

Βιβλιογραφία

Η σημασιολογία των προγραμμάτων Prolog, ο μηχανισμός ενοποίησης, ο μηχανισμός εκτέλεσης και ο μηχανισμός παρακολούθησης της εκτέλεσης των προγραμμάτων Prolog περιέχονται σχεδόν σε όλα τα "κλασικά" βιβλία αναφοράς για τη γλώσσα προγραμματισμού Prolog, όπως τα (Clocksin and Mellish, 2003), (Bratko, 2011), (Sterling and Shapiro, 1994) και (Covington et al., 1988). Ιδιαίτερα το βιβλίο (Sterling and Shapiro, 1994) δίνει έμφαση στην σημασιολογία των Prolog προγραμμάτων σε σύγκριση και με την λογική πρώτης τάξης, ενώ τα υπόλοιπα δίνουν μεγαλύτερη έμφαση στον μηχανισμό εκτέλεσης της Prolog. Ο μηχανισμός παρακολούθησης της Prolog και το λεγόμενο box-model παρουσιάζονται ιδιαίτερα αναλυτικά στο βιβλίο (Clocksin and Mellish, 2003) και λιγότερο αναλυτικά (έως καθόλου) στα υπόλοιπα.

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Clocksin, W. F. and Mellish C. S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

Covington, M. A. and Nute, D. and Vellino A. (1988). *Prolog Programming in Depth*. Scott Foresman & Co. ISBN 9780673186591.

Sterling, L. and Shapiro, E. Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass: MIT Press.

Ασκήσεις

5.1 Ποιο είναι το αποτέλεσμα της ενοποίησης όρων στις παρακάτω περιπτώσεις και γιατί; Στην περίπτωση που υπάρχουν μεταβλητές να αναφέρετε ποια τιμή τους ανατίθεται.

```
point(A,B) = point(1,2)
point(A,B) = point(X,Y,Z)
plus(2,2) = 4
+(2,D) = +(E,2)
triangle(point(-1,0),P2,P3) = triangle(P1,point(1,0),point(0,Y))
f(X,Y) = f(1,1)
f(X,1) = f(2,Y)
f(X,1) = f(2,X)
f(1,X) = f(Y,Y)
f(1,g(X)) = f(Y,Y)
f(Y,g(X)) = f(X,Y)
```

5.2 Ποιο είναι το αποτέλεσμα της ενοποίησης λιστών στις παρακάτω περιπτώσεις και γιατί; Στην περίπτωση που υπάρχουν μεταβλητές να αναφέρετε ποια τιμή τους ανατίθεται.

```
[X,Y,Z] = [a,b,c,d]
[X,Y] = [a|[b]]
[X|[Y|[Z]]] = [a,b,c]
[X|[Y,c]] = [a,Z,Z]
[X|Y] = [a,b|[Y]]
[X,Y,Z,d] = [a,b,c,d]
[X|Y] = [a,b]
[X|[Y,Z]] = [a,b,c]
[X|[Y,Z]] = [a,Z,c]
[X,Y] = [a,b,Y]
```

Σημείωση: Η άσκηση αυτή συνιστάται να επιλυθεί μετά την μελέτη του Κεφαλαίου 7 στο οποίο εξετάζονται οι λίστες.

5.3 Στο παράδειγμα του κοινωνικού δικτύου της ενότητας 4.5 να αναφέρετε με ποιον τρόπο θα απαντήσει η Prolog τα ακόλουθα ερωτήματα και να δικαιολογήστε την απάντησή σας, με την αναλυτική παρουσίαση της εκτέλεσης των ερωτημάτων με την βοήθεια του αλγορίθμου του μηχανισμού εκτέλεσης της ενότητας 5.3. Εάν υπάρχουν εναλλακτικές απαντήσεις δώστε τουλάχιστον τρεις.

```
?- recommend_common_friends(ilias,demos).
?- recommend_common_friends(ilias,sofia).
?- recommend_common_friends(ilias,X).
?- male(X), female(Y), friends(X,Y).
?- same_group(demos,A).
?- same_group(petros,B), female(B).
?- same_group(X,Y), male(X), female(Y).
?- friend_same_gender(sofia,A).
?- friend_same_gender(A,B).
```

Στη συνέχεια, να σχεδιάσετε τα δένδρα εκτέλεσης / αναζήτησης για κάθε ερώτημα, σύμφωνα με την ενότητα 5.4. Τέλος, παρακολουθήστε την εκτέλεση των παραπάνω ερωτημάτων με την χρήση του κατηγορήματος trace/0 και βεβαιωθείτε ότι καταλαβαίνετε με ποιον τρόπο λειτουργεί ο αποσφαλματωτής της Prolog και εάν η κατανόησή σας σχετικά με τον μηχανισμό εκτέλεσης της Prolog ταιριάζει με τον μοντέλο παρακολούθησης της εκτέλεσης του αποσφαλματωτή.

5.4 Έστω το ακόλουθο λογικό πρόγραμμα:

```
f(1,one).
```

```
f(s(l),two).
f(s(s(l)),three).
f(s(s(s(X))),N) :- f(X,N).
```

Με ποιον τρόπο θα απαντήσει η Prolog τα ακόλουθα ερωτήματα; Δικαιολογήστε την απάντησή σας, με την αναλυτική παρουσίαση της εκτέλεσης των ερωτημάτων με την βοήθεια του αλγορίθμου του μηχανισμού εκτέλεσης της ενότητας 5.3. Εάν υπάρχουν εναλλακτικές απαντήσεις δώστε τουλάχιστον δύο.

```
?- f(s(l),A).
?- f(s(s(l)),two).
?- f(s(s(s(s(s(l))))),C).
?- f(D,three).
```

Στη συνέχεια, να σχεδιάσετε τα δένδρα εκτέλεσης / αναζήτησης για κάθε ερώτημα, σύμφωνα με την ενότητα 5.4. Τέλος, παρακολουθήστε την εκτέλεση των παραπάνω ερωτημάτων με την χρήση του κατηγορήματος trace/0 και βεβαιωθείτε ότι καταλαβαίνετε με ποιον τρόπο λειτουργεί ο αποσφαλματωτής της Prolog και εάν η κατανόησή σας σχετικά με τον μηχανισμό εκτέλεσης της Prolog ταιριάζει με τον μοντέλο παρακολούθησης της εκτέλεσης του αποσφαλματωτή.

Σημείωση: Η άσκηση αυτή συνιστάται να επιλυθεί μετά την μελέτη του Κεφαλαίου 6 στο οποίο εξετάζεται η αναδρομή.

ΚΕΦΑΛΑΙΟ 6: Αναδρομή

Λέξεις Κλειδιά
Αναδρομικές συναρτήσεις, Αναδρομή ως εργαλείο επίλυσης προβλημάτων, Αριθμητικές πράξεις και βασικά αριθμητικά κατηγορήματα, Εκφραστικότητα αναδρομικών ορισμών, Η Αναδρομή ως μεθοδολογία Προγραμματισμού

Περίληψη
Η αναδρομή αποτελεί την κύρια τεχνική ανάπτυξης προγραμμάτων Prolog, καθώς η γλώσσα δεν περιλαμβάνει στην βασική της σύνταξη εντολές επανάληψης ή διακλάδωσης υπό συνθήκη, όπως repeat, for, while κλπ. που υπάρχουν σε άλλες γλώσσες. Το κεφάλαιο ξεκινά με την παρουσίαση των στοιχείων ενός αναδρομικού ορισμού, δίνοντας έμφαση μέσω παραδειγμάτων στην εκφραστική δύναμη των αναδρομικών ορισμών και την λακωνικότητα διατύπωσής τους. Τονίζεται ότι η αναδρομή αποτελεί γενική μεθοδολογία προγραμματισμού, εφαρμόσιμη σε όλες τις γλώσσες. Τέλος, παρουσιάζονται τα κατηγορήματα υπολογισμού αριθμητικών πράξεων σε Prolog, με στόχο να παρατεθούν παραδείγματα υλοποίησης για ενδιαφέροντες αλλά και σχετικά απλούς αναδρομικούς ορισμούς.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός να:

- κατανοήσει την έννοια της αναδρομής ως δομής για την υλοποίηση μιας επαναληπτικής διαδικασίας,
- περιγράφει αναδρομικούς ορισμούς λογικών σχέσεων,
- υλοποιεί ορισμούς κατηγορημάτων βασιζόμενος στην αρχή της αναδρομής,
- χρησιμοποιεί βασικά ενσωματωμένα αριθμητικά κατηγορήματα.

Παράδειγμα κίνητρο

Σε προηγούμενο κεφάλαιο δώσαμε ένα παράδειγμα ενός κοινωνικού δικτύου με σχέσεις φιλίας (friends/2). Η σχέση αυτή μπορεί να θεωρηθεί διασύνδεση (connection) 1ου βαθμού μεταξύ φίλων. Αν θα θέλαμε να επεκτείνουμε τις διασυνδέσεις σε μεγαλύτερο βαθμό, για παράδειγμα 2ου, 3ου και 4ου, θα έπρεπε να γράψουμε τέσσερις διαφορετικούς κανόνες:

```
% 1
connection(X,Y):-
  friends(X,Y).
% 2
connection(X,Y):-
  friends(X,Z),friends(Z,Y).
% 3
connection(X,Y):-
  friends(X,Z),friends(Z,W), friends(W,Y).
% 4
connection(X,Y):-
  friends(X,Z),friends(Z,W),friends(W,U),friends(U,Y).
```

Φυσικά, όσο μεγαλύτερου βαθμού διασύνδεση θέλουμε να ορίσουμε, τόσο και περισσότερους κανόνες Prolog πρέπει να γράψουμε. Είναι ενδιαφέρον όμως να γίνουν οι εξής παρατηρήσεις:

Ο κανόνας 4

```
connection(X,Y):-
  friends(X,Z),friends(Z,W),friends(W,U),friends(U,Y).
```

ισοδυναμεί με:

```
connection(X,Y):-friends(X,Z), connection(Z,Y).
```

λόγω του κανόνα 3. Επιπλέον, ο κανόνας 3

```
connection(X,Y):-friends(X,Z), friends(Z,W), friends(W,Y).
```

ισοδυναμεί με:

```
connection(X,Y):-friends(X,Z), connection(Z,Y).
```

λόγω του κανόνα 2. Τέλος, ο κανόνας 2

```
connection(X,Y):-friends(X,Z), friends(Z,Y).
```

ισοδυναμεί με:

```
connection(X,Y):-friends(X,Z), connection(Z,Y).
```

λόγω του κανόνα 1.

Άρα το πρόγραμμα εύρεσης διασύνδεσης μέχρι 4ου βαθμού γίνεται:

```
%1
connection(X,Y):-
  friends(X,Y).
%2
connection(X,Y):-
  friends(X,Z), connection(Z,Y).
%3
connection(X,Y):-
  friends(X,Z), connection(Z,Y).
%4
connection(X,Y):-
  friends(X,Z), connection(Z,Y).
```

Φυσικά, οι κανόνες 2,3,4 κλπ είναι ίδιοι, άρα το πρόγραμμα συμπτύσσεται στο απλό:

```
connection(X,Y):-friends(X,Y).
connection(X,Y):-friends(X,Z), connection(Z,Y).
```

που μπορεί να εκφραστεί ως εξής:

- υπάρχει διασύνδεση μεταξύ δύο ατόμων X και Y, αν αυτοί είναι φίλοι, αλλιώς
- υπάρχει διασύνδεση μεταξύ δύο ατόμων X και Y, αν ο X είναι απευθείας φίλος με κάποιον Z, και υπάρχει διασύνδεση μεταξύ μεταξύ του Z και του Y.

Οι δύο περιπτώσεις εκφράζουν την άμεση διασύνδεση και την έμμεση διασύνδεση αντίστοιχα. Γίνεται εύκολα κατανοητό ότι ο ορισμός ισχύει για διασύνδεση οποιουδήποτε βαθμού.

Μονοπάτι Μάθησης

Το κεφάλαιο αποτελεί είναι πολύ σημαντικό για τη γνώση της βασικής αρχής σχεδιασμού και υλοποίησης Prolog προγραμμάτων, την αναδρομή. Οι ενότητες 6.1, 6.2 και 6.4 παρουσιάζουν την αρχή της αναδρομής και πρέπει απαραίτητα να περιληφθούν στο βασικό μονοπάτι μάθησης. Η ενότητα 6.7 βοηθάει σε κάποιες παρανοήσεις σχετικά με τους αναδρομικούς ορισμούς, τον τρόπο εκτέλεσης και άλλα χαρακτηριστικά της Prolog. Οι ενότητες 6.5 και 6.6 περιέχουν προηγμένα θέματα που θα μπορούσε κάποιος να διαβάσει σε δεύτερο χρόνο. Τέλος η ενότητα 6.3 ρίχνει μια πρώτη ματιά σε αριθμητικές πράξεις και χρειάζεται ως ενότητα αναφοράς σε ανάπτυξη προγραμμάτων που απαιτούν αριθμητική.

6.1 Αναλυτικοί και Αναδρομικοί Ορισμοί

Στα μαθηματικά, οι ορισμοί συναρτήσεων δίνονται με αναλυτικό τρόπο, εκφράζοντας έτσι τον τρόπο με τον οποίον υπολογίζονται οι τιμές με βάση το πεδίο ορισμού τους. Για παράδειγμα, ο αναλυτικός ορισμός της συνάρτησης N παραγοντικού (συμβολίζεται ως N!) είναι:

- $N! = 1 \times 2 \times 3 \times ... \times (N-1) \times N$, για κάθε $N>0$ ή

- N! = 1, για N=0

Ο ορισμός παραπέμπει απευθείας στον τρόπο υπολογισμού μίας τιμής. Για παράδειγμα το 5! υπολογίζεται ως: N! = 1 × 2 × 3 × 4 × 5 = 120.

Σε μία γλώσσα διαδικαστικού προγραμματισμού με εντολές επανάληψης (for, repeat, while κλπ), μια υλοποίηση της συνάρτησης αυτής θα περιείχε ένα βρόχο με N επαναλήψεις, ο οποίος θα άλλαζε την τιμή μίας μεταβλητής τιμής που αναπαριστά το παραγοντικό με συσσωρευτικό τρόπο. Για παράδειγμα, το πρόγραμμα με εντολή επανάληψης for σε Java περιλαμβάνει:

```
... fact = 1;
... for ( c = 1 ; c <= n ; c++ )
        fact = fact*c;
```

Υπάρχει όμως και ένας άλλος τρόπος να οριστεί αυτή η συνάρτηση, αφού παρατηρήσουμε ότι:

1 × 2 × 3 × ... × (N-1) = (N - 1)!

Ο ορισμός γίνεται:

- N! = (N-1)! × N για κάθε N>0 ή
- N! = 1, για N=0

Ο ορισμός αυτός ονομάζεται αναδρομικός και μπορεί εξίσου καλά να χρησιμοποιηθεί για τον υπολογισμό τιμών της συνάρτησης. Για παράδειγμα το 5! με βάση την αναδρομική συνάρτηση υπολογίζεται ως εξής:

5! = 4! × 5 = (3! × 4) × 5 = ((2! × 3) × 4) × 5 = (((1! × 2) × 3) × 4) × 5 =

= ((((0! × 1) × 2) × 3) × 4) × 5 = 1 × 1 × 2 × 3 × 4 × 5 = 120

Αναδρομικός ορισμός (recursive definition), λοιπόν, είναι η ορισμός που χρησιμοποιεί αυτό που ορίζεται για να οριστεί. Έτσι, για παράδειγμα, ο ορισμός του N! είναι αναδρομικός αφού ορίζεται με βάση το (N-1)!, καθώς επίσης και ο ορισμός του connection/2 αφού το κατηγόρημα εμφανίζεται και στην κεφαλή αλλά και στο σώμα του κανόνα.

Πάντα οι αναδρομικοί ορισμοί έχουν ένα μέρος που δεν είναι αναδρομικό. Στα παραπάνω παραδείγματα το 0!=1 και ο πρώτος κανόνας του connection/2 δεν είναι αναδρομικοί και αποτελούν τις βασικές περιπτώσεις (base cases), περιπτώσεις δηλαδή που η λύση υπολογίζεται άμεσα. Η σπουδαιότητα των βασικών περιπτώσεων θα συζητηθεί αναλυτικά παρακάτω.

Η αναδρομή (recursion) βασίζεται στη μαθηματική επαγωγή και είναι ένα πραγματικά ισχυρό εργαλείο δηλωτικού προγραμματισμού, αφού ένας αναδρομικός ορισμός ορίζει "τι" είναι κάτι παρά το "πώς" υπολογίζεται.

6.2 Η αναδρομή ως μεθοδολογία προγραμματισμού στην Prolog

Σε αντίθεση με τις διαδικαστικές γλώσσες προγραμματισμού, η Prolog δεν έχει άλλον τρόπο (τουλάχιστον όχι στα πλαίσια του κλασικού λογικού προγραμματισμού) να εκφράσει επανάληψη εκτός της αναδρομής. Έτσι η αναδρομή αποτελεί τη βασική μεθοδολογία ανάπτυξης προγραμμάτων Prolog, και αντικαθιστά την ανάγκη χρήσης εντολών επανάληψης.

Στη μεθοδολογία αυτή χρησιμοποιούμε ένα, σχεδόν μαγικό, τέχνασμα: υποθέτουμε ότι έχουμε λύσει μέρος του ίδιου προβλήματος του οποίου ζητάμε τη λύση του. Για παράδειγμα, υποθέτουμε ότι έχουμε λύσει το (N-1)! όταν ζητάμε το N!. Ή ότι έχουμε βρει μία διασύνδεση του φίλου Z κάποιου X με έναν Y όταν ζητάμε τη διασύνδεση αυτού του X με τον Y. Αυτό μας οδηγεί φυσικά στη διατύπωση της αρχής της αναδρομής, που χρησιμοποιείται εκτενώς για την ανάπτυξη Prolog προγραμμάτων. Η αρχή εκφράζεται ως εξής:

- Για να επιλύσω **ένα πρόβλημα μίας τάξης μεγέθους (έστω N)**, υποθέτω ότι έχω επιλύσει **το ίδιο πρόβλημα για κάποια τάξη μεγέθους μικρότερη (έστω N-1)**, και μετά **γεφυρώνω την επιμέρους λύση με τη συνολική λύση** που ψάχνω.

- Επιπλέον πρέπει να έχω πάντα μια τερματική συνθήκη, δηλαδή τη λύση **σε ένα απλό πρόβλημα που γνωρίζω ως βασική περίπτωση**.

Με βάση αυτήν την αρχή, το N! εκφράζεται ως:

- Για να επιλύσω το N!, υποθέτω ότι έχω επιλύσει το (N-1)!, και μετά πολλαπλασιάζω αυτό με το N για να βρω αυτό που ψάχνω.
- Γνωρίζω την τερματική συνθήκη, δηλαδή τη λύση στο 0! που είναι 1.

Η αναζήτηση για διασύνδεση μεταξύ ενός X και ενός Y εκφράζεται ως:

- Για να επιλύσω το connection(X,Y) υποθέτω ότι έχω επιλύσει το connection(Z,Y), όπου ο X με το Z είναι φίλοι friends(X,Z).
- Γνωρίζω την τερματική συνθήκη, δηλαδή ότι υπάρχει connection(X,Y) αν ο X και ο Y είναι απευθείας φίλοι friends(X,Y).

Η υπόθεση του ότι έχω επιλύσει το πρόβλημα για μια μικρότερη εκδοχή του, δημιουργεί την εντύπωση ότι η Prolog με κάποιον μαγικό τρόπο μου επιστρέφει την επί μέρους λύση αυτόματα. Αυτό είναι κάτι πολύ χρήσιμο για κάποιον που μαθαίνει προγραμματισμό, και όσο το χρησιμοποιεί στα αρχικά στάδια τόσο πιο εύκολα θα αναπτύσσει αναδρομικά προγράμματα. Στην πραγματικότητα, το τρικ είναι ότι απλά αναβάλλω την επίλυση του "μεγαλύτερου" προβλήματος, μέχρι να έχω στα χέρια μου την λύση του "μικρότερου" προβλήματος.

6.3 Αριθμητικές Παραστάσεις

Στο σημείο αυτό, θα κάνουμε μία μικρή παρένθεση για να δούμε πώς υπολογίζονται οι αριθμητικές παραστάσεις και πράξεις στην Prolog. Η Prolog ως συμβολική γλώσσα προγραμματισμού δεν υπολογίζει τα ορίσματα των κατηγορημάτων (βλέπετε κεφάλαιο 8). Έτσι, παραδείγματος χάριν, για την Prolog η παράσταση 3+2 είναι ένας σύνθετος όρος με συναρτησιακό σύμβολο το “+” και ορίσματα τα 3 και 2. Θυμίζουμε ότι το σύμβολο = είναι δεσμευμένο για την ενοποίηση δύο όρων και όχι για την ισότητα ή την ανάθεση τιμών που προκύπτουν από τον υπολογισμό αριθμητικών παραστάσεων. Για αυτόν τον σκοπό η Prolog διαθέτει το ενσωματωμένο κατηγόρημα is/2, με τη μορφή ενθεματικού τελεστή (Κεφάλαιο 13). Το is/2 δέχεται δύο ορίσματα, με το δεξιό όρισμα να είναι μια αριθμητική παράσταση και το αριστερό όρισμα να είναι η τελική τιμή της παράστασης αυτής. Οι αριθμητικές παραστάσεις μπορούν να περιέχουν όλους τους γνωστούς αριθμητικούς τελεστές, οι βασικότεροι των οποίων εμφανίζονται στον Πίνακα 6.1. Βασικές αριθμητικές συναρτήσεις εμφανίζονται στο Πίνακα 6.2. Για παράδειγμα:

```
?- X is 3+2.
X=5
?- X is ((3+4)*8) mod 20.
X=16
?- 10 is 20/2.
yes
?- 5 is 11//2.
yes
```

Από τα παραπάνω φαίνεται ότι το κατηγόρημα is/2 χρησιμοποιείται και για τον έλεγχο της τιμής μιας παράστασης, όπως στα δύο τελευταία παραδείγματα, όπου το το όρισμα εξ αριστερών έχει συγκεκριμένη τιμή. Υπάρχει ένας σημαντικός περιορισμός στη χρήση του is/2. Η ύπαρξη αριθμητικών πράξεων επιβάλλει το δεξιό μέλος του κατηγορήματος is/2 να είναι πλήρως ορισμένο (χωρίς ελεύθερες μεταβλητές). Δεν μπορεί δηλαδή το is/2 να λειτουργεί ως επιλυτής εξισώσεων της μορφής 5 is X+2. Το σημαντικό αυτό πρόβλημα, δηλαδή της διαχείρισης των αριθμητικών εξισώσεων σαν λογικών σχέσεων, οδήγησε στην εισαγωγή του λογικού προγραμματισμού με περιορισμούς, που εξετάζεται στο Κεφάλαιο 12.

Πίνακας 6.1: *Αριθμητικοί Τελεστές*

Τελεστής	Ερμηνεία
+. -. *, /	πρόσθεση, αφαίρεση, πολλαπλασιασμός, διαίρεση
//	ακέραια διαίρεση
>, <, >=, =<, \=	συνήθεις τελεστές για σύγκριση τιμών (προσοχή στο >= και στο =<)
^	ύψωση σε δύναμη

Οι τελεστές επιβάλουν να είναι πλήρως δεσμευμένα με τιμές και τα δύο ορίσματά τους.

Πίνακας 6.2: *Συχνά χρησιμοποιούμενες συναρτήσεις*

Συνάρτηση	Ερμηνεία
mod/2	υπόλοιπο διαίρεσης
abs/1	απόλυτη τιμή
sqrt/1	τετραγωνική ρίζα

6.4 Αναδρομικά Κατηγορήματα

Παραγοντικό

Σύμφωνα με το μαθηματικό ορισμό του παραγοντικού, το αντίστοιχο κατηγόρημα ορίζεται με δύο φράσεις:

- μία που δηλώνει την τερματική συνθήκη, δηλαδή τι συμβαίνει για N ίσο με το μηδέν.
- και μία που δηλώνει την εναλλακτική αναδρομική συνθήκη, δηλαδή τι συμβαίνει για N μεγαλύτερο του μηδενός.

Καθώς τα κατηγορήματα είναι σχέσεις και όχι συναρτήσεις, δηλαδή αποτιμώνται σε true ή false και δεν επιστρέφουν τιμές, το κατηγόρημα factorial, το οποίο υλοποιεί την συνάρτηση του παραγοντικού, πρέπει να έχει δύο ορίσματα. Το πρώτο είναι το ο αριθμός του οποίου το παραγοντικό θέλουμε να υπολογίσουμε , δηλαδή το N, ενώ το δεύτερο είναι το αποτέλεσμά της. Η πρώτη φράση είναι ένα γεγονός, το όποιο δηλώνει ότι το παραγοντικό του μηδενός είναι το 1:

```
factorial(0,1).
```

Η δεύτερη φράση εμπεριέχει την αναδρομική κλήση στο ίδιο το κατηγόρημα (υποθέτω ότι έχω λύσει το παραγοντικό του N-1), δηλαδή κάτι σαν:

```
factorial(N,F):-
  ...
  factorial(N-1,F1),
  ...
```

Αλλά επειδή η Prolog δεν μπορεί να υπολογίσει το N-1 ως όρισμα, ο υπολογισμός πρέπει να γίνει "εξωτερικά" της κλήσης του κατηγορήματος factorial/2, μέσω is/2, δηλαδή:

```
factorial(N,F):-
  ...
```

```
  N1 is N-1,
  factorial(N1,F1),
  ...
```

Επιπλέον, η φράση αυτή ισχύει για N>0, άρα θα προστεθεί η αντίστοιχη συνθήκη:

```
factorial(N,F):-
  N>0,
  N1 is N-1,
  factorial(N1,F1),
  ...
```

Τέλος, το παραγοντικό του N-1, δηλαδή το F1, πρέπει να πολλαπλασιαστεί με το N για να δώσει το τελικό παραγοντικό F. Άρα, ο πλήρης ορισμός του κατηγορήματος είναι:

```
factorial(0,1).
factorial(N,F):-
  N>0,
  N1 is N-1,
  factorial(N1,F1),
  F is F1*N.
```

Για παράδειγμα, κάποιες ερωτήσεις που δείχνουν την εκτέλεση του κατηγορήματος είναι οι ακόλουθες:

```
?-factorial(5,X).
X=120

?-factorial(4,30).
no
```

Μπορείτε να δείτε εδώ ένα video για το πως ορίζεται το κατηγόρημα factorial/2.

Διαίρεση Ακεραίων

Με βάση την αρχή επίλυσης προβλημάτων με αναδρομή, η ακριβής διαίρεση δύο ακεραίων αριθμών εκφράζεται ως:

- Για να βρω αν ένας αριθμός X διαιρεί έναν άλλον Y υποθέτω ότι ο X διαιρεί το Y-X.
- Γνωρίζω την τερματική συνθήκη**,** δηλαδή ότι ένας αριθμός διαιρεί τον εαυτό του.

Το πρόγραμμα που ανταποκρίνεται στον παραπάνω ορισμό είναι:

```
divides(X,X).
divides(X,Y):-
  X>0, Y>0,
  D is Y-X,
  divides(X,D).
```

Αν θα θέλαμε να βρούμε και πόσες φορές ο X διαιρεί το Y, δηλαδή το πηλίκο της ακέραιας διαίρεσης, η αρχή της αναδρομής θα εκφραζόταν ως εξής:

- Για να βρω πόσες φορές ένας αριθμός X διαιρεί έναν άλλον Y υποθέτω ότι έχω βρει πόσες φορές ο X διαιρεί το Y-X, και προσθέτω 1.
- Γνωρίζω την τερματική συνθήκη**,** δηλαδή ότι ένας αριθμός διαιρεί τον εαυτό του μία φορά.

Ο ορισμός σε Prolog είναι:

```
divides(X,X,1).
divides(X,Y,N):-
  X>0, Y>0,
  D is Y-X,
  divides(X,D,N1),
  N is N+1.
```

Για παράδειγμα, κάποιες ερωτήσεις είναι:

```
?-divides(5,30,D).
```

```
D=6
?-divides(4,10,D).
no
?-divides(2,-8,D).
no
```

Μπορείτε να δείτε εδώ ένα video για το πως ορίζεται το κατηγόρημα divides/2.

Συμβολική Παραγώγιση

Οι παρακάτω είναι κάποιοι από τους κανόνες παραγώγισης μιας παράστασης ως προς x:

dx / dx = 1

dc / dx = 0, όταν το c είναι μια σταθερά ή παράσταση που δεν περιέχει το x,

dsin(x) / dx = cos(x)

dcos(x) / dx = -sin(x)

d(u + v) / dx = d(u)/χ + d(v) / x

d(u × v) / dx = (d(u) / d) × v + u × d(v) / x

Οι παραπάνω κανόνες αντιστοιχούνται ένας προς έναν με τις εξής Prolog φράσεις:

```
derivative(X, X, 1).       % dx/dx=1

derivative(C, X, 0) :-     % dc/dx=0 if c*x
  atomic(C), X\=C.

derivative(sin(X), X, cos(X)).    % dsin(x)/dx=cos(x)

derivative(cos(X), X, -sin(X)).   % dcos(x)/dx=-sin(x)

derivative(U+V, X, DU+DV) :-      % d(u+v)/dx=du/dx+dv/dx
  derivative(U, X, DU),
  derivative(V, X, DV).

derivative(U*V, X, V*DU+U*DV) :- % d(u*v)/dx=v*du/dx+u*dv/dx
  derivative(U, X, DU),
  derivative(V, X, DV).
```

Οι ερωτήσεις περιλαμβάνουν ως πρώτο όρισμα την παράσταση προς παραγώγιση ενώ δεύτερο τον όρο (δηλαδή την αλγεβρική μεταβλητή) ως προς τον οποίο θα γίνει η παραγώγιση. Για παράδειγμα η παραγώγιση:

d(α × α + 3 × α + 5)/dα

εκφράζεται ως ερώτηση:

```
?- derivative(a*a+3*a+5, a, D).
D = a*1 + a*1 + a*0 + 3*1 + 0
```

το οποίο είναι ισοδύναμο με 2*a+3.

6.5 Προχωρημένα Θέματα

Πρόσθεση χωρίς +

Στα μαθηματικά, η πρόσθεση δύο θετικών ακεραίων αριθμών με βάση τον επόμενο (succ) ή προηγούμενο (pred) ενός ακεραίου αριθμού ορίζεται αναδρομικά ως:

- x + 0 = x, δηλαδή η πρόσθεση ενός ακεραίου με το μηδέν έχει σαν αποτέλεσμα τον ίδιο τον αριθμό.

- x+y =succ(x + pred(y)), δηλαδή η πρόσθεση ενός X>0 και ενός Y>0 έχει σαν αποτέλεσμα τον επόμενο της πρόσθεσης του X με τον προηγούμενο του Y.

Η παραπάνω αναπαράσταση των φυσικών αριθμών ονομάζεται αριθμητική Peano. Η SWI-Prolog και ίσως σε άλλες υλοποιήσεις της Prolog υπάρχει ένα ενσωματωμένο κατηγόρημα succ/2 που επιστέφει τον επόμενο ή προηγούμενο αριθμός ενός ακεραίου ανάλογα με το πώς θα κληθεί. Για παράδειγμα:

```
?-succ(4,X).
X=5
?-succ(X,4).
X=3
```

Ο αναδρομικός ορισμός του κατηγορήματος addition/3 που κάνει πρόσθεση χωρίς τη χρήση του τελεστή +/2, είναι:

```
addition(0,N,N).
addition(X,Y,R):-
  X>0, Y>0,
  succ(Previous,X),
  addition(Previous,Y,PreviousPlusY),
  succ(PreviousPlusY,R).
```

6.6 Σημαντικές Παρατηρήσεις

Όπως αναφέρθηκε σε προηγούμενο κεφάλαιο η μεταβλητή στον λογικό προγραμματισμό έχει διαφορετική σημασία από αυτή στον διαδικαστικό προγραμματισμό όπου δηλώνει μία θέση μνήμης η οποία αποθηκεύει μία τιμή που μπορεί να αλλάξει. Έτσι για παράδειγμα στις διαδικαστικές γλώσσες είναι σύνηθες να γράφεται i=i+1, εννοώντας στην πράξη "πάρε την τιμή που βρίσκεται στη θέση μνήμης i, πρόσθεσε 1 και αποθήκευσε τη νέα τιμή στην ίδια θέση μνήμης.

Στην Prolog, μια μεταβλητή αναπαριστά μία και μόνον τιμή. Έτσι εκφράσεις της μορφής X is X+1 δεν είναι "λογικά" σωστές, αφού υποδηλώνουν ότι η μεταβλητή X μπορεί να έχει δύο τιμές ταυτόχρονα, για παράδειγμα 4 και 5 ή αν διαβάσουμε την παραπάνω έκφραση σαν λογική σχέση, είναι προφανές ότι δεν μπορεί να υπάρχει αριθμητική τιμή ή οποία ίση με τον εαυτό της αυξημένο κατά ένα. Κατά την εκτέλεση ενός προγράμματος, μία μεταβλητή μπορεί να είναι μη δεσμευμένη (να μην έχει τιμή) ή δεσμευμένη (να έχει μόνο μία τιμή). Η μετάβαση από μη δεσμευμένη σε δεσμευμένη γίνεται με ενοποίηση ή αριθμητική ανάθεση στην ειδική περίπτωση των αριθμητικών εκφράσεων. Αντίστροφα η μετάβαση από δεσμευμένη σε μη δεσμευμένη γίνεται κατά την οπισθοδρόμηση.

Η αναδρομή έχει τις ρίζες της στην μαθηματική επαγωγή, καθώς ένας αναδρομικός ορισμός μπορεί να αποδειχθεί με τη μέθοδο της επαγωγής. Αυτή η μέθοδος απόδειξης βασίζεται στο ότι, για παράδειγμα, ένας μαθηματικός τύπος Φ(n) αποδεικνύεται για n=k+1 στοιχεία, εφόσον έχουμε υποθέσει ότι ισχύει για n=k στοιχεία. Φυσικά υπάρχει και η βασική στοιχειώδης απόδειξη για n=1. Για παράδειγμα εικάζουμε ότι ο τύπος Φ(n) = 1+2+3+...+n έχει σαν αποτέλεσμα το $n \times (n+1) / 2$. Στην επαγωγή υπάρχουν τρία βήματα:

- Βήμα 1: Ισχύει για την περίπτωση n = 1, δηλαδή:

  $\Phi(1) = 1 = 1 \times (1 + 1) / 2$

- Βήμα 2: Έστω ότι ισχύει για k:

  $\Phi(k) = 1 + 2 + 3 + ... + k = k \times (k + 1) / 2$

- Βήμα 3:Αποδεικνύεται ότι ισχύει για k+1, δηλαδή:

  $\Phi(k + 1) = 1 + 2 + 3 + ... + k + (k + 1) = (k + 1) \times ((k + 1) + 1) / 2$

με αντικατάσταση από το Φ(k). Γίνεται φανερό ότι το βήμα 2 είναι παρόμοιο με την αναδρομική κλήση και το βήμα 1 είναι παρόμοιο με την τερματική συνθήκη.

6.7 Κοινές Παρανοήσεις

Ο αρχάριος προγραμματιστής της Prolog μπορεί να έχει κάποιες απορίες για ορισμένα σημεία των προγραμμάτων που παρουσιάστηκαν στο κεφάλαιο αυτό. Παίρνοντας ως βάση το παράδειγμα του παραγοντικού (factorial/2) απαντάμε σε κάποιες από τις συχνές ερωτήσεις:

Γιατί το κατηγόρημα factorial έχει δύο ορίσματα;

Είναι συχνό λάθος για αυτούς που είναι πιο πολύ εξοικειωμένοι με το συναρτησιακό προγραμματισμό ή τον κλασικό προστακτικό προγραμματισμό, όπου η κλήση μιας συνάρτησης (function) αντικαθίσταται με την τιμή της. Στο λογικό προγραμματισμό, τα κατηγορήματα δεν έχουν άλλη τιμή εκτός της λογικής τιμής της αληθείας ή της αποτυχίας. Οποιαδήποτε άλλη τιμή χρειάζεται να επιστραφεί, για παράδειγμα το αποτέλεσμα του N!, αυτό γίνεται μόνον μέσω ορισμάτων.

Γιατί η τερματική συνθήκη αναγράφεται πρώτη;

Στη Λογική η σειρά των εναλλακτικών περιπτώσεων που ορίζουν ένα κατηγόρημα δεν έχει σημασία γιατί οι περιπτώσεις υποδηλώνουν το λογικό Η'. Στην Prolog όμως, που υπάρχει ένας συγκεκριμένος τρόπος εκτέλεσης προγραμμάτων (βλ. προηγούμενα κεφάλαια), η σειρά παίζει ρόλο και μπορεί να επηρεάσει το αποτέλεσμα, αν η τερματική συνθήκη και ο αναδρομικός κανόνας δεν είναι αμοιβαία αποκλειόμενα Στην περίπτωση αυτή είναι αμοιβαία αποκλειόμενα (λόγω το N>0), και δε θα έπαιζε κανένα ρόλο. Συνίσταται όμως η τερματική συνθήκη ή συνθήκες να αναγράφονται πρώτες.

Γιατί υπάρχει το N>0 και τι θα γινόταν χωρίς αυτό;

Κάποιος θα μπορούσε να πει ότι εφόσον η τερματική συνθήκη ορίζει το 0!, η προϋπόθεση N>0 που αναγράφεται στο δεύτερο κανόνα είναι περιττή. Αν παραληφθεί, τότε το πρώτο αποτέλεσμα που θα δώσει η Prolog είναι σωστό. Όμως σε περίπτωση που ζητηθεί και άλλο αποτέλεσμα, η εκτέλεση του προγράμματος εκτελεί έναν ατέρμονο βρόχο έως ότου εξαντληθεί η μνήμη. Και τούτο γιατί μειώνεται η τιμή του παραγοντικού που ψάχνουμε σε αρνητικούς αριθμούς χωρίς η μείωση να τελειώνει ποτέ. Επίσης, αν παραληφθεί το N>0, τότε οι δύο φράσεις δεν είναι αμοιβαία αποκλειόμενες και συνεπώς αναγραφή της τερματικής συνθήκης σαν δεύτερη φράση δε θα οδηγούσε ούτε σε σωστό πρώτο αποτέλεσμα.

Μπορεί η τερματική συνθήκη να γραφτεί ως κανόνας;

Όντως, θα μπορούσε η τερματική συνθήκη να γραφεί ως κανόνας:

```
factorial(N,F):-
  N=0,
  F=1.
```

Κάποιοι στα αρχικά στάδια της εκμάθησης της Prolog το κάνουν συχνά, και δεν είναι γενικά λάθος. Όμως δεν εκμεταλλεύονται πλήρως το μηχανισμό ενοποίησης της γλώσσας με την κεφαλή μίας φράσης, με αποτέλεσμα να εκτελούνται δύο περιττές κλήσεις σε ταυτοποίηση μέσω του τελεστή =/2.

Γιατί υπάρχει το F1, και δεν αρκεί μόνο το F;

Θυμίζουμε ότι μια μεταβλητή μπορεί να πάρει μόνο μία τιμή. Συνεπώς μια παραλλαγή της μορφής:

```
factorial(N,F):-
  N>0,
  N1 is N-1,
  factorial(N1,F),
  F is F*N.
```

είναι λάθος γιατί το F δε μπορεί να είναι ταυτόχρονα το N! και το (N-1)!. Η περίπτωση ανάγεται στο ίδιο λάθος που αναφέρθηκε για το N is N-1.

Γιατί το N1 is N-1 αναγράφεται πριν την αναδρομική κλήση;

Στη Μαθηματική Λογική η σειρά των κατηγορημάτων στο σώμα ενός κανόνα που ορίζει ένα κατηγόρημα δεν έχει σημασία γιατί υποδηλώνει το λογικό ΚΑΙ. Στην Prolog όμως, η κλήση μέσα σε μία σύζευξη γίνεται με

μία σειρά, και όπως ειπώθηκε σε προηγούμενο κεφάλαιο, αυτή είναι από αριστερά προς τα δεξιά. Άρα, όταν κληθεί το factorial(N1,F1), το N1 πρέπει να έχει τιμή. Ο λόγος είναι ότι αν το N1 δεν έχει τιμή την στιγμή που θα κληθεί το παραπάνω, τότε στην επόμενη ενοποίηση του factorial(N1,F1) με την κεφαλή του δεύτερου κανόνα, στο σώμα του κανόνα θα εκτελεστεί ο έλεγχος N>0, ο οποίος όμως λόγω της μη-ύπαρξης ανατεθειμένης αριθμητικής τιμής στην μεταβλητή N, όχι απλά θα αποτύχει αλλά θα προκαλέσει τερματισμό της εκτέλεσης του προγράμματος σε μήνυμα λάθους.

*Γιατί το F is F1*N αναγράφεται μετά την αναδρομική κλήση;*

Κάτι παρόμοιο συμβαίνει και με την τελευταία κλήση F is F1*N. Λόγω του is/2, όλες οι μεταβλητές της παράστασης στα δεξιά του πρέπει να είναι πλήρως δεσμευμένες, άρα το F1 πρέπει να πάρει τιμή από την αναδρομική κλήση πριν γίνει ο πολλαπλασιασμός.

*Τί θα γινόταν αν αντί F is F1*N αναγραφόταν F=F1*N;*

Αυτό είναι ενδιαφέρον και αποδεικνύει τη δυνατότητα της Prolog να λειτουργεί συμβολικά. Αν στη θέση του F1*N αναγραφόταν F=F1*N, στην ουσία δε ζητάμε να γίνει ο πολλαπλασιασμός αλλά να δημιουργηθεί ο όρος που περιέχει το συναρτησιακό σύμβολο */2. Η απάντηση της Prolog σε μία ενδεχόμενη ερώτηση θα ήταν:

```
?-factorial(5,F).
F=1*1*2*3*4*5
```

Σε μία τέτοια περίπτωση, ενδιαφέρουσα είναι η ερώτηση:

```
?-factorial(5,X), F is X.
X=1*1*2*3*4*5
F=120
```

Μπορεί το κατηγόρημα factorial/2 να λειτουργήσει "ανάποδα", δηλαδή να δοθεί η τελική τιμή του παραγοντικού και να ζητηθεί ο αρχικός αριθμός;

Λογικά ναι, αλλά πρακτικά όχι!! Ο μηχανισμός εκτέλεσης και τα μη-λογικά χαρακτηριστικά κάποιων κατηγορημάτων, όπως το is/2 και το >/2, επιβάλουν οι κλήσεις να γίνουν με συγκεκριμένη σειρά και κάποιες μεταβλητές να έχουν τιμές (να μην είναι ελεύθερες) όταν αυτά κληθούν. Δυστυχώς, αν κάποιος θέλει να λειτουργήσει το factorial/2 "ανάποδα", πρέπει να γράψει περισσότερο κώδικα, στον οποίο να διαχωρίζει τις περιπτώσεις που το N είναι δεσμευμένο με τιμή ή ελεύθερο. Το ίδιο ισχύει και για το divides/3, αν κάποιος θα ήθελε να βρει τους διαιρέτες ενός αριθμού.

Βιβλιογραφία

Το θέμα της αναδρομής σαν βασικό εργαλείο ανάπτυξης προγραμμάτων Prolog αναφέρεται σε όλα τα "κλασικά" βιβλία αναφοράς για τη γλώσσα προγραμματισμού, όπως τα (Clocksin and Mellish, 2003), (Bratko, 2011), (Sterling and Shapiro, 1994) (O'Keefe, 1990) και (Covington et al., 1988).

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Clocksin, W. F. and Mellish C. S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

O'Keefe, R. A. (1990), *The craft of Prolog*, Cambridge, Mass: MIT Press

Sterling, L. and Shapiro, E. Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass: MIT Press.

Άλυτες Ασκήσεις

6.1: Να γραφεί ο ορισμός ενός κατηγορήματος connection_degree/3, σαν επέκταση του connection/2, το οποίο να επιστέφει και το βαθμό διασύνδεσης, δηλαδή 1ο, 2ος, 3ος κλπ.

6.2: Να γραφεί ο ορισμός ενός κατηγορήματος fibonacci/2 το οποίο να υλοποιεί την αναδρομική συνάρτηση Fibonacci f:

- $f_n = 1$, εάν n = 1,
- $f_n = 1$, εάν n = 1,
- $f_n = f_{n+1} + f_{n+2}$, εάν n > 2.

Για παράδειγμα, η ακολουθία Fibonacci μέχρι το ενδέκατο στοιχείο είναι:

f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}
1	1	2	3	5	8	13	21	34	55	89

6.3: Ο αλγόριθμος του Ευκλείδη για την εύρεση του Μέγιστου Κοινού Διαιρέτη (ΜΚΔ) δύο θετικών ακεραίων είναι :

- μκδ(m,n) = n, εάν m=0,
- μκδ(m,n) = μκδ(n,m), εάν m > n,
- μκδ(m,n) = μκδ(m,mod(n,m)), εάν n =<m

όπου mod(n, m) είναι το υπόλοιπο της διαίρεσης n/m. Στην Prolog, αυτό μπορεί να εκφραστεί X is N mod M. Ακολουθώντας τον αλγόριθμο βήμα βήμα, ο ΜΚΔ του 32 και του 122 βρίσκεται ως εξής:

```
ΜΚΔ(32, 122) = ΜΚΔ(32, mod(122, 32)) = ΜΚΔ(32, 26) = ΜΚΔ(26, 32)
= ΜΚΔ(26, mod(32, 26)) = ΜΚΔ(26, 6) = ΜΚΔ(6, 26)
= ΜΚΔ(6, mod(26, 6)) = ΜΚΔ(6, 2) = ΜΚΔ(2, 6) = ΜΚΔ(2, mod(6, 2))
= ΜΚΔ(2, 0) =  ΜΚΔ(0, 2) = 2
```

Να γραφεί ο ορισμός ενός κατηγορήματος mkd/3 το οποίο να υλοποιεί αλγόριθμο του Ευκλείδη για το ΜΚΔ.

6.4: Να γραφεί ο ορισμός ενός κατηγορήματος nat/1 το οποίο να αληθεύει αν ένας δεδομένος αριθμός είναι φυσικός. Να χρησιμοποιηθεί το ενσωματωμένο κατηγόρημα succ/2.

6.5: Να γραφεί ο ορισμός ενός κατηγορήματος sumn/2 το οποίο, δεδομένου ενός φυσικού αριθμού N, βρίσκει το άθροισμα όλων των αριθμών από το 1 έως το N. Για παράδειγμα:

```
?- sumn(4, X).
X=10
```

6.6 Να συμπληρωθεί ο ορισμός της παραγώγισης, ο οποίος δόθηκε παραπάνω,προσθέτοντας και άλλους κανόνες όπως:

$dx^n/dx = n\,x^{(n-1)}$

$de^x/dx = e^x$

$dln(x)/dx = 1/x$

$d(-f(x))/dx = -\,df(x)/dx$

$d(1/f(x))/dx = -\,(df(x)/dx)/f(x)^2$

6.7 Να εξηγήσετε τη λειτουργία του παρακάτω Prolog προγράμματος:

```
simplify(A + 0, A1) :-
  simplify(A, A1).

simplify(0 + B, B1) :-
  simplify(B, B1).

simplify(A + B, E) :-
  simplify(A, A1),
  simplify(B, B1),
  (A1 \= A; B1 \= B),
  simplify(A1 + B1, E).
```

Να ορίσετε την ίδια λειτουργία για τις τρεις άλλες αριθμητικές πράξεις. Αφού προσθέσετε τις φράσεις:

```
simplify(A^0, 1).

simplify(A^1, A1) :-
  simplify(A, A1).

simplify(0^E, 0).

simplify(1^E, 1).

simplify(A^B, E) :-
  simplify(A, A1),
  simplify(B, B1),
  (A1 \= A; B1 \= B),
  simplify(A1^B1, E).

simplify(-(-(A)), A1) :-
  simplify(A, A1).

simplify(A, A).
```

να δοκιμάσετε το πρόγραμμα με κατάλληλες ερωτήσεις.

6.8 Να γραφεί ο ορισμός ενός κατηγορήματος power/3, το οποίο δεδομένου ενός αριθμού και μιας δύναμης στην οποία θέλουμε να υψώσουμε τον αριθμό αυτό, να επιστρέφει το αποτέλεσμα. Για παράδειγμα:

```
?- power(3,2,X).
X=9
?-power(2,6,X).
X=64
```

Ο ορισμός πρέπει να καλύπτει και αρνητικές δυνάμεις. Δεν πρέπει να χρησιμοποιηθεί το σύμβολο ^.

ΚΕΦΑΛΑΙΟ 7: Άπειροι Όροι σε μια μεταβλητή: Λίστες της Prolog

Λέξεις Κλειδιά
Δομή και Ενοποίηση λιστών, Αναδρομή και Διαχείριση Λιστών, Βασικά (κλασσικά) κατηγορήματα Λιστών

Περίληψη
Το κεφάλαιο αυτό πραγματεύεται την πλέον χρησιμοποιούμενη δομή δεδομένων της Prolog, την λίστα όρων (ή απλά λίστα). Οι λίστες έχουν μεγάλη εφαρμογή στον λογικό προγραμματισμό καθώς είναι ίσως η μια και μοναδική "δομή δεδομένων" συλλογών που υποστηρίζεται στην Prolog. Όπως θα δούμε στην συνέχεια, οι λίστες αποτελούν απλώς μια κατηγορία αναδρομικών όρων με ειδική σύνταξη. Ενδιαφέρον παρουσιάζει το πως ο μηχανισμός ενοποίησης εφαρμόζεται στις λίστες, εκμεταλλευόμενος την ειδική σύνταξη που εισήχθηκε για αυτές. Δίνονται παραδείγματα αναδρομικών ορισμών κατηγορημάτων με χρήση λιστών, και τέλος, παρουσιάζονται τα κλασσικά κατηγορήματα διαχείρισης λιστών (member/2, append/3, delete/4, κλπ), και οι υλοποιήσεις τους σαν αναδρομικοί ορισμοί.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός:

- Να κατανοεί ότι οι λίστες είναι ειδική σημειογραφία όρων της Prolog.
- Να αναγνωρίσει και να αναπαραστήσει δεδομένα μέσω της σύνταξης λιστών.
- Να κατανοεί σε βάθος τον μηχανισμό ενοποίησης λιστών και πως ο τελευταίος παράγει δεσμεύσεις μεταβλητών.
- Να επιλύει προβλήματα συνδυάζοντας αναδρομή και λίστες.
- Να γνωρίζει τα ενσωματωμένα κατηγορήματα της Prolog για διαχείριση λιστών και να μπορεί να τα χρησιμοποιεί με διαφορετικούς τρόπους για την ανάπτυξη προγραμμάτων.
- Να εμπεδώσει την ανυπαρξία χαρακτηρισμού των ορισμάτων ενός κατηγορήματος σαν εισόδου εξόδου.

Παράδειγμα κίνητρο

Αν και η αναπαράσταση οντοτήτων με την χρήση όρων είναι ιδιαίτερα ισχυρή, σε πολλές περιπτώσεις είναι αναγκαία η χρήση όρων με μεταβλητό αριθμό ορισμάτων, δηλαδή μιας δομής που να περιέχει μεταβλητό αριθμό στοιχείων. Για παράδειγμα, έστω ότι θα θέλαμε να γράψουμε ένα πρόγραμμα το οποίο εκτελεί την πράξη της λογικής σύζευξης (bitwise_and) ανάμεσα σε δύο δυαδικούς αριθμούς οποιουδήποτε μεγέθους. Η βασική περίπτωση μονοψήφιων αριθμών (1-bit) εκφράζεται απλά:

```
bitwise_and(1,1,1).
bitwise_and(1,0,0).
bitwise_and(0,1,0).
bitwise_and(0,0,0).
```

Όμως, τι γίνεται με αριθμούς 2-bit; Η προσέγγιση μπορεί να γίνει με δύο τρόπους: είτε εκφράζοντας απευθείας την πράξη για αριθμούς 2 -bit:

```
bitwise_and(11,11,11).
bitwise_and(10,11,10).
...
```

ή (ακόμη χειρότερα) αυξάνοντας τον αριθμό των ορισμάτων του κατηγορήματος:

```
bitwise_and(1,1,1,1,1,1).
bitwise_and(1,0,1,1,1,0).
```

. . .

Το πρόβλημα της αναπαράστασης εντείνεται και από το γεγονός ότι ο αριθμός των γεγονότων που θα πρέπει να υλοποιηθούν γίνεται εξαιρετικά μεγάλος, για παράδειγμα για δύο αριθμούς με n ψηφία (n-bit) χρειαζόμαστε 2^{2N} γεγονότα.

Προφανώς, το πρόβλημα δεν επιλύεται αν εκφραστούν οι δυαδικοί αριθμοί σαν όροι με τη χρήση κάποιου συναρτησιακού συμβόλου, πχ. bn (binary number), όπου αν και με αυτό τον τρόπο διατηρώ τον αριθμό των ορισμάτων του κατηγορήματος, μεταθέτω ουσιαστικά το πρόβλημα, όπως φαίνεται στο ακόλουθο παράδειγμα:

```
bitwise_and(bn(1),bn(1),bn(1)).
bitwise_and(bn(1,1),bn(1,1),bn(1,1)).
bitwise_and(bn(1,1,1),bn(1,1,1),bn(1,1,1)).
```

Αυτό που θα έλυνε πραγματικά το πρόβλημα είναι να μπορώ να εκφράσω τον κάθε n-bit αριθμό σαν ένα όρισμα του κατηγορήματος, το οποίο όρισμα περιέχει ένα μεταβλητό και δυνητικά άπειρο (απεριόριστο) αριθμό όρων (0 και 1 στην περίπτωση που εξετάζεται). Αυτό ακριβώς επιτυγχάνεται με τις λίστες, χρησιμοποιώντας μια πολύ συμπαγή σημειογραφία, όπως φαίνεται παρακάτω (και εξηγείται στην συνέχεια):

```
bitwise_and([1,1],[1,1],[1,1]).
```

Μονοπάτι μάθησης

Τα βασικά στοιχεία για το κεφάλαιο αυτό βρίσκονται στις ενότητες 7.1 έως και 7.6 οι οποίες πρέπει να ολοκληρωθούν από τον αναγνώστη. Οι ενότητες περιλαμβάνουν βασικές έννοιες για τις λίστες στην Prolog, την εφαρμογή της μεθόδου της αναδρομής στις λίστες, την παρουσίαση των ενσωματωμένων κατηγορημάτων, καθώς και δύο σημαντικά παραδείγματα εφαρμογής. Η ενότητα 7.7 περιέχει επιπρόσθετα στοιχεία που δεν είναι όμως απαραίτητα για τη συνέχεια. Τέλος, η ενότητα 7.8 περιέχει σημαντικές παρατηρήσεις πάνω στις λίστες και θα πρέπει να περιληφθεί σε μια πρώτη ανάγνωση.

7.1 Οι Λίστες σαν Σύνθετοι Όροι

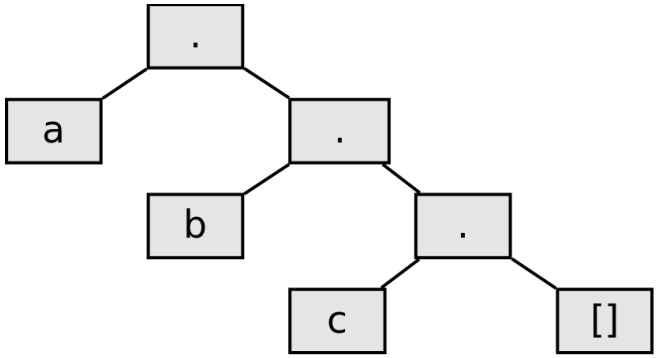
Οι λίστες ανήκουν στην κατηγορία των αναδρομικών σύνθετων όρων, δηλαδή όρων που ορίζονται περιγράφοντας την δομή τους με όρους της ιδίας μορφής. Ο αναδρομικός ορισμός της λίστας είναι:

- **Βασική Περίπτωση**: Μια λίστα δεν περιέχει κανένα στοιχείο και συμβολίζεται με []. Μια τέτοια λίστα ονομάζεται άδεια ή κενή λίστα.
- **Αναδρομική Περίπτωση**: Μια λίστα είναι ένας όρος δύο ορισμάτων, με συναρτησιακό σύμβολο την τελεία (“.”), με το πρώτο όρισμα να είναι ένας οποιοσδήποτε όρος και το δεύτερο μια λίστα. (αναδρομική περίπτωση)

Για παράδειγμα τα ακόλουθα είναι λίστες:

1. [] (κενή λίστα)
2. .(a,[]) (λίστα με μοναδικό στοιχείο το a)
3. .(a, .(b, [])) (λίστα με στοιχεία τα a και b)
4. .(a, .(b, .(c, []))) (λίστα με στοιχεία τα a,b και c)

Η δενδρική αναπαράσταση του τελευταίου παραδείγματος των τριών στοιχείων φαίνεται στο Σχήμα 7.1



Σχήμα 7.1: *Λίστα τριών στοιχείων ως δένδρο*

Δεδομένης της ευρύτατης χρήσης των λιστών, η Prolog προσφέρει μια περισσότερο συμπαγή σημειογραφία, αντικαθιστώντας την δύσχρηστη μορφή .(a, .(b, .(c, []))) με τα σύμβολα "[" και "]", που περικλείουν τα στοιχεία της λίστας, διαχωριζόμενα μεταξύ τους με κόμμα (",").

Έτσι, η λίστα των τριών στοιχείων γράφεται απλά σαν [a,b,c]. Στο παράδειγμα των δυαδικών αριθμών, οι λίστες μπορούν να αναπαραστήσουν αριθμούς οσωνδήποτε bits:

1. [0] δυαδικός αριθμός 1-bit
2. [0,1] δυαδικός αριθμός 2-bit
3. [1,0,1] δυαδικός αριθμός 3-bit
4. [1,0,0,1,0,0] δυαδικός αριθμός 6-bit

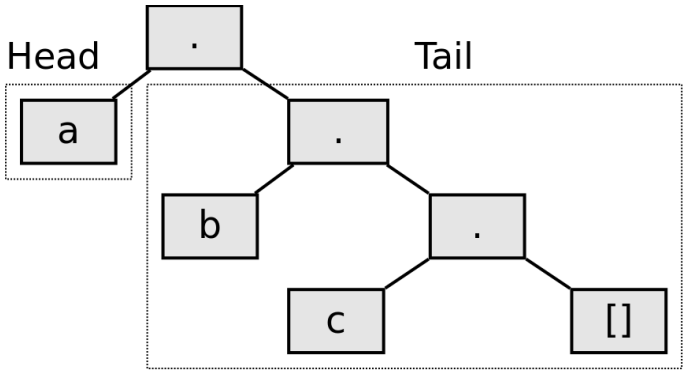
Σημαντικότατες έννοιες στον χειρισμό των λιστών είναι εκείνης της κεφαλής και της ουράς της λίστας.

Κεφαλή και Ουρά

Τα δύο ορίσματα του όρου της λίστας αναφέρονται ως κεφαλή (head) και ουρά (tail) (Σχήμα 7.2).

1. Η **κεφαλή** (head) είναι το πρώτο (first) στοιχείο της λίστας.
2. Η **ουρά** (tail) περιέχει όλα τα υπόλοιπα (rest) στοιχεία της λίστας, εκτός της κεφαλής, επομένως είναι πάντα λίστα.

Θα πρέπει να σημειωθεί ότι η κενή λίστα, όντας ειδική - τερματική περίπτωση, είναι η μοναδική περίπτωση λίστας που δεν έχει ούτε κεφαλή ούτε ουρά, και αποτελεί μία ειδική σταθερά της γλώσσας.



Σχήμα 7.2: *Κεφαλή και ουρά λίστας σε δενδρική μορφή*

Στην σημειογραφία της Prolog, η κεφαλή και η ουρά μπορούν να διαχωριστούν με το σύμβολο "|", δηλαδή μια λίστα μπορεί να αναπαρασταθεί ως [H|T], όπου Η οποιοσδήποτε όρος και Τ απαραίτητα μια λίστα. Το σύμβολο "|" ονομάζεται **κατασκευαστής λίστας** (list constructor). Για παράδειγμα, στην Prolog οι

δύο όροι [a,b,c] και [a|[b,c]] είναι ισοδύναμοι. Η ειδική αυτή μορφή χρησιμοποιείται συνήθως κατά την ενοποίηση, που περιγράφεται παρακάτω.

Ενοποίηση Λιστών

Ο γενικός κανόνας ενοποίησης, ο οποίος δηλώνει ότι για να είναι δύο όροι ενοποιήσιμοι θα πρέπει να έχουν το ίδιο συναρτησιακό σύμβολο, τον ίδιο αριθμό ορισμάτων και τα ορίσματά τους να είναι ενοποιήσιμα, ισχύει και στην περίπτωση των λιστών. Όμως σε μια έγκυρη λίστα το συναρτησιακό σύμβολο είναι το ίδιο (η τελεία ".") και σε κάθε επίπεδο του δένδρου υπάρχουν πάντοτε 2 ορίσματα, το οποίο οδηγεί στον απλούστερο κανόνα:

Δύο λίστες ενοποιούνται όταν έχουν τον ίδιο αριθμό στοιχείων και τα στοιχεία τους στις αντίστοιχες θέσεις είναι ενοποιήσιμα.

Ας εξετάσουμε τα πρώτα τετριμμένα παραδείγματα, που αφορούν ενοποίηση λιστών χωρίς μεταβλητές (Πίνακας 7.1):

Πίνακας 7.1: *Ενοποίηση λιστών χωρίς μεταβλητές*

Αρ.	Λίστα1	Λίστα2	Αποτέλεσμα
(1)	[a, b]	[a, b]	true (επιτυχία)
(2)	[1, 0, 0, 1]	[1, 0, 0, 1]	true (επιτυχία)
(3)	[b, a]	[a, b]	false (αποτυχία)
(4)	[a, b, b]	[a, b]	false (αποτυχία)

Τα παραδείγματα (1) και (2) είναι προφανή, καθώς οι όροι είναι πανομοιότυποι. Το παράδειγμα (3) αποτυγχάνει γιατί αν και τα στοιχεία είναι ίδια σε αριθμό, η σειρά τους είναι λάθος, δηλαδή τα στοιχεία μιας λίστας θεωρούνται διατεταγμένα. Το παράδειγμα (4) αποτυγχάνει καθώς ο αριθμός των στοιχείων των δύο λιστών είναι διαφορετικός.

Πίνακας 7.2: *Ενοποίηση λιστών με ύπαρξη μεταβλητών.*

Αρ.	Λίστα1	Λίστα 2	Αποτέλεσμα
(1)	[a, b]	[a, X]	{X=b} true
(2)	[a, b, c, X, e]	[a, Y, c, d, Z]	{X=d, Y=b, Z=e} true
(3)	[a, X, c, d, e]	[a, b, c, X, e]	false
(4)	[a, b, c, d]	[a, X]	false
(5)	[a, b, c, d]	[a|X]	{X = [b, c, d]} true
(6)	[0, 1]	[0, B]	{B = 1} true
(7)	[0, 1]	[0|B]	{B = [1]} true

Στα παραδείγματα (1) και (2) του Πίνακα 7.2 καθώς οι λίστες έχουν τον ίδιο αριθμό στοιχείων, οι δεσμεύσεις των μεταβλητών έχουν τέτοιες τιμές ώστε να γίνουν οι δύο όροι να γίνουν συντακτικά όμοιοι. Στο παράδειγμα (3) η πρώτη εμφάνιση της μεταβλητής X, αναθέτει σε αυτή τη τιμή b, ενώ η δεύτερη εμφάνιση την τιμή d, και καθώς κάτι τέτοιο δεν επιτρέπεται, η ενοποίηση αποτυγχάνει. Ενδιαφέρον παρουσιάζουν τα παραδείγματα (4) και (5): στην περίπτωση του παραδείγματος (4) η ενοποίηση αποτυγχάνει, καθώς η λίστα [a,X] είναι μια λίστα δύο στοιχειών, εφόσον οι όροι a και X διαχωρίζονται με κόμμα (“,”) και συνεπώς δεν είναι δυνατό να ενοποιηθεί με την λίστα [a,b,c,d], η οποία περιέχει τέσσερα στοιχεία. Αντίθετα στην περίπτωση (5), η σημειογραφία [a|X] δηλώνει μια λίστα της οποίας το πρώτο στοιχείο είναι το a, ενώ η ουρά της είναι οποιαδήποτε λίστα, καθώς η μεταβλητή X μπορεί να δεχθεί σαν τιμή οποιονδήποτε όρο της Prolog, άρα και οποιαδήποτε λίστα. Το γεγονός αυτό οδηγεί σε επιτυχή ενοποίηση, με παραγόμενη δέσμευση της μεταβλητής X τη λίστα [b, c, d] ({X = [b,c,d]}).

Η σημειογραφία κεφαλής-ουράς επιβάλει στον όρο που βρίσκεται στα δεξιά του συμβόλου “|” να είναι λίστα. Έτσι στο παράδειγμα (6), η ενοποίηση επιτυγχάνει με {B =1}, ενώ στο παράδειγμα (7) η δέσμευση της μεταβλητής είναι {B = [1]}, ακόμη και αν στην περίπτωση αυτή έχουμε μόνο ένα στοιχείο στην λίστα B.

Θα πρέπει να σημειωθεί ότι τα παραπάνω επεκτείνονται και όταν τα στοιχεία μιας λίστας είναι σύνθετοι όροι (πιθανά και λίστες), όπως φαίνεται από το ακόλουθο παράδειγμα:

```
[a,b,[21,12,43],name(nick),lang(greek)] = [a,b,X,name(Y),Z].
```

όπου η ενοποίηση επιτυγχάνει με δεσμεύσεις μεταβλητών τις {X = [21, 12, 43], Y = nick, Z = lang(greek)}.

Τέλος, το σύμβολο “|” είναι δυνατό να διαχωρίσει τη λίστα σε οποιοδήποτε σημείο, επιτρέποντας έτσι την προσπέλαση και των ακολούθων στοιχείων εκτός του πρώτου. Για παράδειγμα, η ακόλουθη ενοποίηση επιτυγχάνει:

```
[a,b,c,d] = [a,b|Rest]
```

με δεσμεύσεις μεταβλητών τις {Rest = [c,d]}.

7.2 Επεξεργασία Λιστών

Καθώς οι λίστες αποτελούν αναδρομικές δομές, η καταλληλότερη τεχνική για την επεξεργασία τους είναι η αναδρομή. Η μεθοδολογία της αναδρομής, όπως παρουσιάστηκε στο Κεφάλαιο 6, στην περίπτωση των λιστών γίνεται:

- Για να επιλύσω ένα πρόβλημα για μία **λίστα με N στοιχεία**, θεωρώ ότι έχω επιλύσει το ίδιο πρόβλημα για μία μικρότερη λίστα (έστω με **N-1 στοιχεία**) και μετά απλά συνθέτω την επί μέρους με τη συνολική λύση (αναδρομική περίπτωση).
- Περιγράφω τη λύση για την απλούστερη δυνατή περίπτωση λίστας και αντιπροσωπεύει το τετριμμένο πρόβλημα. Συνήθως αυτή είναι μια λίστα πεπερασμένου και μικρού μεγέθους (βασική περίπτωση).

Στις περισσότερες περιπτώσεις, η βασική περίπτωση αφορά την κενή λίστα ή μια λίστα με ένα στοιχείο. Στην αναδρομική περίπτωση συνήθως επιτελείται κάποια λειτουργία στην κεφαλή της λίστας, καλείται αναδρομικά το ίδιο το κατηγόρημα στην ουρά της λίστας και συνθέτονται οι δύο λύσεις.

Σαν πρώτο παράδειγμα θα εξεταστεί η υλοποίηση του κατηγορήματος is_valid_list/1, το οποίο επιτυγχάνει αν το μοναδικό όρισμα του είναι έγκυρη λίστα της γλώσσας Prolog. Από τον ορισμό της ενότητας 7.1 προκύπτει η ακόλουθη λεκτική περιγραφή:

1. Η κενή λίστα είναι έγκυρη λίστα.
2. Μια έγκυρη λίστα αποτελείται από ένα οποιονδήποτε όρο και μια λίστα.

Η λεκτική περιγραφή εκφράζεται σε Prolog ως ακολούθως (απόρροια του δηλωτικού χαρακτήρα του Λογικού Προγραμματισμού):

```
is_valid_list([]).
is_valid_list([_Head|Tail]):-
  is_valid_list(Tail).
```

Τα ακόλουθα αποτελούν παραδείγματα κλήσης του κατηγορήματος:

```
?- is_valid_list([]).
yes
?- is_valid_list([a,0]).
yes
?- is_valid_list(0).
no
?- is_valid_list([[a,b,c],name(bob,age(34))]).
yes
?- is_valid_list([program|[logic]]).
yes
```

Ο παραπάνω τύπος αναδρομικού ορισμού αποτελεί την απλούστερη περίπτωση αναδρομής, καθώς δεν επιτελείται καμία απολύτως λειτουργία στα στοιχεία της λίστας, παρά μόνο διατρέχονται παθητικά τα στοιχεία της. Το επόμενο παράδειγμα παρουσιάζει μια πολύ συνηθισμένη αναδρομική τεχνική προγραμματισμού και αφορά την εύρεση του τελευταίου στοιχείου μιας λίστας.

Το κατηγόρημα last_element/2 έχει δύο ορίσματα, το πρώτο είναι ένας όρος και το δεύτερο μια λίστα. Το κατηγόρημα επιτυγχάνει όταν το πρώτο όρισμα είναι το τελευταίο στοιχείο της λίστας που εμφανίζεται στο δεύτερο όρισμα. Για παράδειγμα η ακόλουθη ερώτηση:

```
?- last_element(d,[a,b,c,d])
```

θα πρέπει να είναι επιτυχής.

Ως συνήθως η λεκτική περιγραφή οδηγεί εύκολα στην υλοποίηση:

1. Αν μια λίστα περιέχει μόνο ένα στοιχείο, τότε αυτό είναι και το τελευταίο.
2. Εναλλακτικά, το τελευταίο στοιχείο μιας λίστας είναι το τελευταίο στοιχείο της ουράς της.

Η περίπτωση (1) απλά υλοποιείται από το γεγονός:

```
last_element(Last,[Last]).
```

Κάποιος άπειρος στον λογικό προγραμματισμό θα έγραφε:

```
last_element(X,[Last]) :- X=Last.
```

Φυσικά, το παραπάνω δεν είναι λάθος, αλλά ο ορισμός δεν εκμεταλλεύεται πλήρως το μηχανισμό ενοποίησης. Εδώ αξίζει να σημειωθεί η ισχύς του μηχανισμού ενοποίησης στην διατύπωση συνθηκών σε ένα κατηγόρημα. Το δεύτερο όρισμα του κανόνα [Last] ουσιαστικά κάνει δύο ελέγχους: α) ότι το όρισμα είναι όντως λίστα και β) ότι αυτή η λίστα έχει μήκος 1. Η ύπαρξη της ίδιας μεταβλητής Last στα δύο ορίσματα εξασφαλίζει ότι το μοναδικό στοιχείο της λίστας και το πρώτο όρισμα έχουν την ίδια τιμή, λόγω του χαρακτηριστικού μοναδικής ανάθεσης τιμών στις μεταβλητές. Οπότε, σε μια (σύντομη) γραμμή κώδικα ουσιαστικά υλοποιούνται όλοι οι έλεγχοι απαραίτητοι για την υλοποίηση της τερματικής συνθήκης του αναδρομικού κανόνα.

Η περίπτωση (2) υλοποιείται από τον κανόνα:

```
last_element(Last,[_Head|Tail]):-
  last_element(Last, Tail).
```

όπου απλά δηλώνεται ότι το τελευταίο στοιχείο μιας λίστας είναι το τελευταίο στοιχείο της ουράς της. Και στην περίπτωση αυτή εξασφαλίζεται μέσω του μηχανισμού ενοποίησης ότι το δεύτερο όρισμα α) είναι λίστα και β) δεν είναι η κενή λίστα, καθώς η κενή λίστα δεν έχει κεφαλή ούτε ουρά.

Το πλήρες κατηγόρημα είναι:

```
last_element(Last,[Last]).
last_element(Last,[_Head|Tail]):-
  last_element(Last, Tail).
```

Καθώς στα κατηγορήματα (συνήθως) δεν υπάρχει η έννοια της εισόδου/εξόδου όπως σε άλλες γλώσσες προγραμματισμού, ο παραπάνω ορισμός μπορεί να χρησιμοποιηθεί για:

- έλεγχο αν ένας όρος είναι το τελευταίο στοιχείο μιας λίστας, και για

- να επιστρέψει το τελευταίο στοιχείο μιας λίστας.

Για παράδειγμα:

```
?- last_element(5,[1,2,3,4,5]).
yes
?- last_element(X,[1,2,3,4,5]).
X = 5
yes
?- last_element(nick,[john,alex,nick,bob]).
no
?- last_element(X,[]).
no
```

Το τελευταίο παράδειγμα αναδεικνύει ακόμα μία διάσταση της χρησιμότητας του μηχανισμού της ενοποίησης: την εκμετάλλευση της αποτυχίας της ενοποίησης, όταν το κατηγόρημά μας πρέπει όντως να αποτύχει. Για την άδεια λίστα δεν ορίζεται το τελευταίο στοιχείο καθώς αυτή δεν έχει στοιχεία. Το κατηγόρημα που ορίσαμε ανταποκρίνεται αποτυγχάνοντας καθώς το δεύτερο όρισμα και των δύο κανόνων δεν μπορεί να ενοποιηθεί με την άδεια λίστα.

Λογική Σύζευξη Δυαδικών Αριθμών

Το παράδειγμα της λογικής σύζευξης (bitwise_and) των δυαδικών αριθμών που παρατέθηκε στην αρχή του κεφαλαίου, υλοποιείται εύκολα σύμφωνα με τα παραπάνω. Η αναπαράσταση ενός δυαδικού αριθμού ακολουθεί την μορφή μιας λίστας από 0 και 1, πχ. ο αριθμός 0101101 αναπαριστάται ως [0,1,0,1,1,0,1], με τα προφανή πλεονεκτήματα της δυνατότητας αναπαράστασης απεριόριστου μήκους και εύκολης εφαρμογής αναδρομικών τεχνικών.

Η πράξη της λογικής σύζευξης σε ένα μόνο ψηφίο ορίζεται απλά σαν ένα σύνολο γεγονότων:

```
bit_and(0,0,0).
bit_and(0,1,0).
bit_and(1,0,0).
bit_and(1,1,1).
```

Η δηλωτική λεκτική περιγραφή της λογικής σύζευξης (logical and) σε δύο δυαδικούς αριθμούς οποιουδήποτε μεγέθους είναι:

- Η λογική σύζευξη δύο μονοψήφιων δυαδικών αριθμών (1-bit) είναι ένας μονοψήφιος δυαδικός αριθμός που προκύπτει από την λογική σύζευξη των (μοναδικών) ψηφίων των δύο αριθμών (βασική περίπτωση).
- Αλλιώς, η λογική σύζευξη δύο αριθμών περισσότερων bits είναι ένας αριθμός, όπου το πρώτο ψηφίο του είναι η λογική σύζευξη των πρώτων ψηφίων των δύο αριθμών, και τα υπόλοιπα ψηφία η λογική σύζευξη των υπολοίπων στοιχείων των αριθμών πλην του πρώτου.

Η περιγραφή υλοποιείται στους ακόλουθους Prolog κανόνες (κατηγόρημα bitwise_and/3):

```
bitwise_and([Bit1],[Bit2],[BitR]):-
  bit_and(Bit1,Bit2,BitR).
bitwise_and([Bit1|RBits1], [Bit2|RBits2], [BitR|Result]):-
  bit_and(Bit1, Bit2, BitR),
  bitwise_and(RBits1, RBits2, Result).
```

Το τρίτο όρισμα του δεύτερου κανόνα ίσως ξενίσει σε μια πρώτη ανάγνωση, αλλά είναι η ακριβής δηλωτική αποτύπωση του γεγονότος ότι το αποτέλεσμα είναι μια λίστα που αποτελείται από το ψηφίο BitR ως κεφαλή και την λίστα Result ως ουρά. Σε σχέση με τις παραδοσιακές, διαδικαστικές γλώσσες προγραμματισμού, τα κατηγορήματα της Prolog "επικοινωνούν" με τον υπόλοιπο κώδικα (υπό την έννοια εισόδου/εξόδου των κατηγορημάτων) μόνο μέσω της κεφαλής του κανόνα. Συνεπώς η "έξοδος" ενός κατηγορήματος γίνεται μέσω ενός (ή περισσότερων) ορισμάτων στη κεφαλή του κανόνα. Συνεπώς η "σύνθεση" της "εξόδου" γίνεται στην κεφαλή του κανόνα, μέσω μεταβλητών που έχουν ήδη λάβει τιμή, είτε κατά την "είσοδο" στο κατηγόρημα

(μέσω ενοποίησης με την κεφαλή του κανόνα), ή στο σώμα του κανόνα κατά την επιτυχή "εκτέλεση" των κλήσεων του σώματος.

Οι ερωτήσεις στην Prolog θα δώσουν:

```
?- bitwise_and([0, 1], [1, 1], R).
R = [0, 1]
Yes
?- bitwise_and([0, 1, 1, 0], [1, 1, 0, 0], R).
R = [0, 1, 0, 0]
Yes
?- bitwise_and([0, 1, 1, 0, 1], [1, 1, 0, 0, 1], R).
R = [0, 1, 0, 0, 1]
Yes
?- bitwise_and([], [], R).
No
```

Η τελευταία ερώτηση αποτυγχάνει γιατί σύμφωνα με τον ορισμό, δεν ορίζεται η πράξη για αριθμούς 0-bit. Όπως και στην προηγούμενη περίπτωση, το κατηγόρημα μπορεί να χρησιμοποιηθεί με περισσότερους τρόπους, καθένας από τους οποίους αντιστοιχεί σε ένα διαφορετικό ερώτημα:

Ποιου δυαδικού αριθμού η σύζευξη με το [1,0,1] δίνει σαν αποτέλεσμα το [0,0,1];

```
?- bitwise_and(X, [1, 0, 1], [0, 0, 1]).
X = [0, 0, 1]
Yes
```

Ποιων δυαδικών αριθμών η σύζευξη είναι το [0,1,1];

```
?- bitwise_and(X, Y, [0, 1, 1]).
X = [0, 1, 1]
Y = [0, 1, 1]
Yes (more)
X = [0, 1, 1]
Y = [1, 1, 1]
Yes (more)
X = [1, 1, 1]
Y = [0, 1, 1]
Yes (more)
No
```

Η τελευταία ερώτηση δείχνει πόσο ισχυρός μπορεί να είναι ο μηχανισμός οπισθοδρόμησης ο οποίος παράγει εναλλακτικές λύσεις στην περίπτωση των λογικών προγραμμάτων.

7.3 Αναδρομικές Τεχνικές: Μέγιστο Στοιχείο μιας Λίστας

Ένα ενδιαφέρον κατηγόρημα, από την άποψη υλοποίησης διαφορετικών τεχνικών αναδρομής, είναι το κατηγόρημα list_max/2. Αν και θα εξετάσουμε το κατηγόρημα για λίστες αριθμών, ο κώδικας του κατηγορήματος παραμένει σχεδόν ίδιος (με εξαίρεση τον έλεγχο διάταξης) για οποιουσδήποτε όρους. Έτσι, το κατηγόρημα list_max(List,M) το οποίο θα εξετάσουμε, επιτυγχάνει όταν το δεύτερο όρισμά του M είναι το μέγιστο στοιχείο μιας λίστας αριθμών List. Για παράδειγμα, η αναμενόμενη συμπεριφορά του κατηγορήματος θα είναι:

```
?- list_max([2,4,5,10,1,2,2,3],M).
M = 10
```

Το μέγιστο μιας λίστας αποτελεί εξαιρετικό παράδειγμα υλοποίησης αναδρομικών τεχνικών. Στις επόμενες ενότητες θα παρουσιαστούν διαφορετικές υλοποιήσεις του κατηγορήματος, λειτουργώντας σαν παραδείγματα εφαρμογής αναδρομικών τεχνικών σε λίστες.

Η δηλωτική προσέγγιση

Αρχικά θα πρέπει να αναφέρουμε ότι το κατηγόρημα απαιτεί μόνο δυο ορίσματα, ένα για την λίστα αριθμών και ένα για το μέγιστο στοιχείο. Η δηλωτική περιγραφή είναι η ακόλουθη:

- Σε μια λίστα με ένα στοιχείο, μέγιστο είναι αυτό το στοιχείο.
- Αν η κεφαλή της λίστας είναι όρος μεγαλύτερος από το μέγιστο στοιχείο της ουράς της, τότε η κεφαλή αποτελεί το μέγιστο στοιχείο της λίστας.
- Αν το μέγιστο στοιχείο της ουράς της λίστας είναι μεγαλύτερο από την κεφαλή της τότε αυτό είναι το μέγιστο στοιχείο.

Η υλοποίηση του πρώτου μέρους του ορισμού είναι το γεγονός:

```
list_max([X],X).
```

Στην περίπτωση αυτή, όπως και στο κατηγόρημα last_element/2, το πρώτο όρισμα εξασφαλίζει ότι η λίστα έχει ένα και μοναδικό στοιχείο.

Οι υλοποιήσεις του δεύτερου και τρίτου μέρους του αναδρομικού ορισμού είναι κατά σειρά οι κανόνες:

```
list_max([H|T],H):-
  list_max(T,MT),
  H > MT.
list_max([H|T],MT):-
  list_max(T,MT),
  H =< MT.
```

Οι αντίστοιχες ερωτήσεις στην Prolog είναι:

```
?- list_max([2,5,6,4],M).
M = 6 .
?- list_max([2,5,6,4],6).
true .
?- list_max([2,5,6,4],8).
false.
```

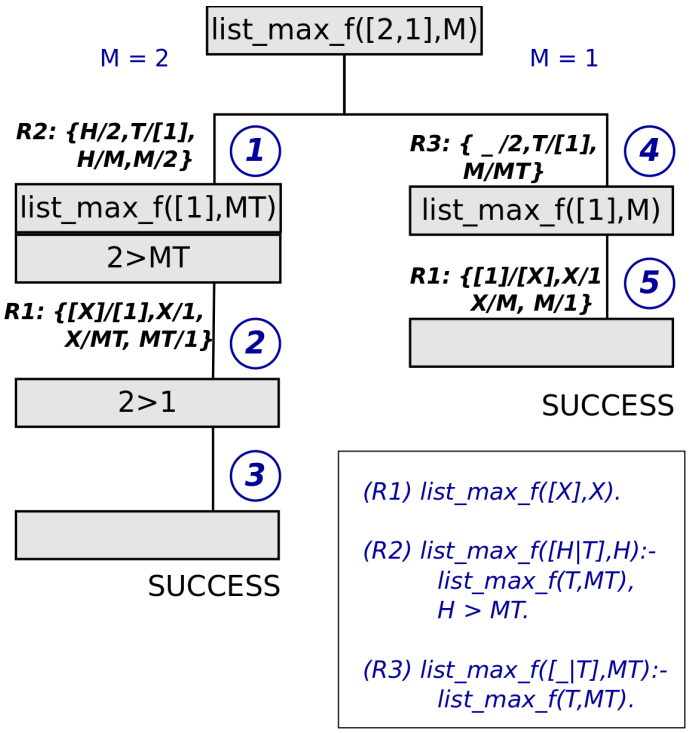
Σε μια πρώτη ανάγνωση, ο έλεγχος που διενεργείται στον τρίτο κανόνα δεν είναι απαραίτητος, καθώς έχει ήδη γίνει έλεγχος ότι η κεφαλή είναι μεγαλύτερη, πριν η εκτέλεση προχωρήσει στον επόμενο κανόνα. Έτσι, θα μπορούσε να προταθεί ένα κατηγόρημα με την ακόλουθη υλοποίηση (το επίθεμα _f σηματοδοτεί το false - το κατηγόρημα είναι **λάθος**):

```
list_max_f([X],X).
list_max_f([H|T],H):-
  list_max_f(T,MT),
  H > MT.
list_max_f([_|T],MT):-
  list_max_f(T,MT).
```

Αν για το παραπάνω κατηγόρημα ζητηθούν όλες οι απαντήσεις μέσω οπισθοδρόμησης, τότε η συμπεριφορά του είναι η ακόλουθη:

```
?- list_max_f([2,1],M).
M = 2 ;
M = 1 ;
```

Η πρώτη λύση είναι αναμενόμενη, ενώ η δεύτερη όχι. Το πρόβλημα υφίσταται γιατί κατά την οπισθοδρόμηση, δεν γίνεται έλεγχος εφαρμογής του τρίτου κανόνα και ελλείψει ενός τέτοιου ελέγχου, ο κανόνας πετυχαίνει πάντα, ακόμα και αν η κεφαλή είναι μεγαλύτερη του μέγιστου της ουράς. Εφόσον, εξετάζονται όλες οι λύσεις που δίνουν οι εναλλακτικοί κανόνες και εφόσον ο τρίτος κανόνας δίνει λύση χωρίς έλεγχο, η λάθος λύση επιστρέφεται από την Prolog. Στο σχήμα που ακολουθεί (Σχήμα 7.3), απεικονίζεται το δένδρο αναζήτησης για τη παραπάνω ερώτηση, μαζί με τη σειρά δημιουργίας των κόμβων. Όπως φαίνεται, τα δύο κλαδιά οδηγούν αμφότερα σε λύσεις, τις οποίες η Prolog θα επιστρέψει.



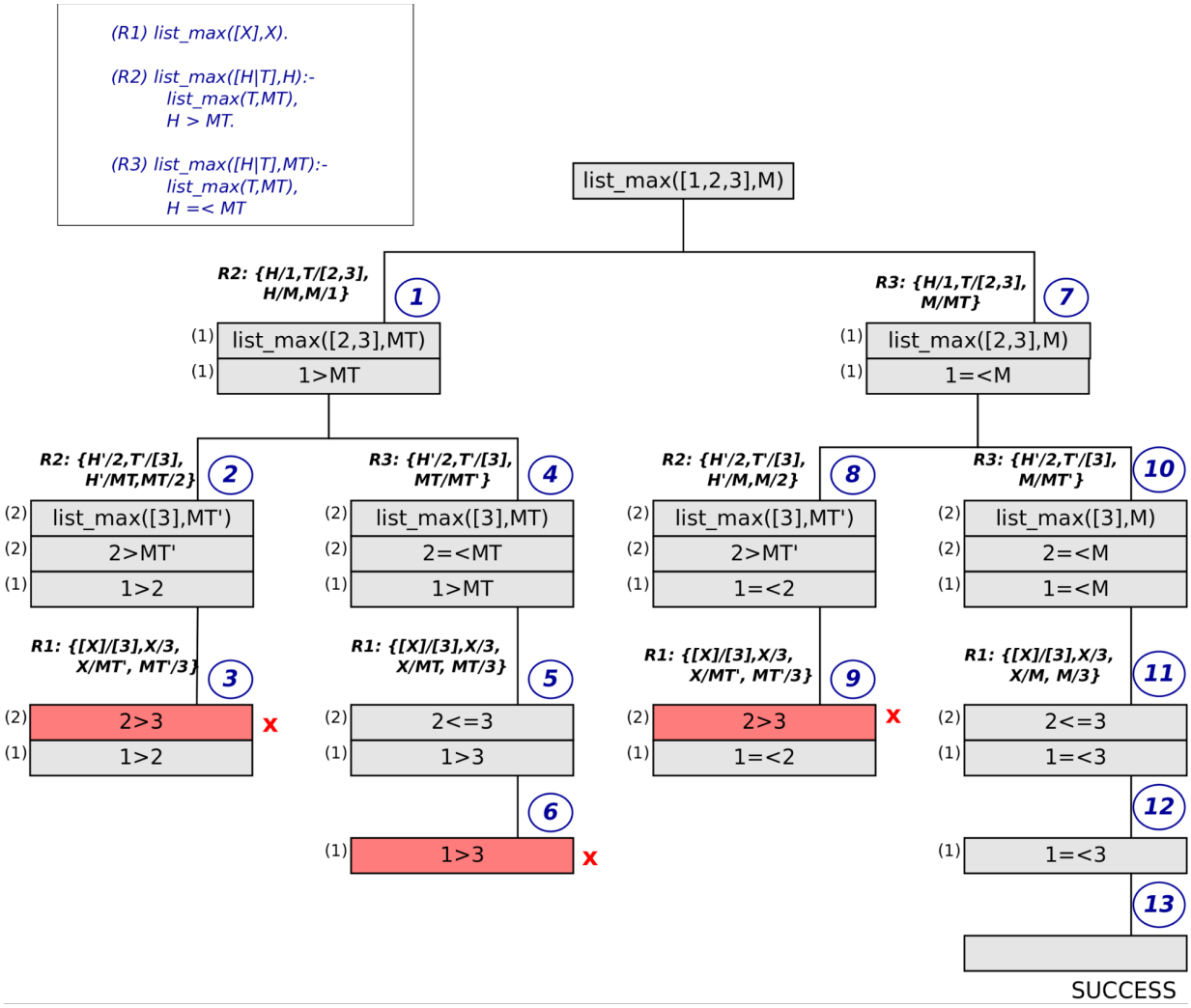
Σχήμα 7.3: *Αναζήτηση λύσεων στο κατηγόρημα list_max_f/2*

Η οπισθοδρόμηση κατά συνέπεια, είναι πολύ ισχυρός μηχανισμός, αλλά θα πρέπει να έχει ληφθεί μέριμνα, ώστε να εξασφαλίζεται ότι επιστρέφονται πάντα οι αναμενόμενες λύσεις. Θα πρέπει να σημειωθεί, ότι είναι δυνατό να αποφευχθούν τέτοιοι περιττοί έλεγχοι με τη χρήση του κατηγορήματος της αποκοπής που θα εξετάσουμε σε επόμενο κεφάλαιο (Κεφάλαιο 9).

Αν και το παραπάνω κατηγόρημα, είναι μια εξαιρετική δηλωτική περιγραφή του μέγιστου μιας λίστας, η υλοποίηση παρουσιάζει σημαντικά προβλήματα απόδοσης. Στη περίπτωση που η λίστα είναι ταξινομημένη κατά αύξουσα σειρά, η Prolog θα εξαντλήσει το δένδρο αναζήτησης, πριν βρει λύση, με σημαντική αύξηση στο χρόνο εκτέλεσης. Ας δούμε ένα απλό παράδειγμα:

```
?- list_max([1,2,3],M).
M = 3 .
```

Ο μηχανισμός εκτέλεσης επιλέγει σε κάθε βήμα, πρώτα τον δεύτερο κανόνα του κατηγορήματος, ο οποίος θέτει προς απόδειξη την υπόθεση ότι η κεφαλή της λίστας είναι μεγαλύτερη από το μέγιστο στοιχείο της ουράς της, το οποίο δεν είναι αληθές σε όλες τις περιπτώσεις. Όπως φαίνεται στο σχήμα Σχήμα 7.4, το κλαδί του δένδρου που οδηγεί σε λύση είναι το δεξιότερο, με αποτέλεσμα να δημιουργείται τελευταίο (χρονικά), όπως φαίνεται από τη σειρά δημιουργίας των κόμβων, με τις αντίστοιχες συνέπειες στο χρόνο εκτέλεσης.



Σχήμα 7.4: *Δένδρο αναζήτησης της ερώτησης list_max([1,2,3],M). Στο σχήμα οι αριθμοί αντιστοιχούν στη σειρά δημιουργίας των κόμβων.*

Βελτιώνοντας την απόδοση

Μια πρώτη προσέγγιση στη επίλυση του προβλήματος είναι να ακολουθηθεί η κλασική τεχνική, δηλαδή να "αποθηκεύεται" σε κάθε βήμα το μεγαλύτερου στοιχείο μεταξύ του μεγίστου που έχει βρεθεί μέχρι το σημείο εκείνο (προσωρινό μέγιστο), και του "τρέχοντος" στοιχείου της λίστας, έως ότου εξαντληθούν όλα τα στοιχεία. Ως "τρέχον" στοιχείο θεωρείται απλά η κεφαλή της λίστας. Εφόσον, η Prolog είναι μια γλώσσα μοναδικής ανάθεσης και η χρήση καθολικών μεταβλητών πρέπει να αποφεύγεται, το προσωρινό μέγιστο μπορεί απλά να "αποθηκευτεί" ως ένα πρόσθετο όρισμα στο κατηγόρημα. Έτσι προκύπτει το κατηγόρημα list_max_aux(List,MSF,M), με το πρώτο όρισμα List τη λίστα της οποίας το μέγιστο αναζητείται, MSF το προσωρινό μέγιστο και M το (τελικό) μέγιστο στοιχείο της λίστας. Ο ορισμός είναι ο ακόλουθος :

- Όταν εξαντληθούν όλα τα στοιχεία της λίστας, το προσωρινό μέγιστο είναι το μέγιστο στοιχείο της λίστας.
- Αν η κεφαλή της λίστας είναι μεγαλύτερη από το προσωρινό μέγιστο, τότε αναζητείτε το μέγιστο της λίστας στην ουρά, με προσωρινό μέγιστο την κεφαλή.
- Αν η κεφαλή της λίστας είναι μικρότερη ή ίση από το προσωρινό μέγιστο, τότε αναζητείτε το μέγιστο της λίστας στην ουρά, με το ίδιο προσωρινό μέγιστο.

Ο παραπάνω ορισμός, δεν ακολουθεί απόλυτα τη δηλωτική προσέγγιση στην περιγραφή των κατηγορημάτων, επιλέχθηκε όμως καθώς στη παραπάνω προσέγγιση η δηλωτική περιγραφή θα ήταν δυσνόητη. Η υλοποίηση του πρώτου μέρους του ορισμού είναι:

```
list_max_aux([],MSF,MSF).
```

Ενώ οι υλοποιήσεις των δύο επομένων:

```
list_max_aux([H|T],MSF,Max):-
  H > MSF,
  list_max_aux(T,H,Max).
list_max_aux([H|T],MSF,Max):-
  H =< MSF,
  list_max_aux(T,MSF,Max).
```

Αξίζει να σημειωθεί ότι η μεταβλητή Max παραμένει αναλλοίωτη στο δεύτερο και τρίτο κανόνα και δεσμεύεται μόνο στο πρώτο γεγονός, δηλαδή στη τερματική συνθήκη του αναδρομικού κατηγορήματος. Αυτή η τιμή "μεταφέρεται" από το φύλλο του κλαδιού αναζήτησης στη ρίζα του δένδρου και αποτελεί μια από τις κλασικές αναδρομικές τεχνικές.

Στο παραπάνω κατηγόρημα υπάρχουν δύο αλληλένδετα θέματα. Το πρώτο αφορά το γεγονός ότι σύμφωνα με τις αρχικές προδιαγραφές το κατηγόρημα που βρίσκει το μέγιστο στοιχείο πρέπει να έχει δύο ορίσματα, και το δεύτερο το πως θα κληθεί το κατηγόρημα list_max_aux/3, δηλαδή κατά την αρχική κλήση ποιο θα είναι το πρώτο στοιχείο που θα χρησιμοποιηθεί ως προσωρινό μέγιστο. Η λύση στο δεύτερο πρόβλημα είναι προφανής: το πρώτο στοιχείο της αρχικής λίστας (κεφαλή) θα αποτελέσει το αρχικό προσωρινό μέγιστο. Η λύση στο πρώτο πρόβλημα είναι επίσης απλή: θα υπάρχει ένα κατηγόρημα δύο ορισμάτων (list_max_b/2) το οποίο θα καλεί το κατηγόρημα list_max_aux/3, και το οποίο θα θέτει τη κεφαλή της αρχικής λίστας ως προσωρινό μέγιστο.

```
list_max_b([H|T],Max):-
  list_max_aux(T,H,Max).
```

Άρα η υλοποίηση της εύρεσης του μεγίστου στοιχείου είναι το κατηγόρημα list_max_b/2 (b - β' υλοποίηση) που χρησιμοποιεί ως βοηθητικό κατηγόρημα το list_max_aux/3 (auxiliary - βοηθητικό). Η συμπεριφορά του κατηγορήματος είναι η αναμενόμενη:

```
?- list_max_b([1,2,3,4],M).
M = 4
?- list_max_b([1,21,3,4],M).
M = 21
```

Γιατί όμως είναι το κατηγόρημα ποιο αποδοτικό; Γιατί απλά ο έλεγχος (για μεγαλύτερο κλπ.) προηγείται της αναδρομικής κλήσης με αποτέλεσμα τα κλαδιά τα οποία δεν οδηγούν σε λύση να αποτυγχάνουν νωρίς, χωρίς να γίνονται περιττοί υπολογισμοί. Αυτό ονομάζεται και έγκαιρο κλάδεμα (early pruning) και αποτελεί καθοριστικό παράγοντα απόδοσης σε οποιαδήποτε μορφή αναζήτησης λύσης.

"Λοξο-κοιτώντας" στον Συναρτησιακό Προγραμματισμό

Είναι δυνατό να υπάρξει λύση με ένα αποδοτικό κατηγόρημα δύο ορισμάτων, που να μην χρησιμοποιεί βοηθητικά κατηγορήματα. Η ιδέα είναι απλή και εδράζεται στον συναρτησιακό προγραμματισμό. Σε κάθε βήμα ελέγχονται τα δύο πρώτα στοιχεία της λίστας, και το μεγαλύτερο των δύο παραμένει στην λίστα κατά την αναδρομική κλήση. Έτσι επιτυγχάνεται α) να μειώνεται το μέγεθός της λίστας (και του προβλήματος κατ' επέκταση) β) η κεφαλή της λίστας να αντιστοιχεί πάντα στο προσωρινό μέγιστο. Προφανώς, η διαδικασία τερματίζει όταν, στη λίστα υπάρχει μόνο ένα στοιχείο. Ο ορισμός του κατηγορήματος list_max_c/2 είναι

- Σε μια λίστα με ένα στοιχείο, μέγιστο είναι αυτό το στοιχείο.
- Αλλιώς, αν το πρώτο στοιχείο της λίστας είναι μεγαλύτερο από το δεύτερο, τότε το μέγιστο στοιχείο της είναι το μέγιστο της λίστας που αποτελείται από το πρώτο στοιχείο και τα υπόλοιπα στοιχεία της λίστας.

- Αλλιώς, αν το πρώτο στοιχείο της λίστας είναι μικρότερο ή ίσο από το δεύτερο, τότε το μέγιστο στοιχείο της είναι το μέγιστο της λίστας που αποτελείται από το δεύτερο στοιχείο και τα υπόλοιπα στοιχεία της λίστας.

Η υλοποίηση του παραπάνω ορισμού είναι η ακόλουθη:

```
list_max_c([M],M).
list_max_c( [H1,H2|T],Max):-
  H1 > H2,
  list_max_c([H1|T],Max).
list_max_c( [H1,H2|T],Max):-
  H1 =< H2,
  list_max_c([H2|T],Max).
```

Οι ερωτήσεις που μπορούν να γίνουν στην Prolog είναι παρόμοιες με εκείνες που παρουσιάστηκαν παραπάνω:

```
?- list_max_c([1,21,3,4],X).
X = 21 .
?- list_max_c([1,21,3,4],21).
true .
```

7.4 Κλασικά Κατηγορήματα Διαχείρισης Λιστών

Στα κλασικά κατηγορήματα διαχείρισης λιστών, περιλαμβάνονται κατηγορήματα που αφορούν τον έλεγχο συμπερίληψης μέλους της λίστας, την εύρεση του μήκους μιας λίστας, την διαγραφή ενός στοιχείου από μια λίστα, την συνένωση δύο λιστών και τέλος την αναστροφή μιας λίστας. Στα επόμενα θα εξετάσουμε αυτά τα κατηγορήματα, δίνοντας τους αντίστοιχους αναδρομικούς ορισμούς και τις υλοποιήσεις τους. Καθώς η διαχείριση λιστών βασίζεται στις έννοιες της κεφαλής και της ουράς της λίστας, έννοιες που αποτυπώνουν ουσιαστικά τους βασικούς τρόπους προσπέλασης των στοιχείων μιας λίστας, όλοι οι αναδρομικοί ορισμοί προκύπτουν σαν συνδυασμός αυτών των βασικών λειτουργιών.

Θα πρέπει να τονιστεί ότι σχεδόν σε όλες τις υλοποιήσεις της γλώσσας Prolog, τα κατηγορήματα αυτά αποτελούν αναπόσπαστο μέρος των βασικών βιβλιοθηκών της γλώσσας (ενσωματωμένα κατηγορήματα).

Μέλος της Λίστας

Το κατηγόρημα list_member/2 συσχετίζει μία λίστα με τα μέλη της και επιτυγχάνει αν ο όρος του πρώτου ορίσματός του αποτελεί μέλος της λίστας που εμφανίζεται στο δεύτερο όρισμα. Η δηλωτική περιγραφή του πλησιάζει εκείνη του κατηγορήματος last_element/2, η οποία παρουσιάστηκε σε προηγούμενη ενότητα του κεφαλαίου. Έτσι:

- Ένας όρος είναι μέλος μιας λίστας αν είναι το πρώτο στοιχείο της, ή
- Ένας όρος είναι μέλος μιας λίστας αν ο όρος είναι μέλος των υπολοίπων στοιχείων πλην του πρώτου.

Ο παραπάνω ορισμός, είναι ισοδύναμος με τον ακόλουθο, ο οποίος εκφράζεται με την ορολογία των λιστών:

Ένας όρος είναι μέλος μιας λίστα αν:

- είναι η κεφαλή της λίστας, ή
- είναι μέλος της ουράς της λίστας.

Η υλοποίηση του πρώτου μέρους είναι απλά το γεγονός:

```
list_member(X,[X|_]).
```

ενώ η υλοποίηση του δεύτερου είναι:

```
list_member(X,[_|Tail]):-
  list_member(X,Tail).
```

Οι ερωτήσεις στην Prolog θα δώσουν:

```
?- list_member(a,[a,b,c]).
```

```
true .
?- list_member(c,[a,b,c,d,e]).
true .
?- list_member(foo,[a,b,c,d,e]).
false.
?- list_member(a,[a,b,a,d,a]).
true ;
true ;
true ;
false.
```

Οι απαντήσεις στις πρώτες ερωτήσεις είναι προφανείς. Η τελευταία ερώτηση δείχνει πως το κατηγόρημα επιτυγχάνει κατά την οπισθοδρόμηση τρεις φορές, όσες δηλαδή εμφανίζεται ο όρος a στην λίστα. Μια τέτοια συμπεριφορά μπορεί να είναι επιθυμητή, αλλά σε κάποιες περιπτώσεις απλά αυξάνει τον χρόνο εκτέλεσης του προγράμματος χωρίς να προσφέρει εναλλακτικές λύσεις και θα πρέπει να αποφεύγεται. Ο τρόπος με τον οποίο μπορεί να γίνει η αποφυγή των περιττών λύσεων θα περιγραφεί σε επόμενο κεφάλαιο (Κεφάλαιο 9).

Τέλος, ενδιαφέρον παρουσιάζει η ερώτηση, στην οποία το πρώτο όρισμα είναι μεταβλητή. Όπως φαίνεται στην παρακάτω ερώτηση:

```
?- list_member(X,[a,b,c,d,e]).
X = a ;
X = b ;
X = c ;
X = d ;
X = e ;
false.
```

η Prolog θα δεσμεύσει διαδοχικά τη μεταβλητή X στα στοιχεία της λίστας κατά την οπισθοδρόμηση. Το παραπάνω γίνεται προφανές, αν "αναγνωσθεί" η ερώτηση ως: "ποιος όρος X είναι μέλος της λίστας [a,b,c,d,e];". Έτσι, ο ίδιος κώδικας, μπορεί να χρησιμοποιηθεί για να προσπελάσουμε με συστηματικό τρόπο τα στοιχεία μιας λίστας. Αυτή η συστηματική προσπέλαση είναι ιδιαίτερα χρήσιμη κατά την εφαρμογή τεχνικών αναζήτησης λύσης όπως θα εξεταστεί στο Κεφάλαιο 11.

Σημείωση: Όλες οι υλοποιήσεις της Prolog προσφέρουν το κατηγόρημα member/2 που αφορά τον έλεγχο συμπερίληψης στοιχείου σε μια λίστα.

Μήκος της Λίστας

Η εύρεση του μήκους μιας λίστας είναι μια από τις κλασικές λειτουργίες στην συγκεκριμένη δομή, σε όλες τις γλώσσες προγραμματισμού. Έτσι, το κατηγόρημα list_length(List,Len) (list_length/2) έχει προφανώς δύο ορίσματα, το πρώτο είναι μια λίστα List, ενώ το δεύτερο το μήκος της Len. Το κατηγόρημα επιτυγχάνει όταν Len είναι το μήκος της λίστας List.

Η δηλωτική περιγραφή του κατηγορήματος είναι:

- Το μήκος της κενής λίστας είναι 0.
- Το μήκος μιας λίστας είναι κατά ένα μεγαλύτερο από το μήκος της ουράς της.

Η υλοποίηση του πρώτου μέρους του ορισμού είναι το γεγονός:

```
list_length([],0).
```

Η υλοποίηση της δεύτερου μέρους του ορισμού είναι ο κανόνας:

```
list_length([_|Tail],Length):-
  list_length(Tail,LengthTail),
  Length is LengthTail + 1.
```

Οι ακόλουθες ερωτήσεις σε Prolog δίνουν:

```
?- list_length([],L).
L = 0.
?- list_length([a,b,c],L).
```

```
L = 3.
?- list_length([a,b,c],4).
false.
```

Στον αναδρομικό κανόνα του κατηγορήματος, θα πρέπει να σχολιαστεί λίγο περισσότερο η σειρά των κλήσεων στο σώμα του κανόνα. Αν και θα ήταν αναμενόμενο η σειρά των κλήσεων να μη παίζει σημαντικό ρόλο (παρά μόνο στην απόδοση του προγράμματος), εντούτοις η ύπαρξη αριθμητικών πράξεων επιβάλλει το δεξιό μέλος του κατηγορήματος is/2 να είναι πλήρως ορισμένο (χωρίς ελεύθερες μεταβλητές). Άρα η μοναδική μεταβλητή LengthTail που εμφανίζεται στο δεξιό μέρος της κλήσης του is/2 θα πρέπει να έχει τιμή πριν η εκτέλεση φτάσει σε αυτό το σημείο και κατά συνέπεια η αναδρομική κλήση στο κατηγόρημα list_length/2 πρέπει να προηγείται.

Σημειώση: Σε όλες οι υλοποιήσεις της Prolog, το αντίστοιχο κατηγόρημα είναι το length/2.

Διαγραφή Στοιχείου Λίστας

Η διαγραφή ενός στοιχείου μιας λίστας απαιτεί την υλοποίηση ενός κατηγορήματος με τρία ορίσματα. Αυτό μπορεί να ξενίσει αρχικά τον προγραμματιστή άλλων γλωσσών προγραμματισμού, καθώς η διαγραφή στοιχείου στις γλώσσες καταστροφικής ανάθεσης (C, Java, κλπ) απαιτεί μόνο δύο ορίσματα, καθώς ενημερώνεται η ίδια δομή σαν αποτέλεσμα της λειτουργίας. Στην περίπτωση των γλωσσών μοναδικής ανάθεσης, όπως είναι η Prolog, το αποτέλεσμα της διαγραφής στοιχείου από μια λίστα είναι μια νέα λίστα, αντίγραφο της πρώτης, από την οποία έχει αφαιρεθεί το στοιχείο προς διαγραφή. Έτσι, το κατηγόρημα list_delete/2 συσχετίζει δύο λίστες και ένα στοιχείο, αυτό κατά το οποίο διαφέρουν. Για παράδειγμα το list_delete(X, List, NewList) επιτυγχάνει όταν η λίστα NewList είναι η λίστα List από την οποία έχει διαγραφεί το στοιχείο X. Η δηλωτική περιγραφή της διαγραφής ενός στοιχείου μιας λίστας είναι:

- Αν το προς διαγραφή στοιχείο είναι η κεφαλή της αρχικής λίστας τότε η νέα λίστα είναι η ουρά της, ή
- η νέα λίστα έχει ως κεφαλή την κεφαλή της αρχικής λίστας και ως ουρά την λίστα που προκύπτει από την διαγραφή του στοιχείου από την ουρά της αρχικής λίστας.

Η υλοποίηση του πρώτου μέρους είναι απλά το γεγονός:

```
list_delete(X,[X|Tail],Tail).
```

Το δεύτερο μέρος υλοποιείται από τον κανόνα:

```
list_delete(X,[Y|Tail],[Y|List]):-
     list_delete(X,Tail,List).
```

Προσοχή θα πρέπει να δοθεί στον δεύτερο κανόνα του κατηγορήματος. Η μεταβλητή Y ουσιαστικά "μεταφέρει" την κεφαλή της αρχικής λίστας που δεν έχει διαγραφεί στην νέα λίστα, σύμφωνα με τον ορισμό που δόθηκε παραπάνω.

```
?- list_delete(a,[a,b,c,d],L).
L = [b, c, d] .
?- list_delete(c,[a,b,c,d],L).
L = [a, b, d] .
?- list_delete(c,[a,b,c,d,c,d,e],L).
L = [a, b, d, c, d, e] ;
L = [a, b, c, d, d, e] ;
false.
```

Η τελευταία ερώτηση έχει ιδιαίτερο ενδιαφέρον. Καθώς το στοιχείο c εμφανίζεται δύο φορές στην αρχική λίστα, η Prolog επιστρέφει δύο απαντήσεις κατά την οπισθοδρόμηση, μια για κάθε εμφάνιση του στοιχείου στη λίστα. Επισημαίνεται ότι ο δεύτερος κανόνας του κατηγορήματος δεν επιβάλλει στις τιμές των μεταβλητών X και Y να πάρουν διαφορετικές τιμές, οπότε επιτρέπει αυτή την συμπεριφορά.

Αν το πρώτο όρισμα είναι μεταβλητή, το κατηγόρημα θα "διαγράψει" διαδοχικά κατά την οπισθοδρόμηση τα στοιχεία της λίστας, όπως φαίνεται στην ακόλουθη ερώτηση:

```
?- list_delete(A,[a,b,c,d],L).
A = a,
L = [b, c, d] ;
```

```
A = b,
L = [a, c, d] ;
A = c,
L = [a, b, d] ;
A = d,
L = [a, b, c] ;
false.
```

Τέλος, ιδιαίτερο ενδιαφέρον παρουσιάζει η περίπτωση, όπου η αρχική λίστα είναι μια μεταβλητή, όπως φαίνεται στο ακόλουθο ερώτημα:

```
?- list_delete(1,L,[a,b,c,d]).
L = [1, a, b, c, d] ;
L = [a, 1, b, c, d] ;
L = [a, b, 1, c, d] ;
L = [a, b, c, 1, d] ;
L = [a, b, c, d, 1] ;
false.
```

Τα αποτελέσματα είναι προφανή αν θεωρήσουμε ότι η ερώτηση ουσιαστικά διατυπώνει το ερώτημα "ποια είναι η λίστα L από την οποία αν διαγράψω το στοιχείο 1 θα έχω σαν αποτέλεσμα τη λίστα [a,b,c,d];". Άρα, το κατηγόρημα list_delete/3 μπορεί να χρησιμοποιηθεί για την διαγραφή ενός στοιχείου από μια λίστα, αλλά και για την εισαγωγή ενός στοιχείου σε μια λίστα.

Σημειώση: Σε όλες οι υλοποιήσεις της Prolog, υπάρχει αντίστοιχο ενσωματωμένο κατηγόρημα με το list_delete/3. Στην SWI-Prolog αυτό είναι το select/3, ενώ στην ECLiPSe Prolog είναι το delete/3.

Συνένωση δύο λιστών

Το τελευταίο κατηγόρημα διαχείρισης λιστών αφορά την συνένωση δύο λιστών σε μια νέα λίστα. Το κατηγόρημα list_append/3 συσχετίζει τρεις λίστες, η τρίτη εκ των οποίων περιέχει τα στοιχεία των δύο πρώτων με την ίδια σειρά. Το κατηγόρημα έχει τρία ορίσματα, με τα πρώτα δύο να είναι οι λίστες προς συνένωση και το τρίτο η λίστα η οποία προκύπτει ως αποτέλεσμα της συνένωσης. Για παράδειγμα, η συνένωση των λιστών L1 = [a,b,c,d] και L2 = [e,f,g] είναι η λίστα L3 =[a,b,c,d,e,f,g]. Όπως φαίνεται από το παράδειγμα, κατά την συνένωση η σειρά των στοιχείων των δύο λιστών διατηρείται. Η δηλωτική περιγραφή της συνένωσης είναι η ακόλουθη:

- Η συνένωση της κενής λίστας με οποιαδήποτε λίστα είναι η δεύτερη λίστα.
- Η συνένωση μιας λίστας με οποιαδήποτε λίστα, είναι μια νέα λίστα της οποίας κεφαλή είναι η κεφαλή της πρώτης λίστας και ουρά είναι η συνένωση της ουράς της πρώτης λίστας με την δεύτερη.

Η υλοποίηση της παραπάνω περιγραφής είναι το ακόλουθο κατηγόρημα:

```
list_append([],List2,List2).
list_append([X|Tail],List2,[X|List3]):-
  list_append(Tail,List2,List3).
```

Οι ακόλουθες ερωτήσεις στην Prolog παράγουν τα αποτελέσματα που φαίνονται παρακάτω:

```
?- list_append([a,b,c,d],[e,f,g],L).
L = [a, b, c, d, e, f, g].
?- list_append([1,2,3],[4,5,6,7],L).
L = [1, 2, 3, 4, 5, 6, 7].
```

Αξίζει να σημειωθεί ότι η αναδρομή γίνεται στην πρώτη από τις δύο λίστες και όχι και στις δύο. Η δεύτερη λίστα παραμένει ανέπαφη σε όλη τη διάρκεια της αναδρομής. Ένα βίντεο που παρουσιάζει την εκτέλεση της append/3, μπορεί να βρεθεί εδώ.

Το κατηγόρημα μπορεί να χρησιμοποιηθεί και με τον ακόλουθο τύπο ερώτησης:

```
?- list_append([1,2,3],L2,[1,2,3,4,5]).
L2 = [4, 5].
```

Η τελευταία ερώτηση παρουσιάζει περισσότερο ενδιαφέρον, καθώς το ερώτημα που διατυπώνεται είναι "με ποια λίστα θα πρέπει να συνενωθεί η λίστα [1,2,3] για να δώσει σαν αποτέλεσμα τη λίστα [1,2,3,4,5];". Η προηγούμενη ερώτηση οδηγεί στη ακόλουθη, που είναι γενικότερη και ουσιαστικά αποτελεί την διατύπωση του ερωτήματος "Ποιων λιστών συνένωση είναι η λίστα [a, b, c, d];":

```
?- list_append(L1,L2,[a,b,c,d]).
L1 = [],
L2 = [a, b, c, d] ;


L1 = [a],
L2 = [b, c, d] ;

L1 = [a, b],
L2 = [c, d] ;

L1 = [a, b, c],
L2 = [d] ;

L1 = [a, b, c, d],
L2 = [] ;
false.
```

Κατά την οπισθοδρόμηση η Prolog επιστρέφει όλους τους πιθανούς συνδυασμούς λιστών, των οποίων η συνένωση δίνει τη λίστα [a,b,c,d].

Άρα το κατηγόρημα list_append/3 μπορεί να χρησιμοποιηθεί τόσο για τη συνένωση δύο λιστών, όσο και για την διάσπαση μιας λίστας σε κατάλληλες υπολίστες. Η δυνατότητα αυτή δίνει ιδιαίτερη ευελιξία στην υλοποίηση λύσεων για σύνθετα προβλήματα με πολύ απλό τρόπο, όπως θα δούμε σε επόμενη ενότητα.

Σημειώση: Σε όλες οι υλοποιήσεις της γλώσσας Prolog, το αντίστοιχο κατηγόρημα είναι το append/3.

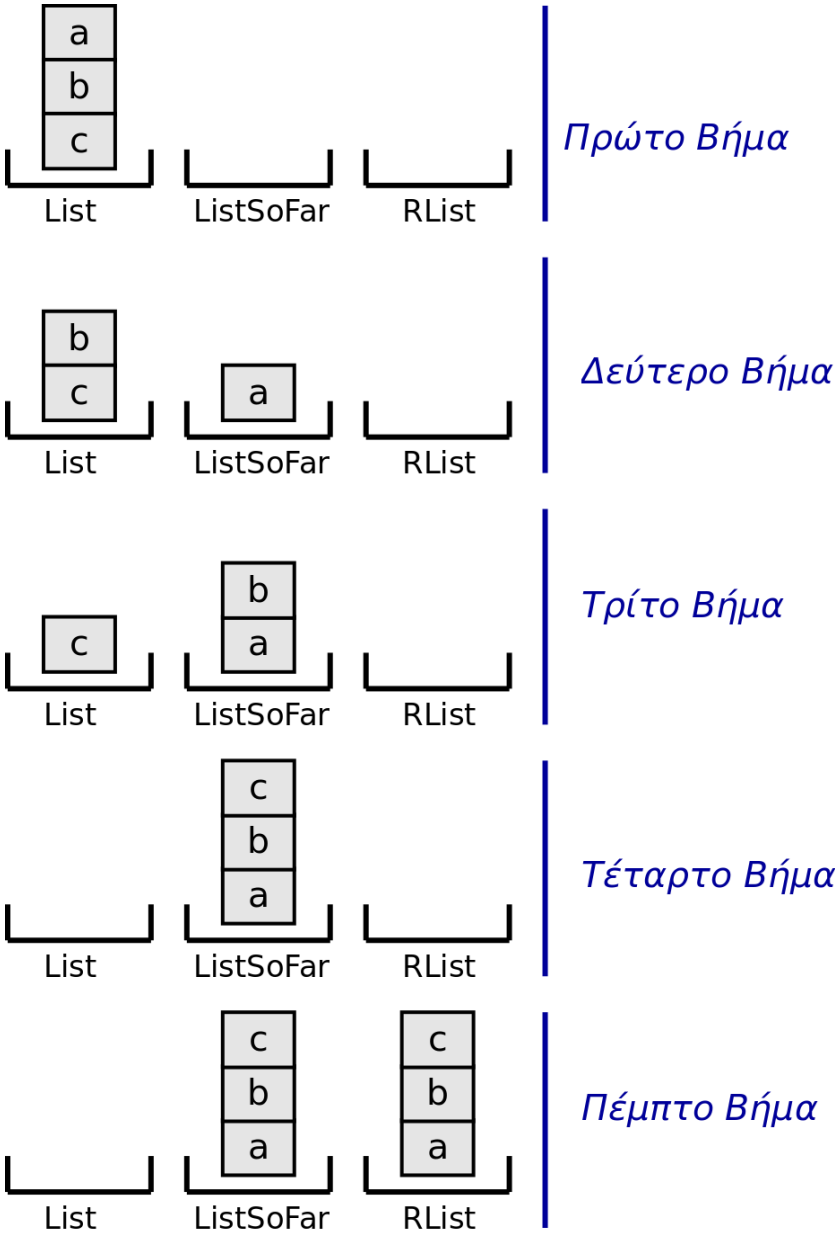
Αναστροφή Στοιχείων Λίστας

Η αναστροφή λίστα μιας λίστας περιέχει όλα τα στοιχεία της αρχικής λίστας αλλά με ανάστροφη σειρά. Για παράδειγμα, η ανάστροφη της λίστας [a,b,c] είναι η λίστα [c,b,a]. Προφανώς, το κατηγόρημα list_reverse το οποίο υλοποιεί το παραπάνω έχει δύο ορίσματα, τις δύο λίστες, και η συμπεριφορά του είναι η ακόλουθη:

```
?- list_reverse([a,b,c],RL).
RL = [c, b, a].
```

Μια προφανής λύση στην υλοποίηση του κατηγορήματος είναι να τοποθετούμε σε κάθε βήμα την κεφαλή της αρχικής λίστας στο τέλος της ανάστροφης, χρησιμοποιώντας το κατηγόρημα συνένωσης λιστών, που εξετάστηκε στην προηγούμενη παράγραφο. Όμως, μια τέτοια λύση δεν είναι αποδοτική ιδίως για μεγάλου μήκους λίστες, καθώς η συνένωση μιας λίστας με N στοιχεία με μια λίστα ενός στοιχείου (τοποθέτηση στο τέλος) απαιτεί N αναδρομικές κλήσεις. Η υλοποίησή αυτής της λύσης όμως είναι αρκετά απλή και αφήνεται σαν άσκηση.

Η τεχνική στην οποία θα βασιστεί η υλοποίηση που παρουσιάζεται παρακάτω, είναι παρόμοια με εκείνη της δεύτερης λύσης του κατηγορήματος max/2, βασίζεται δηλαδή στην κατασκευή της λύσης καθώς η εκτέλεση "κατεβαίνει" το αντίστοιχο κλαδί του δένδρου αναζήτησης. Η μερική (προσωρινή) λύση που δημιουργείται σε κάθε βήμα πρέπει να αποθηκεύεται σε μια θέση, και καθώς η Prolog είναι γλώσσα μοναδικής ανάθεσης, αυτό απαιτεί ένα ακόμη όρισμα. Στο Σχήμα 7.5 παρουσιάζονται τα βήματα δημιουργίας της ανάστροφης μιας λίστας List, χρησιμοποιώντας τη θέση ListSoFar για την αποθήκευση της προσωρινής ανάστροφης λίστας και τη θέση RList για την τελική ανάστροφη λίστα.



Σχήμα 7.5: *Τα βήματα κατασκευής της ανάστροφης μιας λίστας.*

Όπως φαίνεται στο σχήμα, σε κάθε βήμα η κεφαλή της λίστας List, τοποθετείται στην κορυφή της προσωρινής λύσης (ListSoFar). Η διαδικασία επαναλαμβάνεται, τοποθετώντας διαδοχικά τα στοιχεία της λίστας List στην λίστα ListSoFar με ανάστροφη σειρά. Όταν εξαντληθούν τα στοιχεία της List, τότε η ListSoFar είναι η ανάστροφη λίστα. Με βάση την παραπάνω ιδέα, η "διαδικαστική" περιγραφή του κατηγορήματος είναι:

- Όταν εξαντληθούν όλα τα στοιχεία της αρχικής λίστας (List), τότε η ανάστροφη λίστα (RList) είναι η προσωρινή λίστα που κατασκευάστηκε κατά την αναδρομή (ListSoFar).
- Αλλιώς, η ανάστροφη λίστα RList προκύπτει από την αναδρομική κλήση του κατηγορήματος στην ουράς της λίστας List και σε μια νέα λίστα, η οποία αποτελείται από την κεφαλή της List και την προσωρινή λίστα ListSoFar.

Η υλοποίηση της παραπάνω περιγραφής είναι η ακόλουθη:

```
list_reverse([], List, List).
list_reverse([H|T], ListSoFar, RList):-
```

```
  list_reverse(T,[H|ListSoFar], RList ).
```

Όπως και στο max/2, για να ικανοποιηθεί η αρχική προδιαγραφή του κατηγορήματος που ορίζει ότι πρέπει να υπάρχουν μόνο δύο ορίσματα, το κατηγόρημα list_reverse/3 καλείται από το κατηγόρημα list_reverse/2, όπως φαίνεται παρακάτω:

```
list_reverse( List, RList):-
  list_reverse( List, [], RList).
```

Η συμπεριφορά του κατηγορήματος είναι:

```
?- list_reverse([a,b,c],RL).
RL = [c, b, a].
?- list_reverse([a,b,c],[c,b,a]).
true.
?- list_reverse(R,[c,b,a]).
R = [a, b, c] .
```

Η τελευταία ερώτηση δείχνει ότι και στην περίπτωση αυτού του κατηγορήματος, δεν υφίσταται διάκριση εισόδου/εξόδου μεταξύ των ορισμάτων.

Σημειώση: Σε όλες οι υλοποιήσεις της γλώσσας Prolog, το αντίστοιχο κατηγόρημα είναι το reverse/3.

7.5 Παράδειγμα Εφαρμογής: Δυαδικοί αριθμοί.

Το παράδειγμα της λογικής σύζευξης των δυαδικών αριθμών, μπορεί να επεκταθεί εύκολα και σε άλλες πράξεις του συγκεκριμένου πεδίου. Θα εξετάσουμε στα επόμενα, μερικές από αυτές, ξεκινώντας με ένα απλό κατηγόρημα που εξασφαλίζει ότι ένας όρος είναι δυαδικός αριθμός.

Το κατηγόρημα binary_number/1, δέχεται ως όρισμα έναν όρο και επιτυγχάνει όταν ο όρος είναι λίστα, τα στοιχεία της οποίας είναι μηδέν ή ένα. Σύμφωνα με τα προηγούμενα, το κατηγόρημα θα διατρέχει αναδρομικά την λίστα, εξασφαλίζοντας ότι κάθε στοιχείο είναι 0 ή 1. Η υλοποίηση του παραπάνω είναι σχετικά απλή:

```
binary_number([X]):-
  zero_or_one(X).
binary_number([X|Rest]):-
  zero_or_one(X),
  binary_number(Rest).
```

Το παραπάνω κατηγόρημα, εξασφαλίζεται κάθε φορά ότι η μεταβλητή X είναι 0 ή 1, μέσω του κατηγορήματος zero_or_one/1. Το τελευταίο, υλοποιείται απλά σαν ένα ζεύγος, γεγονότων:

```
zero_or_one(0).
zero_or_one(1).
```

Η συμπεριφορά του κατηγορήματος είναι η ακόλουθη:

```
?- binary_number([1,0,0,1,1]).
true
?- binary_number([1,0]).
true .
?- binary_number([1,0,3]).
false.
```

Το κατηγόρημα μπορεί να κληθεί με μια μεταβλητή σαν όρισμα:

```
?- binary_number(Num).
Num = [0] ;
Num = [1] ;
Num = [0, 0] ;
Num = [0, 1] ;
Num = [0, 0, 0] ;
Num = [0, 0, 1] ;
Num = [0, 0, 0, 0] ;
Num = [0, 0, 0, 1] ;
```

```
Num = [0, 0, 0, 0, 0] ;
Num = [0, 0, 0, 0, 1] ;
Num = [0, 0, 0, 0, 0, 0] ;
...
```

Το σημείο αυτό χρήζει λίγο περισσότερης εξήγησης. Αν και θα περίμενε κάποιος να εξαντλούνται οι αριθμοί N-bits (για παράδειγμα 2 bits) πριν το κατηγόρημα επιστρέψει λύσεις με μεγαλύτερο αριθμό bits, επιστρέφονται όλο και μεγαλύτερου μήκους αριθμοί, οι οποίοι διαφέρουν μόνο στο τελευταίο ψηφίο. Αυτό οφείλεται στο γεγονός ότι ο μηχανισμός οπισθοδρόμησης, ερευνά συστηματικά το δένδρο αναζήτησης, ελέγχοντας εξαντλητικά, πρώτα τα αριστερότερα κλαδιά του δένδρου (αναζήτηση κατά βάθος). Καθώς, όμως είναι δυνατό να υπάρχουν (θεωρητικά) αριθμοί με άπειρα bits, το κατηγόρημα παράγει άπειρες λύσεις, επιστρέφοντας αριθμούς με όλο και μεγαλύτερο αριθμό ψηφίων.

Για να παράξει το κατηγόρημα όλους τους αριθμούς N bit, τότε θα έπρεπε πριν από την κλήση της binary_number/1, να οριστεί ότι ή λίστα είναι N στοιχείων. Για παράδειγμα η ακόλουθη ερώτηση επιστρέφει διαδοχικά όλους τους δυαδικούς μήκους 3 bit.

```
?- length(L,3),binary_number(L).
L = [0, 0, 0] ;
L = [0, 0, 1] ;
L = [0, 1, 0] ;
L = [0, 1, 1] ;
L = [1, 0, 0] ;
L = [1, 0, 1] ;
L = [1, 1, 0] ;
L = [1, 1, 1] ;
false.
```

Η πράξη της λογικής διάζευξης δυαδικών αριθμών (bitwise_or/3) είναι παρόμοια με εκείνη της λογικής σύζευξης, που παρουσιάστηκε στη αρχή του κεφαλαίου. Μοναδική διαφορά είναι ότι θα πρέπει να οριστεί κατάλληλα η λογική διάζευξη μέσω του κατηγορήματος bit_or/3, αντίστοιχο εκείνου του bit_and/3:

```
bit_or(0,0,0).
bit_or(0,1,1).
bit_or(1,0,1).
bit_or(1,1,1).
```

Ακολουθώντας μια παρόμοια δηλωτική περιγραφή, η υλοποίηση της λογικής σύζευξης είναι η ακόλουθη:

```
bitwise_or([Bit1], [Bit2], [BitR]):-
  bit_or(Bit1,Bit2,BitR).
bitwise_or([Bit1|RestBits1], [Bit2|RestBits2], [BitR|Result]):-
  bit_or(Bit1, Bit2, BitR),
  bitwise_or(RestBits1, RestBits2, Result).
```

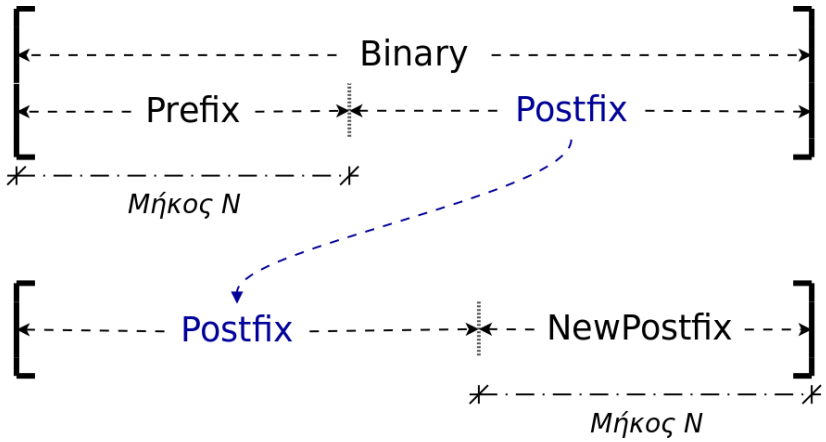
Η συμπεριφορά του κατηγορήματος είναι:

```
?- bitwise_or([1,1,1],[1,0,0],Res).
Res = [1, 1, 1] .
?- bitwise_or([0,0,0],[1,0,1],Res).
Res = [1, 0, 1] .
?- bitwise_or([0,1,0],[1,0,1],Res).
Res = [1, 1, 1] .
?- bitwise_or([0,2,0],[1,0,1],Res).
false.
```

Η τελευταία ερώτηση, δείχνει το πως η Prolog μπορεί να κάνει "έλεγχο τύπων", περιορίζοντας την επιτυχία της κλήσης του κατηγορήματος της πράξης της λογικής σύζευξης μόνο σε λίστες οι οποίες αποτελούν από 0 και 1. Προφανώς με παρόμοιο τρόπο μπορεί να οριστεί η πράξη της αποκλειστικής λογικής διάζευξης (XOR).

Ενδιαφέρον παρουσιάζουν οι πράξεις της λογικής ολίσθησης. Η λογική ολίσθηση N θέσεων προς τα αριστερά, αφορά την "μετατόπιση" των N ψηφίων του δυαδικού αριθμού προς τα αριστερά με ταυτόχρονη απόρριψη των N δεξιότερων ψηφίων και συμπλήρωση των κενών θέσεων που προκύπτουν με 0.

Αν το πρόβλημα διατυπωθεί σαν πρόβλημα λιστών, τότε από τη λίστα που αναπαριστά τον δυαδικό αριθμό (Binary) θα πρέπει να "σπάσει" σε δύο λίστες, μια αρχική υπολίστα (πρόθεμα) N στοιχείων (Prefix), και μια δεύτερη λίστα (Postfix) που θα μετατοπιστεί "προς τα αριστερά", όπως φαίνεται στο Σχήμα 7.6.



Σχήμα 7.6: *Η διαχείριση των λιστών στο κατηγόρημα shift_left/3.*

Η διάσπαση των δύο λιστών θα γίνει με το κατηγόρημα append/3:

```
append(Prefix,Postfix,Binary),
```

και θα εξασφαλιστεί ότι η λίστα Prefix έχει ακριβώς N στοιχεία:

```
length(Prefix,N),
```

Εφόσον η append/3 θα διασπάσει συστηματικά τη λίστα Binary στις υπολίστες που την απαρτίζουν, η κλήση της length/2 θα εξασφαλίσει ότι θα επιλεγεί εκείνη η λίστα η οποία έχει μήκος N. Η ιδέα της "έξυπνης" χρήσης της append/3 είναι αρκετά διαδεδομένη και ιδιαίτερα χρήσιμη, και θα εξεταστεί σε ένα πιο πολύπλοκο παράδειγμα στα επόμενα.

Η λίστα Postfix θα αποτελέσει το αριστερότερο μέρος του νέου δυαδικού αριθμού. Τα N δεξιότερα ψηφία του αριθμού θα είναι μηδενικά, άρα θα πρέπει να δημιουργηθεί μια λίστα (NewPostfix) μήκους N, της οποίας κάθε στοιχείο να είναι 0. Το πρώτο θα υλοποιηθεί απλά σαν μια κλήση στη length/2:

```
length(NewPostfix,N),
```

ενώ το δεύτερο απαιτεί τη δημιουργία ενός κατηγορήματος list_of(El,List), το οποίο εξασφαλίζει ότι η λίστα List αποτελείται από όρους El. Τροποποιώντας λίγο το κατηγόρημα is_list/1, που παρουσιάστηκε παραπάνω, η υλοποίηση γράφεται απλά:

```
list_of(_,[]).
list_of(El,[El|Rest]):-
  list_of(El,Rest).
```

Το παραπάνω, ουσιαστικά εξασφαλίζει ότι σε κάθε αναδρομική κλήση, η κεφαλή της λίστας είναι ενοποιήσιμη με το στοιχείο El που εμφανίζεται στο πρώτο όρισμα. Η συμπεριφορά του είναι:

```
?- list_of(0,L).
L = []
L = [0] ;
L = [0, 0] ;
L = [0, 0, 0] ;
L = [0, 0, 0, 0] ;
L = [0, 0, 0, 0, 0] ;
...
```

Η νέα λίστα NewPostfix θα πρέπει να προσαρτηθεί στο τέλος της λίστας Postfix, με μια απλή κλήση της append/3:

```
append(Postfix,NewPostfix,Result).
```

Δεδομένων των παραπάνω, το κατηγόρημα shift_left(N,Binary,Result), όπου Result είναι ο δυαδικός αριθμός που προκύπτει αν ο δυαδικός Binary ολισθήσει αριστερά κατά N θέσεις, δίνεται ως ακολούθως:

```
shift_left(N,Binary,Result):-
  append(Prefix,Postfix,Binary),
  length(Prefix,N),
  length(NewPostfix,N),
  list_of(0,NewPostfix),
  append(Postfix,NewPostfix,Result).
```

και η συμπεριφορά του είναι:

```
?- shift_left(2,[1,1,1,1],L).
L = [1, 1, 0, 0] .
?- shift_left(3,[1,0,1,1,0,0],L).
L = [1, 0, 0, 0, 0, 0] .
?- shift_left(1,[1,0,1,1,0,0],L).
L = [0, 1, 1, 0, 0, 0] .
```

Το τελευταίο κατηγόρημα που παρουσιάζει ενδιαφέρον λόγω της αναδρομικής τεχνικής που θα εφαρμοστεί είναι η ισοτιμία (parity) του δυαδικού αριθμού. Ένας δυαδικός αριθμός έχει άρτια ισοτιμία αν ο αριθμός των 1 είναι άρτιος (συμπεριλαμβανομένου του bit ελέγχου) και έχει περιττή ισοτιμία αν ο αριθμός των 1 είναι περιττός.

Η ιδέα στην οποία βασίζονται οι ορισμοί που ακολουθούν είναι ιδιαίτερα απλή. Αν γίνεται έλεγχος για άρτια ισοτιμία και το πρώτο στοιχείο της λίστας (κεφαλή) είναι 0 τότε η υπόλοιπη λίστα (ουρά) θα πρέπει να έχει άρτιο αριθμό 1, δηλαδή άρτια ισοτιμία. Αν η κεφαλή της λίστας είναι 1, τότε η ουρά θα πρέπει να έχει περιττό αριθμό 1, άρα και περιττή ισοτιμία. Αντίστοιχα ορίζεται και η περιττή ισοτιμία. Ο αναδρομικός ορισμός της άρτιας ισοτιμίας είναι ο ακόλουθος:

- Ο μονοψήφιος δυαδικός αριθμός 0 έχει άρτια ισοτιμία.
- Ένας δυαδικός αριθμός έχει άρτια ισοτιμία το αριστερότερο ψηφίο του είναι 0 και ο αριθμός που αποτελείται από τα υπόλοιπα ψηφία του έχει άρτια ισοτιμία.
- Ένας δυαδικός αριθμός έχει άρτια ισοτιμία το αριστερότερο ψηφίο του είναι 1 και ο αριθμός που αποτελείται από τα υπόλοιπα ψηφία του έχει περιττή ισοτιμία.

Εφόσον έχουμε εκφράσει τους δυαδικούς σαν λίστες από 0 και 1, ο παραπάνω ορισμός γίνεται:

- Η λίστα [0] έχει άρτια ισοτιμία.
- Μια λίστα έχει άρτια ισοτιμία αν η κεφαλή της είναι 0 και η ουρά της έχει άρτια ισοτιμία.
- Μια λίστα έχει άρτια ισοτιμία αν η κεφαλή της είναι 1 και η ουρά της έχει περιττή ισοτιμία.

Η υλοποίηση του κατηγορήματος even_parity/1 σε Prolog είναι η ακόλουθη:

```
even_parity([0]).
even_parity([0|Rest]):-
  even_parity(Rest).
even_parity([1|Rest]):-
  odd_parity(Rest).
```

Με παρόμοιο τρόπο ορίζεται και η περιττή ισοτιμία:

- Η λίστα [1] έχει περιττή ισοτιμία.
- Μια λίστα έχει περιττή ισοτιμία αν η κεφαλή της είναι 0 και η ουρά της έχει περιττή ισοτιμία.
- Μια λίστα έχει περιττή ισοτιμία αν η κεφαλή της είναι 1 και η ουρά της έχει άρτια ισοτιμία.

που οδηγεί στην υλοποίηση:

```
odd_parity([1]).
odd_parity([1|Rest]):-
  even_parity(Rest).
odd_parity([0|Rest]):-
  odd_parity(Rest).
```

Η συμπεριφορά του κατηγορήματος είναι η ακόλουθη:

```
?- even_parity([1,0,1,1]).
false.
?- even_parity([1,0,1]).
true .
?- odd_parity([1,0,1,0,1]).
true .
?- odd_parity([1,0,1,0,1,1]).
false.
```

Η συγκεκριμένη τεχνική δείχνει ότι η αναδρομή δεν είναι ανάγκη να περιορίζεται σε ένα ορισμό. Παρόμοιες τεχνικές εφαρμόζονται στην κατασκευή μεταγλωττιστών, και ειδικότερα στους μεταγλωττιστές αναδρομικής κατάβασης.

7.6 Παράδειγμα Εφαρμογής: Εύρεση κωδικού

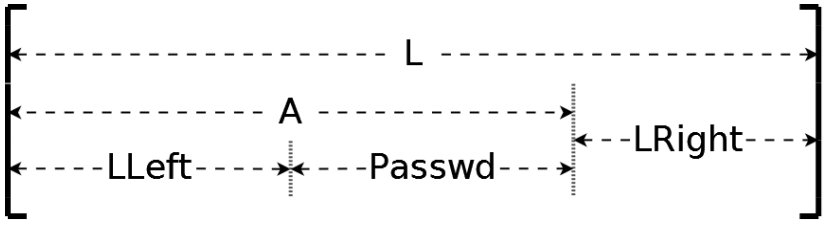
Ο ταμίας μίας τράπεζας ξέχασε τον κωδικό (password) που χρησιμοποιεί για τη διαχείριση των λογαριασμών και πρέπει να τον βοηθήσουμε να τον ανακτήσει γράφοντας ένα πρόγραμμα σε Prolog. Δίνονται οι παρακάτω βοηθητικές πληροφορίες που είχε κρατήσει ή θυμάται ο ταμίας:

- Τα ψηφία του κωδικού password είναι κριμένα κάπου μέσα σε μία μεγαλύτερη λίστα από ψηφία διαδοχικά το ένα μετά το άλλο (η λίστα αυτή μας δίνεται).
- Ο κωδικός (password) είναι ένας οκταψήφιος αριθμός.
- Τα ψηφία του κωδικού έχουν άθροισμα 42.
- Ο κωδικός αρχίζει από το ψηφίο 5 και τελειώνει στο ψηφίο 6.
- Ο κωδικός περιέχει τον αριθμό 3.

Παράδειγμα : Από τη λίστα [2, 3, 2, 6, 1, 5, 2, 3, 9, 8, 5, 4, 6, 4, 1, 3, 2, 8] προέκυψε το password 52398546 (είναι η υπολίστα: [5, 2, 3, 9, 8, 5, 4, 6]).

Στην εφαρμογή αυτή μπορούμε να δούμε τη χρησιμότητα των βασικών κατηγορημάτων χειρισμού λιστών που αναλύθηκαν παραπάνω (member/2, append/3 και length/2). Επίσης θα δούμε την "έξυπνη" χρήση του κατηγορήματος append/3, το οποίο, με βάση το μηχανισμό της ενοποίησης της Prolog, μπορεί να χρησιμοποιείται όχι μόνο για τη συνένωση δύο λιστών σε μία τρίτη, αλλά και να διαχωρίσει μία λίστα σε δύο μέρη (ή με την επαναληπτική εφαρμογή της και σε περισσότερα).

Ας ξεκινήσουμε αναζητώντας υπολίστες της αρχικής λίστας (που μας έδωσε ο ταμίας) που αποτελούν, πιθανά, τη λύση στο πρόβλημά μας. Όπως φαίνεται στο παρακάτω σχήμα (Σχήμα 7.7), εφόσον η υποψήφια υπολίστα Passwd βρίσκεται μέσα στην αρχική λίστα L, τότε υπάρχει ένα αρχικό τμήμα της λίστας, η LLeft που βρίσκεται αριστερά της Passwd και ένα τελικό τμήμα της, η LRight που βρίσκεται δεξιά της.



Σχήμα 7.7: *Υπολίστα μιας Λίστας. Στο σχήμα φαίνονται οι σχέσεις μεταξύ των μεταβλητών του κατηγορήματος*

Το παρακάτω κατηγόρημα επιστρέφει την Passwd ως υπολίστα της L:

```
sublist(L, Passwd) :-
  append(A, LRight, L),
  append(LLeft, Passwd, A).
```

Με τη χρήση της πρώτης append/3 χωρίζουμε τη λίστα σε δύο τμήματα, A και LRight ενώ με τη χρήση της δεύτερης χωρίζουμε την υπολίστα A που προέκυψε σε LLeft και Paswd.

Παρατήρηση (τεχνικής) 1: Θα μπορούσε να υποθέσει κανείς ότι η υπολίστα που αντιστοιχεί στο password βρίσκεται στην αρχή ή στο τέλος της λίστας που μας δίνεται αρχικά, π.χ. [*5, 2, 3, 9, 8, 5, 4, 6*, 4, 1, 3, 2, 8] ή [2, 3, 2, 6, 1, *5, 2, 3, 9, 8, 5, 4, 6*].

Τι γίνεται σε αυτή την περίπτωση; Η απάντηση είναι ότι δεν χρειάζεται να προσθέσουμε τίποτε επιπλέον στο κατηγόρημα sublist/2, καθώς η χρήση της πρώτης append/3 μας δίνει ως πιθανή λύση διαχωρισμού της αρχικής στην κενή λίστα και στον εαυτό της ή αντίστοιχα στον εαυτό της και την κενή λίστα, με τη χρήση της δεύτερης append/3.

Στη συνέχεια, το τελικό πρόγραμμά μας μπορεί να υλοποιηθεί με το παρακάτω κατηγόρημα, το οποίο από τις υπολίστες που δημιουργούνται από το sublist/2 επιλέγει εκείνη με τα επιθυμητά χαρακτηριστικά, δηλώνοντας τα χαρακτηριστικά που θα πρέπει να έχει η λίστα που αντιπροσωπεύει τον κωδικό, με κλήσεις αντίστοιχων κατηγορημάτων. Για παράδειγμα, η κλήση του κατηγορήματος length(Passwd, 8) εξασφαλίζει ότι η λύση (κωδικός) θα έχει 8 ψηφία, και η κλήση member(3, Passwd) ότι ανάμεσα σε αυτά τα 8 ψηφία θα περιλαμβάνεται το ψηφίο 3:

```
recover_passwd(L,Passwd):-
  sublist(L,Passwd),
  length(Passwd,8),
  sumlist(Passwd,42),
  append([5|Rest],[6],Passwd),
  member(3,Rest).
```

Το κατηγόρημα sumlist/2 που φαίνεται παραπάνω, ενοποιεί το δεύτερό του όρισμα στο άθροισμα των στοιχείων της λίστας, και αφήνεται ως άσκηση.

Παρατήρηση (τεχνικής) 2: Στο παραπάνω κατηγόρημα αξίζει να σημειωθεί η χρήση της append/3 (τελευταία γραμμή) η οποία ελέγχει αν η λίστα Passwd αρχίζει με 5 και τελειώνει σε 6. Η τοποθέτηση του ορίσματος [5| Rest] είναι δυνατή, καθώς ο μηχανισμός ενοποίησης της Prolog, σε αντίθεση με τις κλασσικές γλώσσες προγραμματισμού, επιτρέπει τη χρήση όχι μόνο μεταβλητών ή σταθερών αλλά και σύνθετων όρων με όποιο βαθμό λεπτομέρειας επιθυμεί ο προγραμματιστής.

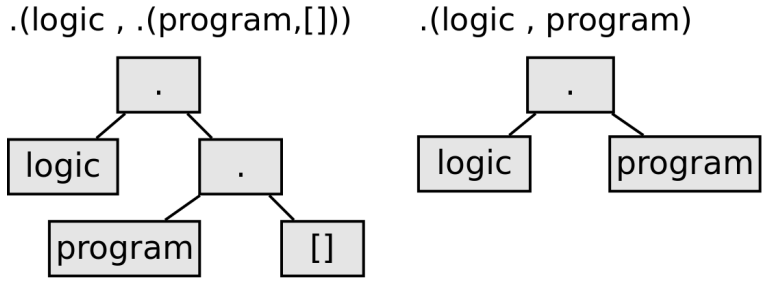
7.7 Προχωρημένα Θέματα

Έγκυρες, μη έγκυρες λίστες και ο κατασκευαστής “|”

Έστω οι ακόλουθες ερωτήσεις στο κατηγόρημα is_valid_list/1 που παρουσιάστηκε παραπάνω:

```
?- is_valid_list([program|[logic]]).
yes
?- is_valid_list([program|logic]).
no
```

Η απάντηση στην πρώτη είναι προφανής. Το δεύτερο παράδειγμα χρήζει ίσως μιας περισσότερο λεπτομερούς εξήγησης. Η σημειογραφία [H|T] είναι ισοδύναμη της .(H,T), όπως αναφέρθηκε παραπάνω. Έτσι ο όρος [program|[logic]] είναι ισοδύναμος με τον όρο .(program, .(logic, [])) ο οποίος αποτελεί έγκυρη λίστα, ενώ ο όρος [program|logic] είναι ισοδύναμος με τον όρο .(program , logic), ο οποίος δεν αποτελεί έγκυρη λίστα καθώς το δεύτερο όρισμά του δεν είναι λίστα. Αυτό καταδεικνύεται στο Σχήμα 7.8:



Σχήμα 7.8: *Έγκυρη και μη έγκυρη λίστα*

Η πλήρης υλοποίηση του μήκους μιας λίστας

Η υλοποίηση της list_length/2 που δόθηκε στην αντίστοιχη ενότητα δεν είναι πλήρης, καθώς δεν μπορεί να διαχειριστεί ερωτήσεις της μορφής:

```
?- list_length(L,3).
L = [_G97337, _G97340, _G97343] ;
< μη τερματισμός >
```

Ενώ η Prolog επιστέφει σωστά την πρώτη απάντηση, δηλαδή μια λίστα μεταβλητών μήκους 3, κατά την οπισθοδρόμηση "πέφτει" σε ατέρμονα βρόχο. Για να αντιληφθούμε το γιατί, ας εξετάσουμε τι επιστρέφει το κατηγόρημα list_length/2, όταν η ερώτηση τεθεί και με τα δύο ορίσματα ως μεταβλητές, οι απαντήσεις της Prolog είναι (ο συμβολισμός _G2060 κλπ, σημαίνει μεταβλητή):

```
?- list_length(List,Len).
List = [],
Len = 0 ;
List = [_G2060],
Len = 1 ;
List = [_G2060, _G2063],
Len = 2 ;
List = [_G2060, _G2063, _G2066],
Len = 3 ;
List = [_G2060, _G2063, _G2066, _G2069],
Len = 4 ;
...
```

Η συμπεριφορά είναι η αναμενόμενη από το κατηγόρημα, δηλαδή κατά την οπισθοδρόμηση επιστρέφονται διαδοχικά λίστες μεταβλητών μεγαλύτερου μεγέθους. Έτσι η ερώτηση list_lenth(L,3), γίνεται λόγω του πρώτου κανόνα του κατηγορήματος:

```
list_length(Tail, LenTail), 3 is LenTail + 1
```

όμως ο πρώτος υποστόχος της σύζευξης που απαρτίζει την ερώτηση επιστρέφει όπως είδαμε άπειρες λύσεις με αυξανόμενο συνεχώς μέγεθος. Δηλαδή η μεταβλητή LenTail θα είναι διαδοχικά 0, 1, 2, 3, … κοκ. Για την τιμή LenTail = 2 θα επιτύχει επιστρέφοντας την αντίστοιχη λύση, ενώ αν ζητήσουμε εναλλακτικές λύσεις, η τιμή της μεταβλητής θα αυξάνει συνεχώς και άρα ο δεύτερος υποστόχος (3 is LenTail + 1) θα αποτυγχάνει πάντα, πέφτοντας έτσι σε ατέρμονα βρόχο.

Για να επιλυθεί το πρόβλημα, απαιτείται να έχουμε δύο κανόνες στο κατηγόρημα, έναν που θα αφορά στην περίπτωση που το δεύτερο όρισμα είναι μεταβλητή και ο δεύτερος την περίπτωση που το όρισμα είναι αριθμός. Φυσικά, η τερματική συνθήκη παραμένει η ίδια:

```
list_length_b([],0).
```

Στον πρώτο κανόνα, χρησιμοποιούμε το ενσωματωμένο κατηγόρημα var/1 που παρουσιάστηκε στο Κεφάλαιο 4. Ο κανόνας θα είναι παρόμοιος με τον κανόνα του list_length/2, δηλαδή:

```
list_length_b([_|Tail],Len):-
  var(Len),
  list_length_b(Tail,LenTail),
  Len is LenTail + 1.
```

Η κλήση var(Len) εξασφαλίζει ότι στο σημείο εκείνο της εκτέλεσης και πριν την αναδρομική κλήση, η μεταβλητή Len δεν είχε πάρει ακόμη τιμή. Ο δεύτερος κανόνας έχει περισσότερο ενδιαφέρον, καθώς τώρα δεν "μετράμε" ανεβαίνοντας από την αναδρομή, αλλά μειώνουμε το δεύτερο όρισμα (μήκος της λίστας) κατά 1, πριν από κάθε αναδρομική κλήση της list_length_b/2. Αυτό είναι εφικτό, αν εξασφαλίσουμε ότι το δεύτερο όρισμα είναι αριθμός, με κλήση του κατηγορήματος number/1:

```
list_length_b([_|Tail],Len):-
  number(Len),
  Len > 0,
  LenTail is Len - 1,
  list_length_b(Tail,LenTail).
```

Για την αποφυγή του ατέρμονα βρόχου, πρέπει φυσικά να εξασφαλίσουμε ότι στον κανόνα το μήκος είναι μεγαλύτερο από 0, πριν μειώσουμε την τιμή του κατά 1, όπως παρουσιάστηκε στο κεφάλαιο της αναδρομής (Κεφάλαιο 6). Η συμπεριφορά του κατηγορήματος είναι πλέον η αναμενόμενη.

```
?- list_length_b(List,3).
List = [_G271273, _G271276, _G271279] ;
false.

?- list_length_b([a,v,c,f],3).
false.

?- list_length_b([a,b,c],L).
L = 3 ;
false.

?- list_length_b([a,b,c],3).
true.
```

Η list_length_b/2 επιτρέπει έτσι την δημιουργία λιστών, όταν είναι γνωστό το μήκος τους. Το κατηγόρημα length/2 σε όλες τις υλοποιήσεις της Prolog ακολουθεί αυτή την προσέγγιση.

7.8 Σημαντικές Παρατηρήσεις

Προσοχή στον έλεγχο των κατηγορημάτων

Κατά την ανάπτυξη αναδρομικών ορισμών γενικά και ιδιαίτερα στις λίστες, θα πρέπει τα κατηγορήματα να ελέγχονται για την ορθότητά τους, όχι μόνο για την πρώτη λύση που παράγουν, αλλά και για όλες τις εναλλακτικές (μέσω οπισθοδρόμησης). Είναι συχνό σφάλμα η πρώτη λύση να είναι ορθή, αλλά κατά την οπισθοδρόμηση, η εκτέλεση να επιστρέφει λάθος λύσεις, ή να πέφτει σε ατέρμονα βρόχο.

Οι λίστες δεν είναι απαραίτητο να περιέχουν στοιχεία του ιδίου "τύπου"

Όπως συζητήθηκε, τα στοιχεία μιας λίστας είναι όροι, άρα η ίδια λίστα μπορεί να περιέχει αριθμούς, σταθερές, σύνθετους όρους και άλλες λίστες. Για παράδειγμα τα ακόλουθα είναι έγκυες Prolog λίστες:

```
[1,2,m,3,4,m,5,6,7]
[a,b,name(john)]
[a,b,[34,cost],[price,22],byer(philip)]
```

Αυτό φαίνεται να έρχεται σε αντίθεση με εκείνο που συμβαίνει στις κλασσικές γλώσσες προγραμματισμού, όπου οι λίστες είναι συλλογές οντοτήτων του ιδίου τύπου (ακέραιοι, κινητής υποδιαστολής, κλπ). Εδώ θα πρέπει ο προγραμματιστής να "θυμάται" πάντα ότι η Prolog υποστηρίζει ένα και μοναδικό τύπο: τους όρους, άρα η Prolog είναι συμβατή με την "αρχή" που ισχύει στις περισσότερες γλώσσες προγραμματισμού.

Μη πλήρως ορισμένοι όροι (λίστες).

Όπως είδαμε στο παράδειγμα της εύρεσης κωδικού, είναι δυνατό να δημιουργηθούν λίστες, μερικώς ορισμένες, δηλαδή στη γενική περίπτωση η χρήση όρων όπου δεν είναι πλήρως γνωστοί. Για παράδειγμα, έστω μια λίστα ακεραίων της μορφής:

```
[1,2,m,3,4,m,5,6,7]
```

η οποία αποτελείται από τρεις υπολίστες ακεραίων αριθμών που διαχωρίζονται από το σύμβολο m. Έστω ότι θα θέλαμε να "ανακτήσουμε" εκείνη την υπολίστα R που περιέχει 3 στοιχεία. Μια λύση είναι η ακόλουθη σύνθετη ερώτηση:

```
?- append(X,[m|R],[1,2,m,3,4,m,5,6,7]), length(R,3).
X = [1, 2, m, 3, 4],
R = [5, 6, 7] ;
```

δηλαδή πρώτα να σπάσουμε την λίστα σε δύο υπολίστες, όπου η λίστα R να έπεται του συμβόλου m και έπειτα να ελέγξουμε το μήκος της. Όμως η ακόλουθη ερώτηση δίνει το ίδιο αποτέλεσμα:

```
?- length(R,3),append(X,[m|R],[1,2,m,3,4,m,5,6,7]).
R = [5, 6, 7],
X = [1, 2, m, 3, 4] ;
```

Στην περίπτωση αυτή πρώτα δημιουργήθηκε μια λίστα μήκους 3, με στοιχεία τρεις ελεύθερες μεταβλητές και έπειτα η append/3 "έσπασε" κατάλληλα την αρχική λίστα.

Βιβλιογραφία

Οι λίστες αποτελούν αντικείμενο όλων των "κλασικών" βιβλίων αναφοράς για τη γλώσσα προγραμματισμού Prolog, όπως τα (Clocksin and Mellish, 2003), (Bratko, 2011), (Sterling and Shapiro, 1994) (O'Keefe, 1990) και (Covington et al., 1988).

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Clocksin, W. F. and Mellish, C. S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

Covington, M. A. and Nute, D. and Vellino A. (1988). *Prolog Programming in Depth*. Scott Foresman & Co. ISBN 9780673186591.

O'Keefe, R. A. (1990), *The craft of Prolog*, Cambridge, Mass: MIT Press

Sterling, L. and Shapiro, E. Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass: MIT Press.

Άλυτες Ασκήσεις

7.1 Υλοποιείστε ένα κατηγόρημα που προσθέτει τα στοιχεία μιας λίστας.

```
?-sumoflist([2,3,7,10,4],X).
X=26
```

7.2 Υλοποιείστε ένα κατηγόρημα που βρίσκει το πόσες φορές εμφανίζεται ένα συγκεκριμένο στοιχείο σε μια λίστα:

```
?- occurs(5,[2,3,4,5,3,6,5,3,5,3,8],X).
X=3
```

7.3 Υλοποιείστε ένα κατηγόρημα που βρίσκει πόσες φορές εμφανίζονται περιττοί σε μια λίστα:

```
?-count_odd([2,3,1,2,2,6,8,5],X).
X=3
```

Σημείωση: Ένας αριθμός N είναι περιττός αν δεν διαιρείται με το 2 ...,1 is N mod 2, ...

7.4 Υλοποιείστε ένα κατηγόρημα που υπολογίζει πόσα φωνήεντα υπάρχουν σε μια λίστα:

```
?- count_vowels([p,r,o,l,o,g],X).
```

```
X=2
```

7.5 Ορίστε ένα Prolog κατηγόρημα sum_even/2 το οποίο πετυχαίνει όταν το δεύτερο του όρισμα είναι το άθροισμα όλων των άρτιων (ζυγών) αριθμών που εμφανίζονται στην λίστα ακεραίων του πρώτου ορίσματος. Για παράδειγμα:

```
?- sum_even([2,4,1],L).
L = 6
Yes
?- sum_even([2,3,4,1,5,4],L).
L = 10
Yes
?- sum_even([1,3,5,7,9],L).
L = 0
Yes
```

Σημείωση: Ο έλεγχος για το αν ο αριθμός X είναι άρτιος γίνεται με την κλήση 0 is X mod 2.

7.6 Ορίστε ένα κατηγόρημα replace/4 (replace(X, Y, List, ResultList)) το οποίο πετυχαίνει όταν η λίστα ResultList είναι η λίστα List, όπου μια εμφάνιση του στοιχείου X έχει αντικατασταθεί με το στοιχείο Y. Στην οπισθοδρόμηση το κατηγόρημα θα πρέπει να αντικαθιστά την επόμενη εμφάνιση του στοιχείου X, κλπ. Για παράδειγμα:

```
?- replace(1,a,[1,2,3],R).
R = [a, 2, 3]
Yes
?- replace(1,a,[1,2,1,1,3],R).
R = [a, 2, 1, 1, 3] ;
R = [1, 2, a, 1, 3] ;
R = [1, 2, 1, a, 3] ;
No
```

7.7 Γράψτε ένα κατηγόρημα Prolog που πετυχαίνει αν μια λίστα αποτελείται από 2 ίδια μέρη:

```
?- symmetric([1,2,3,4,1,2,3,4]).
yes.
```

7.8. Αναπτύξτε ένα κατηγόρημα το οποίο ελέγχει αν το πρώτο όρισμα (λίστα) είναι επίθεμα (suffix) του δεύτερου ορίσματος (λίστα) :

```
?-end_sublist([e,r,t],[s,f,j,u,e,r,t]).
yes.
```

7.9 Αναπτύξτε ένα κατηγόρημα το οποίο πετυχαίνει όταν η λίστα του πρώτου ορίσματος εμφανίζεται δύο φορές μέσα στη λίστα του δεύτερου ορίσματος:

```
?- twice_sublist([3,4],[1,2,3,4,5,3,4,6]).
yes.
```

7.10 Υλοποιείστε ένα Prolog κατηγόρημα που βρίσκει το τελευταίο στοιχείο μιας λίστας, χρησιμοποιώντας το ενσωματωμένο κατηγόρημα append/3:

```
?- last_element([3,4,5,1,6],X).
X=6
```

7.11 Υλοποιείστε ένα κατηγόρημα που βρίσκει το γράμμα το οποίο λείπει από μια λέξη (ορθογραφικό λάθος). Για παράδειγμα:

```
?- missing_letter([p,r,l,o,g],X,W).
X=o
W = [p,r,o,l,o,g]
```

Υποθέστε ότι έχετε μια βάση με όλες τις λέξεις της Αγγλικής:

```
word([p,r,o,l,o,g]).
word([m,a,t,h,s]).
...
```

7.12 Να ορίσετε το κατηγόρημα reverse_alt/2 {reverse_alt(List1, Lis2)} το οποίο πετυχαίνει όταν η List2 είναι η ανάστροφη λίστα της List1, χρησιμοποιώντας το ενσωματωμένο κατηγόρημα append/3.

7.13 Χρησιμοποιώντας το ενσωματωμένο κατηγόρημα append/3 ορίστε ένα κατηγόρημα rotate_left(Pos,List,RotatedList), το οποίο πετυχαίνει όταν το όρισμα RotatedList είναι η λίστα List που έχει υποστεί αριστερή περιστροφή κατά Pos θέσεις. Για παράδειγμα:

```
?- rotate_left(1,[1,2,3,4,5],L).
L = [2, 3, 4, 5, 1]
Yes
?- rotate_left(3,[1,2,3,4,5],L).
L = [4, 5, 1, 2, 3]
Yes
```

ΚΕΦΑΛΑΙΟ 8: Ευφυείς Τεχνικές Λογικού Προγραμματισμού

Λέξεις Κλειδιά
Αντίστροφη χρήση κατηγορημάτων, Ιντετερμινιστικός Προγραμματισμός, Συμβολικός Προγραμματισμός, Επαυξητικός Προγραμματισμός, "Από άνω προς τα κάτω" ανάπτυξη προγραμμάτων

Περίληψη
Στο κεφάλαιο αυτό θα ασχοληθούμε με ένα σύνολο τεχνικών που διαφοροποιούν το λογικό προγραμματισμό, από τον προγραμματισμό με τη βοήθεια των κλασσικών διατακτικών γλωσσών προγραμματισμού. Οι τεχνικές αυτές αναφέρονται (α) στη δυνατότητα χρήσης κατηγορημάτων με αντίστροφη λογική που προκύπτει ως συνέπεια της δηλωτικότητας του Λογικού Προγραμματισμού (ΛΠ) και της απουσίας προκαθορισμένων ορισμάτων εισόδου-εξόδου, (β) στον μη-αιτιοκρατικό ή ιντετερμινιστικό προγραμματισμό (nondeterministic programming) και πώς μια τέτοια προσέγγιση υποστηρίζεται εγγενώς από το μοντέλο του Λογικού Προγραμματισμού και της Prolog, και (γ) στο συμβολικό προγραμματισμό κυρίως μέσω της δυνατότητας αναπαράστασης και επίλυσης προβλημάτων με ελλιπή δεδομένα. Οι τεχνικές αυτές είναι ιδιαίτερα χρήσιμες για την αντιμετώπιση προβλημάτων της Τεχνητής Νοημοσύνης και υπό αυτήν την έννοια χαρακτηρίζονται ως ευφυείς. Επίσης γίνεται αναφορά στην τεχνική του Αυξητικού (Incremental) Προγραμματισμού, ιδιαίτερα δημοφιλή στο Λογικό Προγραμματισμό καθώς οι περισσότερες υλοποιήσεις είναι διερμηνευόμενες (interpreted) και στην "από άνω προς τα κάτω" (Top Down) προσέγγιση στην ανάπτυξη προγραμμάτων.

Μαθησιακοί Στόχοι

Ο στόχος αυτού του κεφαλαίου είναι διπλός: Από τη μία πλευρά δημιουργείται στον προγραμματιστή η "κουλτούρα" του εναλλακτικού τρόπου σκέψης στη διαδικασία επίλυσης προβλημάτων, δίνοντας του έτσι τη δυνατότητα να αντιμετωπίζει δύσκολα προβλήματα αναζητώντας συνεχώς νέες τεχνικές. Από την άλλη πλευρά, η εξοικείωση με τις παρουσιαζόμενες τεχνικές του ΛΠ δίνει τη δυνατότητα στον προγραμματιστή να αντιμετωπίζει τα προγραμματιστικά προβλήματα σε ένα υψηλότερο επίπεδο αφαίρεσης και να μεταφέρει αυτήν την δυνατότητα στον προγραμματισμό σύνθετων συστημάτων με οποιαδήποτε γλώσσα ή εργαλείο προγραμματισμού χρησιμοποιεί ή θα χρησιμοποιήσει στο μέλλον.

Παράδειγμα κίνητρο

Το κεφάλαιο αυτό δεν περιλαμβάνει ένα κεντρικό παράδειγμα αναφοράς και ο αναγνώστης καλείται να μελετήσει τα αντίστοιχα παραδείγματα των αντίστοιχων ενοτήτων.

Μονοπάτι Μάθησης

Οι ενότητες αυτού του κεφαλαίου αναπτύσσονται στις παραγράφους 8.1, 8.2, 8.3 και 8.4. Οι τρεις πρώτες ενότητες αναφέρονται σε έξυπνες τεχνικές που μπορούν να υλοποιηθούν με την Prolog και καλό είναι να διαβαστούν με τη σειρά που εμφανίζονται. Η τέταρτη ενότητα αναφέρεται σε γενικότερα ζητήματα μεθοδολογίας και μπορεί να διαβαστεί ανεξάρτητα και με οποιαδήποτε σειρά, σε σχέση με τις υπόλοιπες ενότητες.

8.1 Αντίστροφη χρήση κατηγορημάτων

Τα κατηγορήματα, όταν τα αντιμετωπίσει κανείς με την ευρύτερη λογική των γλωσσών προγραμματισμού, ορίζουν στην ουσία υποπρογράμματα της Prolog (το σύνολο των φράσεων που ορίζουν ένα κατηγόρημα αποτελεί ένα υποπρόγραμμα). Θα μπορούσε να θεωρηθεί, παραδείγματος χάριν, ότι μοιάζουν με boolean μεθόδους ή συναρτήσεις που χρησιμοποιούνται στη Java και στη C/C++ αντίστοιχα για να οριστούν τα υποπρογράμματα. Τα κατηγορήματα όμως αποτελούν πολύ ισχυρότερες προγραμματιστικές οντότητες από τα αντίστοιχα υποπρογράμματα των άλλων γλωσσών, κυρίως λόγω του μηχανισμού της ενοποίησης που

αποτελεί και τον μηχανισμό περάσματος παραμέτρων, όταν καλείται/εκτελείται ένα κατηγόρημα. Δύο είναι τα βασικά χαρακτηριστικά της ενοποίησης που προσάπτουν ιδιαίτερη ευελιξία στη χρήση των όρων που χρησιμοποιούνται ως παράμετροι ενός κατηγορήματος:

(α) Λόγω της αμφίδρομης λογικής της ενοποίησης δύο όρων, οι παράμετροι ενός κατηγορήματος δεν ορίζονται κατ΄αποκλειστικότητα είτε ως παράμετροι εισόδου είτε ως παράμετροι εξόδου, αλλά μπορούν να έχουν εναλλακτικά και τις δύο ιδιότητες ανάλογα με τα δεδομένα κατά τον χρόνο εκτέλεσης (at run-time). Το χαρακτηριστικό αυτό των παραμέτρων το έχουμε ήδη δει σε προηγούμενα κεφάλαια: Ας θυμηθούμε για παράδειγμα τους διαφορετικούς τρόπους εκτέλεσης του κατηγορήματος follows/2 του κεφαλαίου 4:

```
?- follows(ilias,petros).
?- follows(ilias,X).
?- follows(X,petros).
?- follows(X,Y).
```

Στην πρώτη περίπτωση και οι δύο παράμετροι του κατηγορήματος λειτουργούν ως παράμετροι εισόδου. Στην δεύτερη περίπτωση η πρώτη παράμετρος είναι εισόδου και η δεύτερη εξόδου, ενώ το αντίστροφο συμβαίνει στην τρίτη περίπτωση. Στην τέταρτη περίπτωση και οι δύο παράμετροι είναι εξόδου.

(β) Η παράμετρος ενός κατηγορήματος μπορεί να είναι ένας σύνθετος όρος και με την έννοια αυτή μία παράμετρος μπορεί να περιγράφει τη δομή δεδομένων στην οποία αντιστοιχεί με όση λεπτομέρεια επιθυμεί ο προγραμματιστής. Και το χαρακτηριστικό αυτό το έχουμε δει σε προηγούμενα κεφάλαια. Ας δούμε για παράδειγμα έναν από τους κανόνες που χρησιμοποιήθηκε στην ενότητα 7.3 του Κεφαλαίου 7 για τον ορισμό του κατηγορήματος που βρίσκει το μέγιστο στοιχείο μιας λίστας:

```
list_max_c([H1,H2|T],Max):-
  H1 > H2,
  list_max_c([H1|T],Max).
```

Στο σώμα του κανόνα, η πρώτη παράμετρος [H1|T] αναπαριστά την δομή δεδομένων λίστα, περιγράφοντας ρητά ότι το πρώτο στοιχείο της (η κεφαλή) είναι το H1 και τα υπόλοιπα στοιχεία βρίσκονται στην ουρά της λίστας T, ενώ στο αριστερό μέλος του κανόνα η πρώτη παράμετρος [H1,H2|T] αναπαριστά τη δομή δεδομένων με μεγαλύτερη λεπτομέρεια: το πρώτο στοιχείο της λίστας είναι το H1, το δεύτερο το H2 και τα υπόλοιπα στοιχεία αποτελούν τη λίστα T.

Στην συνέχεια θα δούμε πώς μπορούμε να εκμεταλλευτούμε τα παραπάνω χαρακτηριστικά των κατηγορημάτων ως μία ευφυή τεχνική προγραμματισμού. Ας ξεκινήσουμε με μερικά απλά παραδείγματα:

Εισαγωγή Στοιχείου σε μία Λίστα

Θέλουμε να ορίσουμε το κατηγόρημα list_insert/3, το οποίο εισάγει ένα στοιχείο σε μία τυχαία θέση μιας λίστας, επιστρέφοντας μία νέα λίστα που περιέχει και το στοιχείο αυτό.

Έχουμε ήδη υλοποιήσει το κατηγόρημα list_delete/3 (βλέπε παράγραφο 7.4) το οποίο διαγράφει ένα στοιχείο (1η παράμετρος) από μία λίστα (2η παράμετρος) και επιστρέφει τη λίστα χωρίς αυτό το στοιχείο (3η παράμετρος). Λειτουργώντας "αντίστροφα", αν θεωρήσουμε άγνωστη τη δεύτερη παράμετρο και την τρίτη παράμετρο γνωστή (τη λίστα αποτέλεσμα), έχουμε:

```
?- list_delete(a,L,[b,c,d]).
L = [a,b,c,d] ;
L = [b,a,c,d] ;
L = [b,c,a,d] ;
L = [b,c,d,a] ;
```

Βασιζόμενοι στην αντίστροφη χρήση των ορισμάτων μπορούμε πολύ απλά να ορίσουμε το list_insert/3:

```
list_insert(X,L1,L2):-
  list_delete(X,L2,L1).
```

Θα πάρουμε φυσικά τα ίδια αποτελέσματα:

```
?- list_insert(a,[b,c,d],L).
L = [a,b,c,d] ;
L = [b,a,c,d] ;
L = [b,c,a,d] ;
```

```
L = [b,c,d,a] ;
```

Το κατηγόρημα list_insert/3, λοιπόν χρησιμοποιώντας "έξυπνα" το κατηγόρημα list_delete/3 δεν είναι τίποτε άλλο παρά ένας καλύτερος τρόπος διασύνδεσης με το χρήστη που επιθυμεί να εισάγει ένα στοιχείο σε μία λίστα.

Εναλλακτική υλοποίηση του μέλους μιας λίστας

Ας θεωρήσουμε τώρα ότι έχουμε υλοποιήσει το κατηγόρημα list_append/3 και επιθυμούμε να ορίσουμε το κατηγόρημα ελέγχου αν ένα στοιχείο αποτελεί μέλος μιας λίστας. Έχουμε:

```
list_member2(X,L):-
  list_append(_,[X|_],L).
```

Στην περίπτωση αυτή εκμεταλλευόμαστε τη δυνατότητα αντίστροφης χρήσης της list_append/3 καθώς και της δυνατότητας περιγραφής της δεύτερης παραμέτρου με τον αναλυτικό τρόπο κεφαλή-ουρά. Σκεφτόμαστε ως εξής: είναι λογικό εάν ένα στοιχείο X είναι μέλος μιας λίστας L, τότε αυτό θα βρίσκεται σε κάποιο σημείο της και θα τη χωρίζει σε δύο υπο-λίστες, μία με τα στοιχεία πριν από αυτό και μία με τα στοιχεία που το ακολουθούν. Στο σώμα του παραπάνω κανόνα, θεωρούμε γνωστό το τρίτο όρισμα και μέρος του δεύτερου ορίσματος του κατηγορήματος list_append/3. Τα στοιχεία της λίστας που βρίσκονται πριν και μετά από αυτό δεν μας ενδιαφέρουν και για το λόγο αυτό έχουμε χρησιμοποιήσει ανώνυμες μεταβλητές.

Η έξυπνη επαναχρησιμοποίηση κατηγορημάτων που έχουμε ήδη υλοποιήσει μας δίνει τη δυνατότητα να υλοποιήσουμε γρήγορα και άλλα κατηγορήματα αρκεί να βρούμε πώς αυτά σχετίζονται λογικά μεταξύ τους (τουλάχιστον στην περίπτωση που δεν μας ενδιαφέρει ιδιαίτερα ο χρόνος εκτέλεσης του νέου κατηγορήματος). Ας δούμε μία εναλλακτική υλοποίηση του μέλους λίστας με τη βοήθεια της διαγραφής στοιχείου από λίστα. Έχουμε:

```
list_member3(X,L):-
  list_delete(X,L,_).
```

Στην περίπτωση αυτή σκεφτόμαστε πολύ απλά (και λογικά) ότι εάν μπορέσουμε να διαγράψουμε ένα στοιχείο από μία λίστα τότε αυτό συνεπάγεται ότι υπήρχε στη λίστα. Το τρίτο όρισμα της προϋπόθεσης του παραπάνω κανόνα (η λίστα που προκύπτει μετά τη διαγραφή) δεν μας ενδιαφέρει και για το λόγο αυτό χρησιμοποιούμε την ανώνυμη μεταβλητή.

Εναλλακτική υλοποίηση της διαγραφής στοιχείου λίστας

Στην περίπτωση αυτή θεωρούμε εκ νέου ότι έχουμε στη διάθεσή μας το κατηγόρημα list_append/3 και επιθυμούμε να ορίσουμε το κατηγόρημα διαγραφής στοιχείου από μία λίστα. Έχουμε:

```
list_delete2(X,L1,L2):-
  list_append(A,[X|B],L1),
  list_append(A,B,L2).
```

Σκεφτόμαστε ως εξής (πολύ απλά και λογικά): Το στοιχείο της λίστας X που θέλουμε να διαγράψουμε από τη λίστα L1, εφόσον υπάρχει, τη χωρίζει σε δύο υπολίστες A και B που περιλαμβάνουν τα στοιχεία πριν και μετά από αυτό αντίστοιχα. Αν τις συνενώσουμε έχουμε τη λίστα L2 που προκύπτει ως αποτέλεσμα της διαγραφής. Στην πρώτη προϋπόθεση (υποστόχο) του παραπάνω κανόνα θεωρούμε γνωστά το τρίτο όρισμα και την κεφαλή της λίστας του δεύτερου ορίσματος και αναζητούμε τις υπολίστες A και B, τις οποίες χρησιμοποιούμε ως γνωστές στον δεύτερο υποστόχο για να δημιουργήσουμε τη λίστα αποτέλεσμα του τρίτου ορίσματος.

Ένα πιο σύνθετο παράδειγμα: Έλεγχος ορθογραφίας

Μια ιδιαίτερα χρήσιμη εφαρμογή είναι η διόρθωση ορθογραφικών λαθών στις λέξεις ενός κειμένου. Βασικό συστατικό ενός ορθογράφου που συνοδεύει έναν επεξεργαστή κειμένου είναι η ύπαρξη κάποιου λεξικού, δηλαδή κάποιας βάσης δεδομένων που περιέχει τις αποδεκτές λέξεις. Μία λέξη ενός κειμένου θεωρείται λάθος αν δεν μπορεί να βρεθεί στο λεξικό. Η διόρθωση της λάθος λέξης βασίζεται σε μία διαδικασία αλλαγών της, μέχρις ότου αυτή να μπορεί να βρεθεί στο λεξικό. Στη διαδικασία αυτή των αλλαγών μπορεί να περιλαμβάνονται ενδεικτικά: η διαγραφή ενός γράμματος, η προσθήκη ενός γράμματος, η αντιμετάθεση δύο γραμμάτων, η μετακίνηση γράμματος σε άλλο σημείο της λέξης ή και ο συνδυασμός δύο ή περισσότερων

αλλαγών. Αν η λέξη αντιμετωπιστεί ως μία συμβολοσειρά (string) και η διαδικασία των αλλαγών ως μία εξαντλητική επεξεργασία των χαρακτήρων της, το πρόβλημα καθίσταται πολύπλοκο και η επίλυσή του εξαιρετικά χρονοβόρα. Συνήθως λοιπόν καθορίζονται εμπειρικοί κανόνες, οι οποίοι περιγράφουν τους τύπους λαθών, με προτεραιότητα από τους πιο συνηθισμένους στους λιγότερο και οι αλλαγές πραγματοποιούνται με βάση αυτούς τους κανόνες. Σημειώνοντας ότι η αναλυτική παρουσίαση ενός ολοκληρωμένου ορθογράφου ξεφεύγει από τα όρια αυτού του παραδείγματος, ενδεικτικά αναφέρουμε μερικούς κανόνες:

- Προσπάθησε να αλλάξεις ένα μόνο γράμμα της λέξης και δες αν η νέα που προκύπτει είναι στο λεξικό.
- Αντιμετάθεσε δύο διπλανά γράμματα της λέξης και δες αν η νέα που προκύπτει είναι στο λεξικό.
- Σβήσε ένα γράμμα της λέξης και δες αν η νέα που προκύπτει είναι στο λεξικό.
- κ.ο.κ.

Θα δούμε στη συνέχεια ότι στην Prolog οι κανόνες διόρθωσης ενός ορθογράφου υλοποιούνται πολύ εύκολα με τη χρήση του κατηγορήματος append/3, και του γεγονότος ότι ο **μηχανισμός ενοποίησης (unification)** δίνει τη δυνατότητα να καθορίζονται τα ορίσματα των κατηγορημάτων άλλοτε ως εισόδου και άλλοτε ως εξόδου.

Για απλούστευση, θα θεωρήσουμε ότι οι λέξεις βρίσκονται σε μορφή λίστας από γράμματα. Για παράδειγμα:

```
word([h,e,l,l,o]).
word([m,a,n]).
word([m,e,n]).
word([w,o,m,a,n]).
word([b,o,o,k]).
word([m,o,t,h,e,r]).
word([c,o,m,p,u,t,e,r]).
word([c,a,r]).
word([p,r,o,l,o,g]).
word([p,e,n,c,i,l]).
word([t,a,b,l,e]).
```

Τα παραπάνω γεγονότα δηλώνουν τη λίστα των αποδεκτών λέξεων. Φυσικά σε ένα πραγματικό ορθογράφο η παραπάνω λίστα θα προκύπτει από την ανάγνωση ενός αρχείου λεξικού, θα είναι πολύ μεγαλύτερη και πληρέστερη, καλύπτοντας - αν αυτό είναι δυνατό - όλες τις λέξεις της γλώσσας που μας ενδιαφέρει. Η τυχαία σειρά με την οποία δηλώνονται εδώ οι λέξεις δεν έχει φυσικά στην Prolog καμία σημασία.

Μια απλή ορθογραφική διόρθωση είναι η ανίχνευση μιας λέξης που γράφτηκε με ένα γράμμα λάθος. Για παράδειγμα θα μπορούσαμε να ορίσουμε το κατηγόρημα

```
% changeletter(Word,CorrectWord)
```

το οποίο δέχεται ως πρώτο όρισμα μια λέξη γραμμένη με ένα γράμμα λάθος και επιστρέφει στο δεύτερο όρισμα τη λέξη (και κατά την οπισθοδρόμηση τις λέξεις) του λεξικού που ταιριάζουν με αυτήν αν αλλάξουμε το λάθος γράμμα. Για παράδειγμα, αναμένουμε τα εξής αποτελέσματα κατά τη διάρκεια της εκτέλεσης:

```
?- changeletter([m,o,n],X).
X = [m,a,n];
X = [m,e,n];
```

Οι λέξεις "mon" και "man" διαφέρουν σε ένα μόνο γράμμα, και η λέξη "man" είναι μια λέξη που βρίσκεται στο λεξικό. Εναλλακτικά το ίδιο ισχύει και για τις λέξεις "mon" και "men".

Το κατηγόρημα changeletter υλοποιείται ως εξής:

```
changeletter(Word,CorrectWord) :-
  append(Begin,[Letter|End],Word),
  word(CorrectWord),
  append(Begin,[NewLetter|End],CorrectWord),
  NewLetter\= Letter.
```

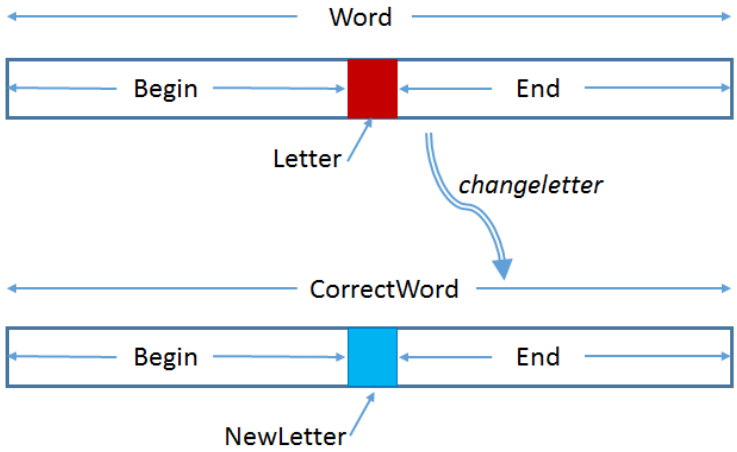
Η κλήση του κατηγορήματος word(CorrectWord) απαιτεί η λίστα CorrectWord, που αναζητούμε ως αποτέλεσμα, να ανήκει στη λίστα των αποδεκτών λέξεων του λεξικού.

Η πρώτη κλήση του κατηγορήματος append/3 πριν την κλήση της word(CorrectWord) εκμεταλλεύεται την ιδιότητα της αντίστροφης κλήσης του κατηγορήματος, κατά την οποία το τρίτο όρισμα είναι γνωστό (η λάθος λέξη Word) και αναζητούνται τα δύο πρώτα ορίσματα ως δύο συνεχόμενα τμήματα της λέξης.

Πράγματι, η append(Begin, [Letter|End], Word) τεμαχίζει την λίστα Word σε δύο τμήματα, τις λίστες: Begin και [Letter|End]. Η δεύτερη λίστα αποτελείται από την κεφαλή της, που είναι το γράμμα Letter, και την ουρά, που είναι η λίστα End. Άρα, συνολικά, η λίστα Word έχει τεμαχιστεί σε τρία κομμάτια, δηλαδή τα Begin (λίστα), Letter (ένα στοιχείο/γράμμα) και End (λίστα). Το Letter θεωρούμε ότι είναι το λάθος γράμμα του οποίου ζητούμε την αντικατάσταση.

Αντίστοιχα, η δεύτερη κλήση της append/3 μετά την κλήση της word(CorrectWord), συνενώνει τις δύο λίστες Begin και [NewLetter|End] για να παράξει ως αποτέλεσμα ορθή λέξη CorrectWord (Σχήμα 8.1).

Αυτό σημαίνει ότι οι δύο λίστες Word και CorrectWord έχουν τις ίδιες υπολίστες στην αρχή και στο τέλος τους (Begin και End) αλλά διαφέρουν σε κάποιο γράμμα ενδιάμεσα και συγκεκριμένα το γράμμα Letter το οποίο διαφέρει από το γράμμα NewLetter (το Letter **δεν ενοποιείται** με το NewLetter).



Σχήμα 8.1: *Διόρθωση ενός γράμματος με το κατηγόρημα changeletter/2*

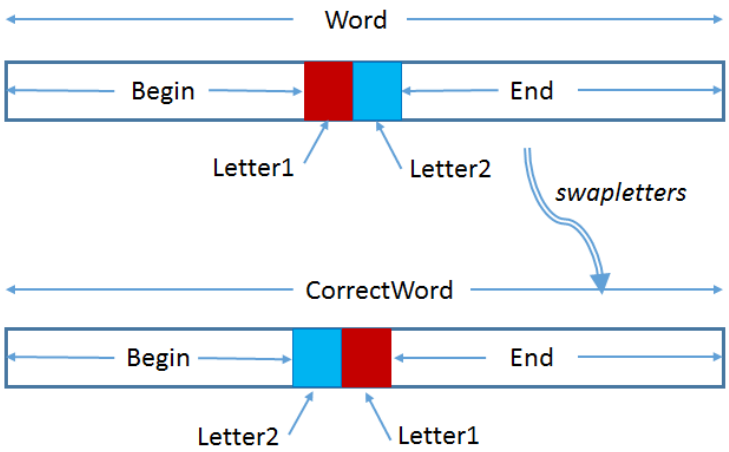
Τέλος, πρέπει να παρατηρήσουμε ότι με βάση τη λογική του append/3 τόσο η λίστα Begin όσο και εναλλακτικά η λίστα End μπορεί να είναι κενές. Στην πρώτη περίπτωση το λάθος εντοπίζεται στο πρώτο γράμμα, ενώ στη δεύτερη λάθος είναι το τελευταίο γράμμα.

Κατηγορήματα υλοποίησης μερικών ακόμα κανόνων διόρθωσης δίνονται στη συνέχεια. Για παράδειγμα, το κατηγόρημα swapletters(Word, CorrectWord), το οποίο δέχεται ως είσοδο τη λίστα Word και επιστρέφει στο δεύτερο όρισμα μια λέξη του λεξικού που είναι ίδια με την Word με εξαίρεση το γεγονός ότι έχει γίνει ανταλλαγή δύο γειτονικών γραμμάτων (Σχήμα 8.2):

```
swapletters(Word, CorrectWord) :-
  append(Begin, [Letter1,Letter2|End], Word),
  word(CorrectWord),
  append(Begin, [Letter2,Letter1|End], CorrectWord),
  Letter1 \= Letter2.
```

Παράδειγμα εκτέλεσης, όπου γίνεται ανταλλαγή των γραμμάτων o και m:

```
?- swapletters([w,m,o,a,n],X).
X = [w,o,m,a,n]
```



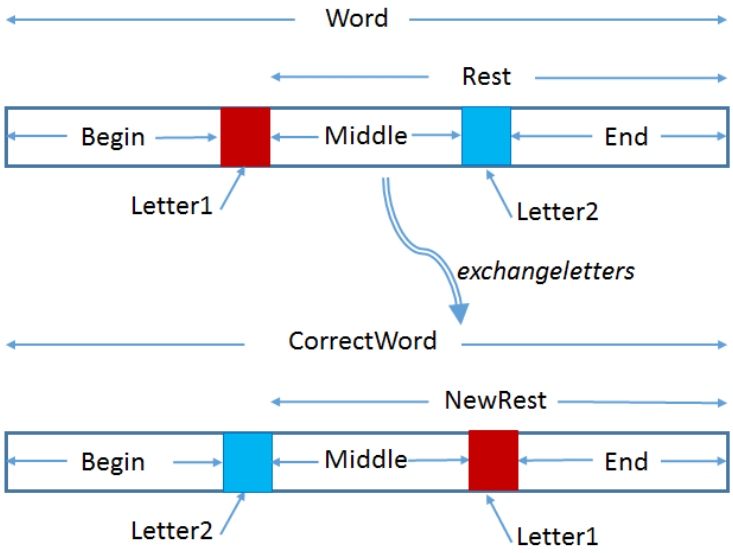
Σχήμα 8.2: *Ανταλλαγή δύο διπλανών γραμμάτων με το κατηγόρημα swapletters/2*

Το κατηγόρημα exchangeletters(Word, CorrectWord), το οποίο δέχεται ως είσοδο τη λίστα Word και επιστρέφει στο δεύτερο όρισμα μια λέξη του λεξικού που είναι ίδια με την Word με εξαίρεση το γεγονός ότι έχει γίνει ανταλλαγή δύο γραμμάτων που δεν είναι κατ' ανάγκη γειτονικά (Σχήμα 8.3):

```
exchangeletters(Word, CorrectWord) :-
  append(Begin, [Letter1|Rest], Word),
  append(Middle, [Letter2|End], Rest),
  word(CorrectWord),
  append(Middle, [Letter1|End], NewRest),
  append(Begin, [Letter2|NewRest], CorrectWord),
  Letter1 \= Letter2.
```

Παράδειγμα εκτέλεσης με ανταλλαγή των γραμμάτων t και o :

```
?- exchangeletters([c,t,m,p,u,o,e,r],X).
X = [c,o,m,p,u,t,e,r];
```



Σχήμα 8.3: *Ανταλλαγή δύο απομακρυσμένων γραμμάτων με το κατηγόρημα exchangeletters/2*

Εδώ πρέπει να παρατηρήσουμε ότι το κατηγόρημα exchangeletters/2 μπορεί να θεωρηθεί ότι καλύπτει και την περίπτωση την οποία υλοποιεί το κατηγόρημα swapletters/2. Η λίστα Middle, σύμφωνα με τις λύσεις που μπορεί να επιστρέψει η append/3 είναι δυνατόν να είναι κενή, περίπτωση κατά την οποία τα δύο γράμματα Letter1 και Letter2 καθίστανται γειτονικά. Αν επιθυμούμε το κατηγόρημα exchangeletters/2 να

ανταλλάσσει γράμματα τα οποία είναι κατ' αποκλειστικότητα απομακρυσμένα, θα έπρεπε να το τροποποιήσουμε ως εξής:

```
exchangeletters(Word, CorrectWord) :-
  append(Begin, [Letter1|Rest], Word),
  append([Letter|Middle], [Letter2|End], Rest),
  word(CorrectWord),
  append([Letter|Middle], [Letter1|End], NewRest),
  append(Begin, [Letter2|NewRest], CorrectWord),
  Letter1 \= Letter2.
```

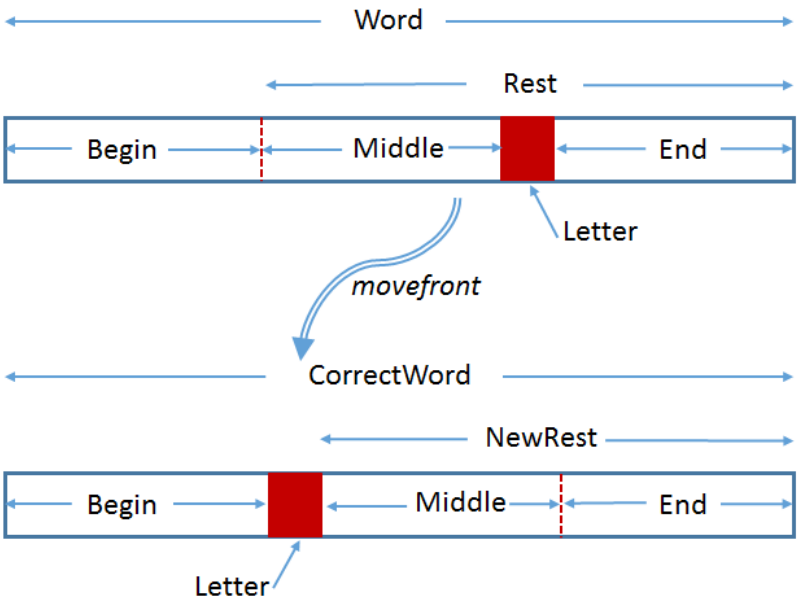
Τοποθετώντας τη λίστα [Letter|Middle] στη θέση της λίστας Middle την εξαναγκάζουμε να έχει τουλάχιστον ένα στοιχείο και άρα της "απαγορεύουμε" να είναι κενή.

Το κατηγόρημα movefront(Word, CorrectWord), το οποίο δέχεται ως είσοδο την λίστα Word που αντιστοιχεί σε μία πιθανά ανορθόγραφη λέξη και επιστρέφει στη λίστα CorrectWord μια ορθογραφημένη λέξη η οποία προήλθε από τη μετακίνηση προς τα εμπρός ενός γράμματος της λέξης Word (Σχήμα 8.4):

```
movefront(Word, CorrectWord) :-
  append(Begin, Rest, Word),
  append(Middle, [Letter|End], Rest),
  word(CorrectWord),
  append([Letter|Middle], End, NewRest),
  append(Begin, NewRest, CorrectWord).
```

Παραδείγματα εκτέλεσης:

```
?- movefront([m,t,h,e,o,r],L).
L = [m, o, t, h, e, r]
?- movefront([o,m,w,a,n],L).
L = [w, o, m, a, n]
```



Σχήμα 8.4: *Μετακίνηση ενός γράμματος προς τα εμπρός με το κατηγόρημα movefront/2*

Το κατηγόρημα delete2letters(Word, CorrectWord), το οποίο δέχεται ως είσοδο τη λίστα Word που αντιστοιχεί σε μία πιθανά ανορθόγραφη λέξη και επιστρέφει στη λίστα CorrectWord μια ορθογραφημένη λέξη η οποία προήλθε μετά την αφαίρεση δύο γραμμάτων από τη λέξη Word. Τα γράμματα μπορούν να αφαιρεθούν από οποιαδήποτε σημεία της λέξης που δεν είναι αναγκαστικά γειτονικά.

```
delete2letters(Word, CorrectWord) :-
  append(Begin, [Letter1|Rest], Word),
  append(Middle, [Letter2|End], Rest),
```

```
    word(CorrectWord),
    append(Middle, End, NewRest),
    append(Begin, NewRest, CorrectWord).
```

Παραδείγματα εκτέλεσης:

```
?- delete2letters([m,o,o,t,h,e,r,w],L).
L = [m, o, t, h, e, r]
?- delete2letters([w,i,o,m,e,a,n],L).
L = [w, o, m, a, n]
```

Συνοψίζοντας, με βάση τα παραπάνω παραδείγματα κανόνων διόρθωσης λαθών που υλοποιήσαμε, μπορούμε να συνάγουμε την εξής γενική τεχνική: Χρησιμοποιώντας το κατηγόρημα append/3 αντίστροφα, θεωρώντας δηλαδή γνωστό το τρίτο του όρισμα που είναι η λίστα που αναπαριστά τη λάθος λέξη, τη χωρίζουμε σε τμήματα προσπαθώντας να απομονώσουμε τα σωστά τμήματα της λέξης από τα λάθος. Αν η αρχική λέξη πρέπει να χωριστεί σε περισσότερα τμήματα χρησιμοποιούμε αντίστοιχα περισσότερες φορές το append/3 χωρίζοντας αρχικά τη λίστα σε δύο υπολίστες και συνεχίζοντας το χωρισμό των υπολιστών όσες φορές απαιτείται. Στη συνέχεια χρησιμοποιώντας το κατηγόρημα append/3, όσες φορές απαιτείται σε κάθε περίπτωση, με την ορθή του φορά (γνωστά τα δύο πρώτα ορίσματα) ανασυνθέτουμε τη λέξη κάνοντας τις αντίστοιχες τροποποιήσεις στις υπολίστες που συνενώνουμε, απαιτώντας η τελική λέξη που δημιουργείται να βρίσκεται στο λεξικό των ορθογραφημένων λέξεων. Στα παραπάνω παραδείγματα αξίζει επίσης να σημειώσουμε τον ευέλικτο τρόπο αναπαράστασης των ορισμάτων που μας επιτρέπει η Prolog ως αποτέλεσμα της χρήσης του ισχυρού μηχανισμού της ενοποίησης για το πέρασμα παραμέτρων. Με βάση αυτόν είχαμε τη δυνατότητα π.χ. να αναπαραστήσουμε ένα μέρος της λέξης ως την υπολίστα [Letter1, Letter2|End] στο κατηγόρημα swapletters/2, δηλώνοντας ρητά ότι αυτή πρέπει ξεκινά με τα δύο γράμματα Letter1 και Letter2.

8.2 Ιντετερμινιστικός προγραμματισμός

Η έννοια του Ιντετερμινισμού (Indeterminism ή Nondeterminism) αναφέρεται στην περίπτωση εκείνη κατά την οποία κάποια συμβάντα δεν ακολουθούν τη λογική του αιτίου – αποτελέσματος, αλλά μπορούν να προκύψουν με τυχαίο ή απρόβλεπτο τρόπο.

Ένας αλγόριθμος θεωρείται ιντετερμινιστικός όταν υπάρχουν σημεία του στα οποία μπορούν να προκύψουν διάφορες εναλλακτικές περιπτώσεις χωρίς αυτές να είναι ρητά προδιαγεγραμμένες στα βήματα του αλγορίθμου. Σε αντίθεση με τη λογική των εντολών επιλογής (if-then-else) με τις οποίες ο προγραμματιστής καθορίζει με σαφήνεια τις πιθανές εναλλακτικές ροές εκτέλεσης του προγράμματος που υλοποιεί τον αλγόριθμο, ένα ιντετερμινιστικό πρόγραμμα διαλέγει στη διάρκεια εκτέλεσης μία επιλογή βασιζόμενο σε έναν μηχανισμό που είναι ανεξάρτητος του κώδικα του. Στη θεωρία των αλγορίθμων η έννοια του ιντετερμινισμού είναι γνωστή από τα Μη Αιτιοκρατικά Πεπερασμένα Αυτόματα, στα οποία από την τρέχουσα κατάσταση το ίδιο σύμβολο (ή τελεστής μετάβασης) ενδέχεται να οδηγεί σε περισσότερες της μιας καταστάσεις.

Από πρακτική άποψη ένας ιντετερμινιστικός αλγόριθμος βασίζεται στην εξής αρχή: με τη βοήθεια κάποιας διαδικασίας επιλέγεται μία τιμή (ή ένα σύνολο τιμών). Αν αυτή η τιμή οδηγήσει σε ένα αποδεκτό αποτέλεσμα έχει καλώς. Αν όχι η διαδικασία επιλογής επαναλαμβάνεται για να βρεθεί μία νέα τιμή, κ.ο.κ.

Η παραπάνω αρχή αναφέρεται στην κατάσταση όπου ξέρουμε ότι κάποια ή κάποιες εναλλακτικές περιπτώσεις οδηγούν σε σωστό ή σωστά αποτελέσματα, αλλά δεν γνωρίζουμε εκ των προτέρων ποιες είναι αυτές Ένας αλγόριθμος που βασίζεται σε αυτή την αρχή αναφέρεται ως ιντετερμινιστικός τύπου "δεν γνωρίζω" (don't know indeterminism).

Μία άλλη απλή μορφή ιντετερμινιστικού αλγόριθμου σχετίζεται με την κατάσταση κατά την οποία υπάρχουν πολλές εναλλακτικές περιπτώσεις, από τις οποίες αν επιλεγεί οποιαδήποτε, παράγεται αποτέλεσμα που μας ικανοποιεί. Ένας τέτοιος αλγόριθμος αναφέρεται ως ιντετερμινιστικός τύπου "δεν με νοιάζει" (don't care indeterminism).

Μία άλλη περίπτωση ιντετερμινιστικού αλγορίθμου μπορεί να προκύψει όταν ένας προγραμματιστής χρησιμοποιεί κάποια πιθανολογική μέθοδο για την εύρεση λύσης η οποία βασίζεται σε μία γεννήτρια τυχαίων

αριθμών, όπως είναι, παραδείγματος χάριν, η χρήση της μεθόδου Monte-Carlo για την εύρεση του εμβαδού ενός κύκλου εγγεγραμμένου σε τετράγωνο γνωστής πλευράς. Στη μέθοδο αυτή επιλέγεται με τυχαίο τρόπο ένας αριθμός σημείων του τετραγώνου. Αν πολλαπλασιάσουμε το ποσοστό των σημείων που βρέθηκαν εντός του κύκλου με το εμβαδόν του τετραγώνου προσεγγίζουμε το εμβαδόν του κύκλου.

Ο λογικός προγραμματισμός αποτελεί κατά βάση μία ιντετερμινιστική γλώσσα προγραμματισμού. Ο ιντετερμινισμός προκύπτει από το γεγονός ότι δοθέντος ενός προγράμματος και μίας ερώτησης που το ενεργοποιεί μπορούν να πυροδοτηθούν περισσότερες από μία διαδικασίες έγκυρων υπολογισμών. Οι διαφορετικές αυτές διαδικασίες υπολογισμών μπορούν να προκύψουν καθώς:

- (α) Στην περίπτωση που η ερώτηση περιλαμβάνει περισσότερα από ένα υπο-ερωτήματα αυτά μπορούν να εκτελεστούν με μία οποιαδήποτε σειρά, και
- (β) στο πρόγραμμα μπορούν να υπάρχουν περισσότερες από μία φράσεις που ταιριάζουν και μπορούν να επιλεγούν για την ικανοποίηση ενός ερωτήματος.

Στην Prolog παρόλο που μία διαδικασία υπολογισμού βασίζεται σε έναν αλγόριθμο "από αριστερά στα δεξιά και από πάνω προς τα κάτω" (βλέπε κεφάλαιο 5), η δυνατότητα ανάπτυξης ιντετερμινιστικών αλγορίθμων είναι έμφυτη λόγω του μηχανισμού εκτέλεσης που διαθέτει, ο οποίος με τη σειρά του βασίζεται στον αυτόματο μηχανισμό της οπισθοδρόμησης. Η μηχανισμός αυτός απαλλάσσει τον προγραμματιστή από το να καθορίσει ρητά τα βήματα επιλογής των διαφορετικών λύσεων στο πρόγραμμα και μπορεί να υποστηρίξει τους δύο τύπους ιντετερμινισμού "δεν γνωρίζω" και "δεν με νοιάζει".

Ένα απλό πρόγραμμα που μπορεί να περιγράψει την ιντετερμινιστική τεχνική ανάπτυξης ενός αλγορίθμου είναι αυτό που αποφασίζει αν δύο λίστες έχουν τουλάχιστον ένα κοινό στοιχείο:

```
common_element(List1,List2):-
  list_member(X,List1),
  list_member(X,List2).
```

Η λογική του κατηγορήματος common_element/2 είναι η εξής: αν βρεθεί ένα στοιχείο X της λίστας List1, το οποίο υπάρχει και στη λίστα List2, τότε οι δύο λίστες έχουν κοινό στοιχείο. Η πρώτη κλήση του κατηγορήματος list_member/2 παράγει μέλη της πρώτης λίστας (η μεταβλητή X κατά την ώρα της κλήσης είναι ελεύθερη) και η δεύτερη κλήση της list_member/2 ελέγχει αν το X υπάρχει στη δεύτερη λίστα (η μεταβλητή X τώρα είναι δεσμευμένη). Αν σκεφτούμε ιντετερμινιστικά, η πρώτη κλήση "μαντεύει" το κοινό στοιχείο των δύο λιστών και η δεύτερη κλήση το επιβεβαιώνει. Αν σκεφτούμε με βάση τη λογική της εκτέλεσης του παραπάνω προγράμματος, στην πραγματικότητα, έχουμε δύο φωλιασμένους βρόχους επανάληψης. Ο εξωτερικός βρόχος διατρέχει επαναληπτικά όλα τα στοιχεία της πρώτης λίστας και ο εσωτερικός διατρέχει τα στοιχεία της δεύτερης λίστας μέχρι ότου βρεθεί το κοινό στοιχείο. Έναν τέτοιο αλγόριθμο θα αναπτύσσαμε στην περίπτωση που θα υλοποιούσαμε το ίδιο πρόβλημα με τη βοήθεια μία κλασσικής γλώσσας προγραμματισμού όπως είναι η Java ή η C++. Το ενδιαφέρον με την υλοποίηση στην Prolog είναι ότι ο προγραμματιστής απαλλάσσεται τελείως από το να δηλώσει ρητά όλες τις λεπτομέρειες των επαναλήψεων και των σχετικών συνθηκών τερματισμού.

Το παραπάνω πρόγραμμα μπορεί να χαρακτηριστεί ιντετερμινιστικό του τύπου "δεν γνωρίζω" όσο επίσης και του τύπου "δεν με νοιάζει". Στην πρώτη περίπτωση ξέρουμε από την αρχή ότι για να έχουν οι λίστες κοινό στοιχείο πρέπει αυτό να βρίσκεται και στην πρώτη και στη δεύτερη, χωρίς όμως να γνωρίζουμε ποιο είναι αυτό. Στη δεύτερη περίπτωση μπορούμε να θεωρήσουμε ότι μπορεί να υπάρχουν περισσότερα του ενός στοιχεία που κάνουν το κατηγόρημα common_element/2 να αληθεύει, χωρίς όμως να μας νοιάζει ποιο από όλα θα είναι αυτό που θα μας δώσει το αποτέλεσμα.

Ας δούμε επίσης το παρακάτω κατηγόρημα list_sort/2, με την βοήθεια του οποίου είναι δυνατή η ταξινόμηση μίας λίστας αριθμών σε αύξουσα τάξη με βάση ένα ιντετερμινιστικό σχήμα.

```
list_sort(L,SL):-
  permutate(L,SL),
  isordered(SL).

permutate([],[]).
permutate(L,[H|PL]):-
```

```
  list_delete(H,L,NL),
  permutate(NL,PL).

isordered([]).
isordered([X]).
isordered([X1,X2|T]):-
  X1 =< X2,
  isordered([X2|T]).
```

Μέσω του κατηγορήματος permutate/2 επιλέγεται μία τυχαία μετάθεση των στοιχείων της αρχικής λίστας που θέλουμε να ταξινομήσουμε και στη συνέχεια μέσω του κατηγορήματος isordered/1 ελέγχουμε αν η μετάθεση αυτή "έτυχε" να είναι μία λίστα που τα στοιχεία της βρίσκονται σε αύξουσα τάξη. Αν αυτό δεν συμβεί τότε ο αυτόματος μηχανισμός της οπισθοδρόμησης προκαλεί νέα εκτέλεση του κατηγορήματος permutate/2 το οποίο θα δώσει μία νέα μετάθεση των στοιχείων προς έλεγχο αν αυτά είναι διατεταγμένα. Η παραπάνω διαδικασία επαναλαμβάνεται μέχρι την ικανοποίηση της κλήσης του κατηγορήματος isordered/1.

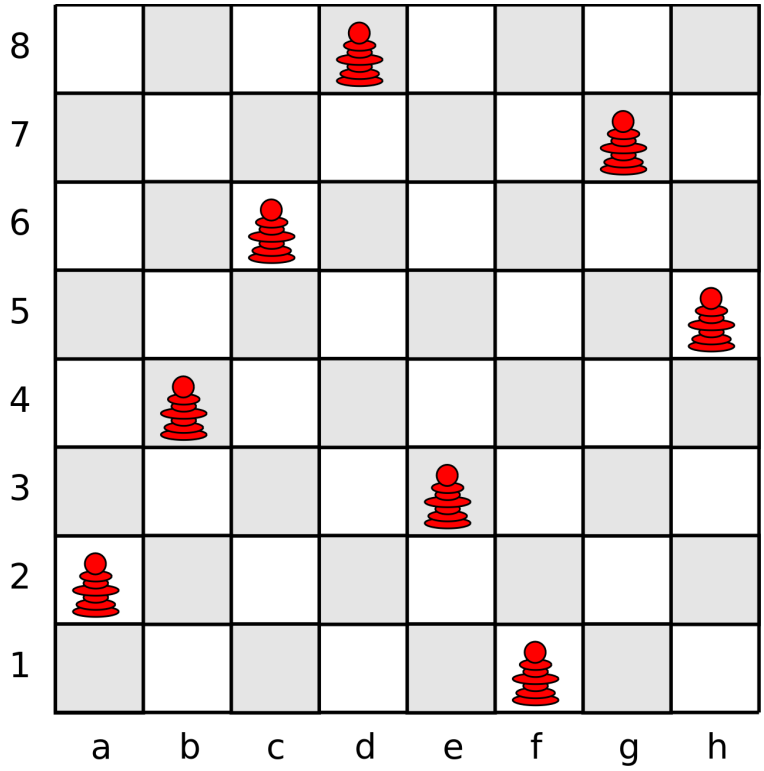
Το κατηγόρημα permutate/2, με τη σειρά του βασίζεται στην εξής λογική: Διαγράφουμε ένα τυχαίο στοιχείο Η της αρχικής λίστας, το οποίο θέτουμε ως κεφαλή της λίστας που περιλαμβάνει τη μετάθεση. Τα στοιχεία της ουράς είναι αυτά που προκύπτουν αναδρομικά ως μετάθεση της λίστας που προέκυψε μετά τη διαγραφή του Η από την αρχική λίστα.

Το πρόβλημα των 8 βασιλισσών

Η λογική του ιντετερμινισμού του τύπου "δεν γνωρίζω" μπορεί να χρησιμοποιείται με επαναληπτικό τρόπο. Αν όλες οι πιθανές εναλλακτικές λύσεις σε κάποιο σημείο επιλογής ενός προγράμματος εξαντληθούν μέσω της διαδικασίας οπισθοδρόμησης χωρίς επιτυχία τότε το πρόγραμμα πυροδοτεί εκ νέου οπισθοδρόμηση σε προηγούμενο σημείο επιλογής.

Ας δούμε την περίπτωση αυτή σε ένα πιο σύνθετο παράδειγμα αλγορίθμου: Πρόκειται για την επίλυση ενός γνωστού αλγοριθμικού προβλήματος, αυτού των 8 βασιλισσών: Προσπαθούμε να τοποθετήσουμε σε μία σκακιέρα 8 βασίλισσες με τέτοιο τρόπο ώστε καμία από αυτές να μην απειλεί τις υπόλοιπες, καμία βασίλισσα δηλαδή να μην βρίσκεται στην ίδια γραμμή, στην ίδια στήλη και στην ίδια διαγώνιο της σκακιέρας με οποιαδήποτε άλλη.

```
solve(QueenList):-
  template(QueenList),
  solution(QueenList).

template([[1,Y1],[2,Y2],[3,Y3],[4,Y4],[5,Y5],[6,Y6],[7,Y7],[8,Y8]]).

solution([]).
solution([[X,Y]|RestQueens]):-
  list_member(Y, [1,2,3,4,5,6,7,8]),
  solution(RestQueens),
  safe([X,Y],RestQueens).

safe(_,[]).
safe([X,Y],[[Xa,Ya]|RestQueens]):-
  Y=\=Ya,
  X-Xa =\=Y-Ya,
  X-Xa =\=Ya-Y,
  safe([X,Y],RestQueens).
```



Σχήμα 8.5: *Μία λύση του προβλήματος των 8 βασιλισσών*

Ως λύση του παραπάνω προβλήματος θεωρούμε μία οκτάδα ζευγαριών που αναφέρουν την αντίστοιχη γραμμή και στήλη που έχει τοποθετηθεί η κάθε βασίλισσα. Παραδείγματος χάριν, η λίστα [[1,6],[2,1],[3,5],[4,2],[5,8],[6,3],[7,7],[8,4]] αντιστοιχεί στη λύση τοποθέτησης των 8 βασιλισσών που φαίνεται στο Σχήμα 8.5. Οι δυάδες αναπαριστάνονται με λίστες δύο στοιχείων στις οποίες για την αρίθμηση των στηλών έχουμε αντιστοιχίσει στα γράμματα a έως h τους αριθμούς 1 έως 8. Στόχος του προγράμματος είναι να δημιουργήσει τη λίστα-λύση με την παραπάνω μορφή. Παρατηρούμε ότι με βάση τους περιορισμούς του προβλήματος, γνωρίζουμε εκ των προτέρων ότι σε κάθε γραμμή θα υπάρχει μία και μόνο μία βασίλισσα κάτι που μπορούμε να το δηλώσουμε με τη βοήθεια του κατηγορήματος/γεγονότος template/1, το οποίο περιλαμβάνει μία πρώτη προσέγγιση της λίστας αποτέλεσμα, με προ-τοποθετημένες στις δυάδες τις τιμές των γραμμών και μη-τοποθετημένες μεταβλητές στις θέσεις των αντίστοιχων στηλών.

Το κατηγόρημα solve/2 καλεί το template/1 και στη συνέχεια μέσω της κλήσης του κατηγορήματος solution/1 συμπληρώνει τη λίστα-λύση με τις αντίστοιχες τιμές των στηλών.

Στο αναδρομικό κατηγόρημα solution/1 φαίνεται η επαναληπτική χρήση του ιντετερμινιστικού σχήματος (επιλογή μίας τιμής/λύσης, έλεγχος στη συνέχεια αν αυτή είναι αποδεκτή, οπισθοδρόμηση για αναζήτηση εναλλακτικής τιμής/λύσης). Αρχικά μέσω του list_member/2 επιλέγεται μία τιμή για τη στήλη που πρέπει να τοποθετηθεί η πρώτη από τις βασίλισσες στην αντίστοιχη γραμμή στην οποία βρίσκεται (υπενθυμίζουμε ότι το X είναι ήδη τοποθετημένο μέσω του template/1). Στη συνέχεια αναζητείται λύση μέσω της αναδρομικής κλήσης του κατηγορήματος solution/1 για την τοποθέτηση όλων των υπόλοιπων βασιλισσών σύμφωνα με τους κανόνες του προβλήματος (καμία δεν απειλεί τις υπόλοιπες). Τέλος μέσω της κλήσης του κατηγορήματος safe/2 ελέγχεται αν η πρώτη βασίλισσα που αντιστοιχεί στη δυάδα [X,Y] δεν απειλεί όλες τις υπόλοιπες που αντιστοιχούν στις δυάδες της λίστας RestQueens.

Υπάρχουν δύο σημεία στις προϋποθέσεις του κανόνα που υλοποιεί το κατηγόρημα solution/1 που βασίζονται στο ιντετερμινιστικό σχήμα τύπου “δεν γνωρίζω”:

- Μέσω της list_member/2 ο προγραμματιστής θεωρεί ότι υπάρχει η σωστή τιμή, από 1 έως 8, για τη στήλη που πρέπει να τοποθετηθεί η βασίλισσα, χωρίς όμως να γνωρίζει ποια ακριβώς είναι αυτή. Στο

σημείο αυτό χρησιμοποιεί τη list_member/2 με τη λογική "έστω ότι μπορεί να βρεθεί κατάλληλη τιμή για το Y ώστε να τοποθετηθεί η βασίλισσα στη σκακιέρα".

- Μέσω της solution/1 ο προγραμματιστής θεωρεί ότι υπάρχει ένα σύνολο δυάδων της λίστας RestQueens για την τοποθέτηση όλων των υπολοίπων βασιλισσών, χωρίς επίσης να γνωρίζει ποιες ακριβώς είναι αυτές. Και εδώ χρησιμοποιεί την αναδρομική κλήση του solution/1 με τη λογική "έστω ότι μπορεί να βρεθεί η κατάλληλη λίστα τιμών για την τοποθέτηση των υπολοίπων βασιλισσών στη σκακιέρα με σωστό τρόπο".

Αν η κλήση solution/1 αποτύχει αυτό σημαίνει ότι η επιλογή τιμής μέσω της list_member/2 δεν ήταν η κατάλληλη, οπότε προκαλείται οπισθοδρόμηση για να αναζητηθεί νέα τιμή για το Y (εφόσον υπάρχουν διαθέσιμες τιμές που δεν έχουν δοκιμαστεί) από τη list_member/2. Με τον ίδιο τρόπο αν η κλήση safe/2 αποτύχει αυτό σημαίνει ότι η επιλογή της λίστας των θέσεων των βασιλισσών που παρήγαγε η solution δεν ήταν κατάλληλη, οπότε και στο σημείο αυτό προκαλείται οπισθοδρόμηση για να αναζητηθεί νέα λίστα με θέσεις βασιλισσών.

Όπως φαίνεται από τα παραπάνω παραδείγματα, η κύρια διαφορά μεταξύ της μη-ντετερμινιστικής συμπεριφοράς από την ντετερμινιστική είναι ότι η πρώτη εξετάζει τα διάφορα είδη των επιλογών χωρίς να διευκρινίζει πώς αυτές θα προκύψουν, ενώ η δεύτερη περιγράφει με λεπτομέρεια τόσο πώς θα γίνουν αυτές οι επιλογές όσο και τι θα συμβεί μετά από κάθε επιλογή που δοκιμάζεται και αποτυγχάνει. Στην Prolog ο μηχανισμός της οπισθοδρόμησης είναι σε θέση να επαναφέρει την κατάσταση ενός προγράμματος αναιρώντας όλες τις αντικαταστάσεις μεταβλητών με όρους που πραγματοποιήθηκαν από την κλήση του ενός κατηγορήματος που τελικά απέτυχε. Η διαδικασία αυτή γίνεται αυτόματα, χωρίς ο προγραμματιστής να χρειάζεται να προσθέσει οποιοδήποτε τμήμα κώδικα.

Με τη λογική του ιντετερμινιστικού προγραμματισμού συνδέονται πολλές τεχνικές επίλυσης προβλημάτων τις οποίες θα δούμε σε επόμενα κεφάλαια. Μερικά από αυτά είναι τα προβλήματα:

- Παραγωγής και Δοκιμής (Genarate and Test) ή Προσπάθειας και Αποτυχίας (Trial and Error),
- Ικανοποίησης περιορισμών (constraint satisfation problems),
- Επίλυσης γράφων,
- Αναζήτησης.

8.3 Συμβολικός προγραμματισμός

Ένα πολύ σημαντικό πεδίο εφαρμογών, για το οποίο η Prolog είναι κατάλληλη γλώσσα, είναι ο συμβολικός προγραμματισμός (symbolic programming). Ο συμβολικός προγραμματισμός σχετίζεται κυρίως με την επεξεργασία συμβόλων ή συμβολικών δεδομένων σε αντιδιαστολή με την επεξεργασία αριθμητικών δεδομένων. Τα συμβολικά δεδομένα μπορούν να αναπαριστούν αντικείμενα ή οντότητες καθώς και τις σχέσεις ανάμεσά τους. Η συνεισφορά του συμβολικού προγραμματισμού ήταν σημαντική στην ανάπτυξη εφαρμογών της Τεχνητής Νοημοσύνης, καθώς εισήγαγε έναν εύκολο τρόπο δημιουργίας, αναπαράστασης και επεξεργασίας σύνθετων δομών δεδομένων, όπως είναι οι λίστες, τα δέντρα, οι γράφοι και άλλες μορφές αναπαράστασης του χώρου αναζήτησης ενός προβλήματος. Η παραπάνω ευκολία είναι ευδιάκριτη σε κάποιο προγραμματιστή που έχει εμπειρία στη δημιουργία προγραμμάτων που χειρίζονται συνδεδεμένες λίστες ή άλλες δυναμικές δομές δεδομένων σε γλώσσες όπως η C/C++ και η Java και τα αντιπαραβάλει με τα αντίστοιχα της Prolog.

Στην Prolog, δύο είναι τα βασικά χαρακτηριστικά που δίνουν τη δυνατότητα συμβολικού προγραμματισμού: (α) Η δυνατότητα μίας μεταβλητής να παραμένει μη-δεσμευμένη (ελεύθερη) σε κάποια τιμή κατά τη διάρκεια εκτέλεσης ενός προγράμματος και (β) το γεγονός ότι η Prolog (επίσης κατά τη διάρκεια της εκτέλεσης) δεν υπολογίζει τις εκφράσεις που εμφανίζονται ως παράμετροι των κατηγορημάτων.

Η δυνατότητα χρήσης σύνθετων όρων που περιλαμβάνουν ελεύθερες μεταβλητές μας δίνει τη δυνατότητα αναπαράστασης σύνθετων δομών δεδομένων με τη συμβολική τους μορφή. Ένα τέτοιο παράδειγμα, που έχουμε ήδη δει στην προηγούμενη παράγραφο, είναι αυτό της αναπαράστασης της λύσης

του προβλήματος των 8 βασιλισσών στην οποία χρησιμοποιήσαμε μία λίστα με στοιχεία, που ήταν λίστες δύο στοιχείων, στην οποία συμπεριλαμβάνονταν οκτώ ελεύθερες μεταβλητές:

```
template([[1,Y1],[2,Y2],[3,Y3],[4,Y4],[5,Y5],[6,Y6],[7,Y7],[8,Y8]]).
```

Το ενδιαφέρον με την παραπάνω αναπαράσταση της λίστας είναι ότι η Prolog δεν αναζητά τιμές για τις οκτώ μεταβλητές και μπορεί να τις μεταβιβάζει ως έχουν (ως ελεύθερες) από κλήση κατηγορήματος σε κλήση κατηγορήματος κατά τη διάρκεια εκτέλεσης ενός προγράμματος. Με άλλα λόγια η Prolog θεωρεί τις μεταβλητές ως ένα είδος συμβολικής τιμής που επιστρέφεται.

Στις περισσότερες γλώσσες προγραμματισμού όταν σε κάποιο σημείο ενός προγράμματος εμφανίζεται μία αριθμητική έκφραση αυτή αμέσως παραπέμπει στον υπολογισμό της. Αντίθετα στην Prolog η έκφραση αυτή αντιστοιχεί σε ένα σύνθετο (συναρτησιακό) όρο που αναπαριστά μία δομή δεδομένων με τη συμβολική της μορφή. Παραδείγματος χάριν αν ρωτήσουμε:

```
?- X = 3+5.
```

η απάντηση δεν θα είναι 8, καθώς το 3+5 αντιμετωπίζεται ως μία συμβολική έκφραση, δεν είναι δηλαδή τίποτε περισσότερο παρά ο σύνθετος όρος +(3,5) γραμμένος με ενθεματικό τρόπο ως προς το σύμβολο +. Ας δούμε πώς λειτουργεί η δυνατότητα της Prolog να θεωρεί τις εκφράσεις με τη συμβολική τους μορφή και να μην τις υπολογίζει στο παρακάτω παράδειγμα:

```
symbolic_sum([],0).
symbolic_sum([H|T],S):-
  symbolic_sum(T,ST),
  S=H+ST.
```

Αν εκμεταλλευτούμε το γεγονός ότι το "=" στην Prolog είναι ο τελεστής ενοποίησης μπορούμε να μεταφέρουμε το αποτέλεσμα της ενοποίησης του S με το H+ST, στο δεύτερο όρισμα του κατηγορήματος, οπότε ο κανόνας του παραπάνω προγράμματος αποκτά την παρακάτω μορφή, που δείχνει καλύτερα τη δυνατότητα συμβολικής αναπαράστασης (και η οποία μάλιστα εκμεταλλεύεται και την αναδρομή ουράς):

```
symbolic_sum([H|T],H+ST):-
  symbolic_sum(T,ST).
```

Αν τώρα ενεργοποιήσουμε την εκτέλεση του παραπάνω κατηγορήματος, θέτοντας ως πρώτο όρισμα μία λίστα αριθμών, θα δούμε ότι αυτό επιστρέφει, μία συμβολική έκφραση αθροίσματος των αριθμών αυτών χωρίς να έχουν γίνει οι πράξεις:

```
?- symbolic_sum([8,1,2,4],S).
S = 8+1+2+4+0.
```

Αν στη λίστα εκτός από αριθμούς περιλαμβάνονται και μεταβλητές, με την έννοια ότι σε κάποια εφαρμογή δεν γνωρίζουμε όλα τα δεδομένα τη στιγμή της εκτέλεσης του προγράμματος, τότε το αποτέλεσμα είναι αντίστοιχο ως προς τη συμβολική του μορφή:

```
?- symbolic_sum([8,X,2,Y,4]).
S = 8+X+2+Y+4+0.
```

Να τονίσουμε φυσικά εδώ, ότι όπως έχουμε δει, όταν επιθυμούμε να υπολογιστεί η τιμή μίας αριθμητικής έκφρασης μπορούμε να χρησιμοποιήσουμε το ενσωματωμένο κατηγόρημα is/2. Στο παρακάτω παράδειγμα εκτέλεσης φαίνεται ότι αφού έχει επιστραφεί η συμβολική έκφραση του αθροίσματος στη μεταβλητή S, στη συνέχεια ζητάμε να υπολογιστεί η τιμή της και να επιστραφεί στη θέση της μεταβλητής SA:

```
?- symbolic_sum([8,1,2,4],S), SA is S.
S = 8+1+2+4+0,
SA = 15.
```

Θα μπορούσαμε λοιπόν να δημιουργήσουμε ένα ειδικό κατηγόρημα αριθμητικού αθροίσματος με βάση το κατηγόρημα του συμβολικού αθροίσματος, ως εξής:

```
arithmetic_sum(L,SA):-
  symbolic_sum(L,S),
  SA is S.
```

Ο παραπάνω τρόπος διαχωρισμού του συμβολικού από τον αριθμητικό υπολογισμό του αθροίσματος μπορεί να λειτουργήσει ως παράδειγμα καλής πρακτικής: Προσπαθούμε να ξεχωρίσουμε ρητά στα προγράμματα της

Prolog το συμβολικό μέρος του προγραμματισμού από το αριθμητικό, γράφοντας ξεχωριστά κατηγορήματα. Με τον τρόπο αυτό, το συμβολικό μέρος του κώδικα είναι γραμμένο σε "καθαρή" (pure) Prolog διατηρώντας όλα τα πλεονεκτήματα του λογικού προγραμματισμού και καταφεύγουμε στη χρήση ενσωματωμένων κατηγορημάτων όταν επιθυμούμε στο πρόγραμμά μας να πραγματοποιηθεί κάποιος αριθμητικός υπολογισμός.

Με το παραπάνω απλό παράδειγμα φαίνεται η δυνατότητα της Prolog να χρησιμοποιεί μερικά ορισμένες δομές δεδομένων (partially defined data structures), που είναι απαραίτητες για την αντιμετώπιση προβλημάτων της Τεχνητής Νοημοσύνης στα οποία συχνά έχουμε ελλιπή δεδομένα. Η διαδικασία της εκτέλεσης ενός προγράμματος μπορεί να προχωρά ανεξάρτητα από την ύπαρξη κάποιων δεδομένων του προβλήματος ή όχι. Αν στη συνέχεια, σε κάποιο σημείο της εκτέλεσης, μπορούν τα δεδομένα αυτά να γίνουν γνωστά ή να υπολογιστούν από κάποια άλλη διαδικασία, ή ακόμα και να προβλεφθούν ή να παραχθούν υποθετικά, ο υπολογισμός μπορεί να τα λάβει υπόψη του παράγοντας αποτελέσματα. Στο παρακάτω παράδειγμα εκτέλεσης μπορούμε να θεωρήσουμε ότι το κατηγόρημα guess/1 παράγει κάποια τιμή για το X (παραδείγματος χάριν μέσω τυχαίας παραγωγής τιμών ή ρωτώντας το χρήστη του προγράμματος). Στη συνέχεια η τιμή αντικαθιστά το X της συμβολικής έκφρασης και γίνεται δυνατός ο αριθμητικός υπολογισμός της.

```
?- symbolic_sum([8,1,2,X,4],S), guess(X), SA is S.
S = 8+1+2+9+4+0,
X = 9,
SA = 24.
```

Συμβολική παραγώγιση

Για να κατανοήσουμε καλύτερα την αξία του συμβολικού προγραμματισμού ας ξαναδούμε ως πιο σύνθετο παράδειγμα προγράμματος αυτό της παραγώγισης που παρουσιάστηκε στην ενότητα 6.4, με δύο ενδεικτικά παραδείγματα εκτέλεσης:

```
derivative(X, X, 1).
derivative(C, X, 0) :-
  atomic(C),
  C \=X.

derivative(sin(X), X, cos(X)).

derivative(cos(X), X, -sin(X)).

derivative(U+V, X, A+B) :-
  derivative(U, X, A),
  derivative(V, X, B).

derivative(U*V, X, A*V+B*U) :-
  derivative(U, X, A),
  derivative(V, X, B).
```

Παραδείγματα εκτέλεσης:

```
?- derivative(x*x + 2, x, P).
P = 1*x+1*x+0
?- derivative(x*x*x + 2, x, P).
P = (1*x+1*x)*x + 1*(x*x)+0
```

Οι απαντήσεις που παράγονται με την υποβολή των ερωτήσεων και επιστρέφονται στο τρίτο όρισμα του κατηγορήματος derivative/3 αποτελούν τις μαθηματικές εκφράσεις, με τη συμβολική τους μορφή, των συναρτήσεων που αντιστοιχούν στις παραγώγους των συναρτήσεων που τίθενται στο πρώτο όρισμα. Πρέπει να παρατηρήσουμε ότι στις εκφράσεις που τοποθετούμε στα ορίσματα της ερώτησης, το x αναπαριστάνεται με τη βοήθεια σταθεράς και όχι ως μεταβλητή X. Γενικότερα οι μεταβλητές μιας μαθηματικής έκφρασης πρέπει να αναπαρασταθούν με σταθερές, σε αντίθεση με τα ορίσματα στις φράσεις που ορίζουν το κατηγόρημα derivative στα οποία χρησιμοποιούμε μεταβλητές της Prolog. Επίσης υπενθυμίζουμε ότι αν

θέλουμε οι εκφράσεις των συναρτήσεων που επιστρέφονται να εμφανίζονται με την απλοποιημένη τους μορφή, θα πρέπει να προσθέσουμε τις αντίστοιχες φράσεις που την ορίζουν. Η απλοποίηση των εκφράσεων έχει αφεθεί ως άσκηση στο Κεφάλαιο 6 (κατηγόρημα simplify/2 της άσκησης 6.7).

Για να γίνει κατανοητή η σημασία της συμβολικής έκφρασης της παραγώγου θα μπορούσαμε να συγκρίνουμε το πρόγραμμα της συμβολικής παραγώγισης με οποιοδήποτε υλοποίηση προγράμματος βασισμένου σε τεχνικές αριθμητικής ανάλυσης. Στην δεύτερη περίπτωση αναζητείται (προσεγγίζεται) η αριθμητική τιμή της παραγώγου σε κάποιο συγκεκριμένο σημείο του πεδίου ορισμού της συνάρτησης, π.χ. αναζητούμε ποία είναι η τιμή της παραγώγου της συνάρτησης X^3+2 για X=5. Στην πρώτη περίπτωση εφόσον έχουμε τη συμβολική έκφραση της συνάρτησης, μπορούμε να βρούμε την τιμή της σε οποιοδήποτε σημείο του πεδίου ορισμού της με απλή αντικατάσταση και εκτέλεση των πράξεων.

8.4 Επαυξητικός προγραμματισμός και προγραμματισμός "από άνω προς τα κάτω"

Με βάση τα χαρακτηριστικά της η Prolog μπορεί να υποστηρίξει με φυσικό τρόπο δύο συμπληρωματικές μεθοδολογίες ανάπτυξης προγραμμάτων: Τον επαυξητικό προγραμματισμό και τον προγραμματισμό από άνω προς τα κάτω.

Η σειρά με την οποία τοποθετούνται οι φράσεις της Prolog σε ένα πρόγραμμα είναι αδιάφορη (τουλάχιστον στις περισσότερες περιπτώσεις). Παραδείγματος χάριν, φράσεις που αναπαριστάνονται με τη βοήθεια γεγονότων μπορούν να τοποθετηθούν στο πρόγραμμα με οποιαδήποτε σειρά χωρίς αυτή να επηρεάζει το τελικό νόημα του προγράμματος. Το ίδιο ισχύει, στις περισσότερες των περιπτώσεων και για φράσεις που αντιστοιχούν σε κανόνες. Αυτό με τη σειρά του σημαίνει ότι μπορεί κανείς επαυξήσει την εκφραστική ικανότητα ενός προγράμματος προσθέτοντας διαδοχικά νέες φράσεις χωρίς να χρειάζεται καμία τροποποίηση των παλαιότερων. Η διαδικασία αυτή αναφέρεται ως επαυξητικός προγραμματισμός (incremental programming) και αντιστοιχεί στην "από κάτω προς τα άνω" (bottom-up) ανάπτυξη ενός προγράμματος. Στην πιο απλή περίπτωση μπορούμε να θεωρήσουμε ότι ένα πρόγραμμα έχει αναπτυχθεί σταδιακά και τα επιμέρους τμήματά του βρίσκονται σε αντίστοιχα αρχεία κώδικα Prolog, file1, file2, . . . , fileN, τα οποία φορτώνονται στον διερμηνέα (interpreter) της Prolog με τη βοήθεια του κατηγορήματος consult, δημιουργώντας ένα περιβάλλον εκτέλεσης για την επίλυση του προβλήματος:

```
?- consult(File1),consult(File2), . . . , consult(FileN).
```

Μία πάρα πολύ γνωστή και αποτελεσματική μεθοδολογία ανάπτυξης προγραμμάτων είναι αυτή του "διαίρει και βασίλευε" (divide and conquer), που αναφέρεται και ως προγραμματισμός "από άνω προς τα κάτω" ή ακόμα και ως μεθοδολογία των διαδοχικών προσεγγίσεων (successive refinements). Σύμφωνα με τη μεθοδολογία αυτή, ο προγραμματιστής προσπαθεί να χωρίσει το αρχικό πρόβλημα που έχει να επιλύσει σε υποπροβλήματα τα οποία είναι όσο το δυνατόν ανεξάρτητα μεταξύ τους. Η ίδια διαδικασία εφαρμόζεται για κάθε υποπρόβλημα ξεχωριστά το οποίο με τη σειρά του χωρίζεται σε επιμέρους μικρότερα υποπροβλήματα, κ.ο.κ. Η διαδικασία αυτή σταματά όταν τα επί μέρους υποπροβλήματα είναι άμεσα υλοποιήσιμα. Η διαδικαστική ερμηνεία των φράσεων της Prolog παραπέμπει κατ' ευθείαν σε αυτόν τον τρόπο ανάπτυξης προγραμμάτων με βάση το παρακάτω γενικό σχήμα:

```
problem(parameter_list):-
  subproblem1(parameter_list_1),
  subproblem2(parameter_list_2),
  . . .
  subproblemN(parameter_list_N).

subproblem1(parameter_list_1):-
  subproblem1_1(parameter_list_1_1),
  subproblem1_2(parameter_list_1_2),
  . . .
  subproblem1_K(parameter_list_1_K).
. . .
κ.ο.κ.
```

Το αρχικό πρόβλημα αναπαριστάνεται με το κατηγόρημα problem, το οποίο περιλαμβάνει έναν αριθμό παραμέτρων parameter_list. Υλοποιείται με τη βοήθεια ενός κανόνα, οι προϋποθέσεις του οποίου αντιστοιχούν στα υποπροβλήματα subproblem1, subproblem2, . . . , subproblem_N, με τις αντίστοιχες λίστες παραμέτρων, στα οποία έχει αναλυθεί το πρόβλημα και που αν με τη σειρά τους επιλυθούν θα έχει επιλυθεί και το αρχικό. Η ίδια λογική εφαρμόζεται για κάθε κατηγόρημα που αντιστοιχεί στα υποπροβλήματα. Το πόσο ανεξάρτητα μεταξύ τους είναι δύο από τα παραπάνω υποπροβλήματα καθορίζεται από τον αριθμό των κοινών παραμέτρων που εμφανίζονται στις αντίστοιχες λίστες παραμέτρων. Αξίζει να σημειωθεί ότι εάν δύο υποπροβλήματα είναι εντελώς ανεξάρτητα μεταξύ τους (οι δύο αντίστοιχες λίστες παραμέτρων τους δεν έχουν κανένα κοινό όρισμα) θα μπορούσαν να εκτελεστούν παράλληλα, εφόσον η υλοποίηση της Prolog που χρησιμοποιείται διέθεται αντίστοιχο μηχανισμό παράλληλης εκτέλεσης. Επίσης, εφόσον η υλοποίηση της Prolog το επιτρέπει, μπορούμε να θεωρήσουμε ότι ένα ή περισσότερα υποπροβλήματα είναι υλοποιημένα με κάποια άλλη γλώσσα προγραμματισμού (π.χ. Java) και οι αντίστοιχες κλήσεις των κατηγορημάτων που βρίσκονται στις προϋποθέσεις των παραπάνω κανόνων μετατρέπονται σε εξωτερικές κλήσεις προς τον κώδικα της γλώσσας αυτής.

Κοινές Παρανοήσεις

Πρέπει να τονίσουμε ότι η αντιστρεψιμότητα των κατηγορημάτων της Prolog δεν ισχύει πάντοτε. Πολλές φορές αυτή εξαρτάται από την σειρά των φράσεων που ορίζουν το κατηγόρημα ή από τη σειρά τοποθέτησης των προϋποθέσεων στο σώμα ενός κανόνα ή και από το αν έχουν χρησιμοποιηθεί ενσωματωμένα κατηγορήματα που από τη φύση τους δεν είναι (όλα) αντιστρέψιμα.

Παραδείγματος χάριν δεν μπορούμε, (τουλάχιστον χωρίς να ξαναγράψουμε τον κώδικα του κατηγορήματος) να χρησιμοποιήσουμε το κατηγόρημα list_sort/2 της παραγράφου 8.2 αντίστροφα και να ρωτήσουμε ποιες είναι οι αρχικές λίστες από τις οποίες προέρχεται η ταξινομημένη:

```
?- list_sort(L,[1,2,3,4]).
L = [1,2,3,4] ;
ERROR: Out of global stack
```

Ενώ θεωρητικά όλες οι μεταθέσεις των 4 στοιχείων της λίστας μπορούν να παράξουν την ταξινομημένη λίστα [1,2,3,4] το πρόγραμμα επιστρέφει (σωστά) ως πρώτη λύση την ταξινομημένη λίστα και στη συνέχεια πέφτει σε ατέρμονα βρόχο, καθώς κατά τη διάρκεια της οπισθοδρόμησης προκαλείται εκτέλεση της list_delete/3 με μη-δεσμευμένα το δεύτερο και τρίτο ορίσματα (ελεύθερα) μέσω της οποίας παράγονται συνεχώς λίστες τα στοιχεία των οποίων είναι με τη σειρά τους μη-δεσμευμένα.

Βιβλιογραφία

Ο αναγνώστης του κεφαλαίου που ενδιαφέρεται για μια πιο θεωρητική προσέγγιση στα θέματα της αντίστροφης χρήσης των κατηγορημάτων και στον ιντετερμινιστικό προγραμματισμό θα βρει χρήσιμα στοιχεία στην εργασία του Robert Kowalski, όπου αναλύει πώς η κατηγορηματική λογική μπορεί να λειτουργήσει ως γλώσσα προγραμματισμού (Kowalski, 1974) και στο βιβλίο του που πραγματεύεται την επίλυση προβλημάτων με τη βοήθεια της λογικής (Kowalski, 1979). Ο συμβολικός προγραμματισμός έχει τις ρίζες του στην εργασία του John McCarthy (McCarthy, 1960), στην οποία βασίστηκε η LISP ως πρώτη συμβολική γλώσσα προγραμματισμού της Τεχνητής Νοημοσύνης.

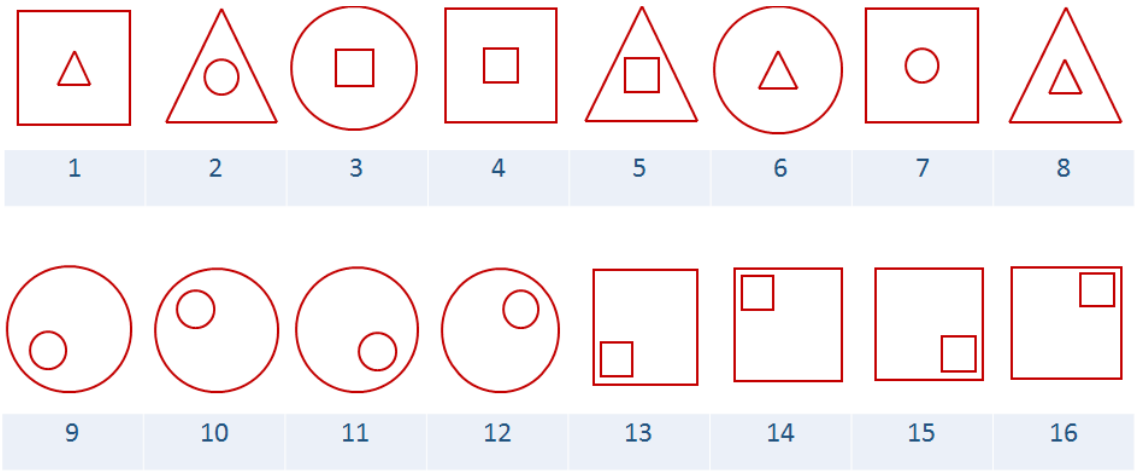
McCarthy, J. (1960) Recursive Functions of Symbolic Expressions. *Communications of the ACM*, Vol. *3* No. 4, p.p. 184-195.

Kowalski, R. (1974). Predicate Logic as a Programming Language, *Proc IFIP Cong 1974*, North-Holland Pub Co, Amsterdam, pp 569-574

Kowalski, R. (1979). *Logic for Problem Solving*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Άλυτες Ασκήσεις

8.1 Έστω ότι έχουμε να λύσουμε το πρόβλημα των αναλογιών γεωμετρικών εικόνων που συναντάμε συχνά σε τεστ νοημοσύνης και αποτελεί ένα κατεξοχήν συμβολικό πρόβλημα. Ένα σύνολο από σχετικές εικόνες είναι το παρακάτω:



Σχήμα 8.6: *Το πρόβλημα των αναλογιών*

Μια ερώτηση θα μπορούσε να είναι η εξής: "αν το σχήμα 1 σχετίζεται με το σχήμα 5, τότε ποιο σχήμα σχετίζεται με το σχήμα 3;" (απάντηση: το σχήμα 7). Η εικόνα 1 είναι ένα τρίγωνο μέσα σε ένα τετράγωνο, ενώ η εικόνα 5 είναι ένα τετράγωνο μέσα σε ένα τρίγωνο, δηλαδή τα δύο σχήματα είναι αντίστροφα. Έτσι θα πρέπει να ψάξουμε για το αντίστροφο της εικόνας 3 που είναι ένα τετράγωνο μέσα σε ένα κύκλο, δηλαδή η εικόνα 7 η οποία έχει ένα κύκλο μέσα σε ένα τετράγωνο.

Θα μπορούσαμε να αναπαραστήσουμε τις εικόνες ως γεγονότα της Prolog ως εξής:

```
figure(1,middle(triangle,square)).
```

Το παραπάνω γεγονός δηλώνει ότι η εικόνα 1 έχει ένα τρίγωνο στη μέση ενός κύκλου.

Οι σχέσεις μεταξύ των εικόνων θα μπορούσαν να αναπαρασταθούν ως εξής:

```
relation(middle(S1,S2),middle(S2,S1),inverse).
```

Το παραπάνω γεγονός δηλώνει ότι υπάρχει σχέση αντιστροφής (inverse) μεταξύ της πρώτης εικόνας η οποία περιέχει το σχήμα S1 μέσα στο σχήμα S2 και της δεύτερης εικόνας με το σχήμα S2 μέσα στο σχήμα S1.

1. Γράψτε γεγονότα για όλες τις εικόνες (1-16) που εμφανίζονται παραπάνω. (Υπόδειξη: χρησιμοποιήστε όρους όπως middle(), topleft(), bottomright(), κλπ, όπως και στην περίπτωση της εικόνας 1)
2. Γράψτε γεγονότα ή κανόνες που να περιγράφουν τις παρακάτω σχέσεις μεταξύ εικόνων (όπως για τη σχέση inverse που δόθηκε).
    - inverse: όπως αυτή παρουσιάστηκε και υλοποιήθηκε παραπάνω
    - changeout: οι δύο εικόνες διαφέρουν στο εξωτερικό τους σχήμα
    - changein: οι δύο εικόνες διαφέρουν στο εσωτερικό τους σχήμα
    - lrmirror: οι δύο εικόνες έχουν τα ίδια σχήματα μέσα και έξω αλλά το εσωτερικό σχήμα βρίσκεται π.χ. στην πρώτη εικόνα δεξιά πάνω ενώ στη δεύτερη εικόνα αριστερά πάνω (καθρεπτισμός δεξιά-αριστερά)

- tbmirror: οι δύο εικόνες έχουν τα ίδια σχήματα μέσα και έξω αλλά το εσωτερικό σχήμα βρίσκεται π.χ. στην πρώτη εικόνα δεξιά πάνω ενώ στη δεύτερη εικόνα δεξιά κάτω (καθρεπτισμός πάνω-κάτω)
- diagmirror: οι δύο εικόνες έχουν τα ίδια σχήματα μέσα και έξω αλλά το εσωτερικό σχήμα βρίσκεται π.χ. στην πρώτη εικόνα δεξιά πάνω ενώ στη δεύτερη εικόνα αριστερά κάτω (διαγώνιος καθρεπτισμός)

Με βάση τις περιγραφές σχημάτων και σχέσεων ανάμεσά τους, που δώσατε παραπάνω, ορίστε το κατηγόρημα analogy που λύνει το πρόβλημα της αναλογίας. Για παράδειγμα, η ερώτηση

```
?- analogy(1,5,3,X).
X=7.
```

σημαίνει ότι η εικόνα 1 και η εικόνα 5 είναι σχετικές και ζητάει ποια εικόνα X έχει κατ' αναλογία την ίδια σχέση με την εικόνα 3 (όπως είδαμε X=7).

ΚΕΦΑΛΑΙΟ 9: Εξωλογικά Χαρακτηριστικά της Prolog

Λέξεις Κλειδιά:
Η υπόθεση του κλειστού Κόσμου, Άρνηση ως Αποτυχία - Το κατηγόρημα not/1, Έλεγχος Εκτέλεσης: Αποκοπή, Αποκοπή και Άρνηση, Ρεύματα Εισόδου Εξόδου στην Prolog.

Περίληψη
Το κεφάλαιο παρουσιάζει τα χαρακτηριστικά της γλώσσας Prolog τα οποία αναφέρονται ως εξωλογικά, δηλαδή χαρακτηριστικά τα οποία αν και ξεφεύγουν της κατηγορηματικής λογικής πρώτης τάξης (εξω-λογικά) είναι απαραίτητα για την ανάπτυξη πρακτικών εφαρμογών. Η παρουσίαση ξεκινά με την υπόθεση του κλειστού κόσμου, η οποία επιτρέπει την υλοποίηση της άρνησης ως αποτυχία. Παρατίθενται παραδείγματα κατηγορημάτων με την χρήση του not/1, που αποτελεί το κατηγόρημα της άρνησης στην Prolog. Το κεφάλαιο συνεχίζει με το κατηγόρημα της αποκοπής, που μεταβάλλει την συμπεριφορά του μηχανισμού εκτέλεσης. Παρουσιάζονται παραδείγματα προγραμμάτων όπου η χρήση του τελεστή της αποκοπής αλλάζει την σημασιολογία τους ("κόκκινη" αποκοπή), όπως επίσης και παραδείγματα όπου η αποκοπή χωρίς να αλλάζει την σημασιολογία, βελτιώνει την ταχύτητα εκτέλεσης των προγραμμάτων ("πράσινη" αποκοπή), όπως και ο συνδυασμός αποκοπής και άρνησης. Τέλος, το κεφάλαιο κλείνει με την παρουσίαση των ρευμάτων εισόδου-εξόδου (οθόνη/αρχεία).

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός:

- Να κατανοεί την προσέγγιση της Prolog στην άρνηση ως αποτυχία.
- Να κατανοεί και να χρησιμοποιεί με επιτυχία τον τελεστή αποκοπής (cut - !) για τον έλεγχο του μηχανισμού εκτέλεσης της Prolog.
- Να μπορεί να διακρίνει μεταξύ “κόκκινων” και “πράσινων” αποκοπών.
- Να κατανοεί τους τρόπους ελέγχου της οπισθοδρόμησης και να υλοποιεί επαναληπτικές δομές βασισμένες στην αποτυχία (failure driven loops).
- Να μπορεί να διαχειριστεί πληροφορία αποθηκευμένη σε αρχεία, χρησιμοποιώντας τα κλασσικά κατηγορήματα της Prolog, και να δημιουργεί νέα αρχεία.

Παράδειγμα κίνητρο

Στο Κεφάλαιο 4, παρουσιάστηκε η κωδικοποίηση των σχέσεων ενός κοινωνικού δικτύου στον παγκόσμιο ιστό, το οποίο βασίζεται σε σχέσεις φιλίας που δημιουργούνται μεταξύ χρηστών, όταν ο ένας ακολουθεί (follows) τον άλλο. Οι χρήστες έχουν αποθηκευμένα στοιχεία του προφίλ τους, όπως για παράδειγμα το φύλλο (male/female), την ηλικία, το επάγγελμα. Βάσει των πληροφοριών των χρηστών, ήταν δυνατό να οριστεί ένα πλήθος σχέσεων. Έστω ότι θέλουμε το πρόγραμμα να προτείνει στον χρήστη νέους φίλους, δηλαδή χρήστες που ήδη τον ακολουθούν και θα έπρεπε να τους ακολουθήσει και ο ίδιος. Η νέα αυτή σχέση μπορεί να γραφεί απλά:

```
recommend_new_friend(User,Followee):-
  follows(Followee,User).
```

Όμως η παραπάνω κωδικοποίηση εμφανίζει ένα σημαντικό πρόβλημα, καθώς προτείνει και χρήστες τους οποίους ήδη ακολουθεί ο χρήστης User:

```
?- recommend_new_friend(ilias,Followee).
Followee = petros ;
Followee = nikos.
```

Ο χρήστης ilias ήδη ακολουθεί τον χρήστη petros, οπότε δεν θα έπρεπε να προτείνεται. Για να επιτευχθεί η επιθυμητή συμπεριφορά, είναι απαραίτητο να κωδικοποιήσουμε "αρνητική" πληροφορία, δηλαδή την μη αλήθεια της σχέσης follows/2, χρησιμοποιώντας για παράδειγμα ένα κατηγόρημα της μορφής:

```
does_not_follow(ilias,nikos).
```

Το κατηγόρημα θα επέτρεπε να ελεγχθεί ότι ο χρήστης δεν ακολουθεί ήδη τον προτεινόμενο χρήστη:

```
recommend_new_friend(User,Followee):-
  follows(Followee,User),
  does_not_follow(User,Followee).
```

Αν και η παραπάνω προσέγγιση λύνει το πρόβλημα, παρουσιάζεται το σημαντικό μειονέκτημα ότι πρέπει ρητά να κωδικοποιήσουμε και τις "αρνητικές" σχέσεις, με αποτέλεσμα η συντήρηση του προγράμματος να γίνεται περισσότερο πολύπλοκη, και εξαιρετικά δύσκολη σε μεγάλα προγράμματα. Για το λόγο αυτό η άρνηση στην Prolog, όπως θα δούμε στις επόμενες παραγράφους ακολουθεί την υπόθεση του κλειστού κόσμου.

Υπάρχει ένα πλήθος λειτουργιών που θα θέλαμε να υλοποιήσουμε στο κοινωνικό δίκτυο, όπως για παράδειγμα να μπορούμε να βρούμε τον νεότερο/γηραιότερο χρήστη, να τυπώνουμε τα ονόματα των χρηστών στην οθόνη και να αποθηκεύουμε πληροφορίες του δικτύου σε αρχεία. Οι παραπάνω υλοποιήσεις πολλές φορές απαιτούν χαρακτηριστικά τα οποία δεν εντάσσονται στα πλαίσια της κλασικής κατηγορηματικής λογικής, αλλά κάνουν την γλώσσα περισσότερο πρακτική. Αυτά τα εξω-λογικά χαρακτηριστικά της γλώσσας πραγματεύεται το παρόν κεφάλαιο.

Μονοπάτι Μάθησης

Τα βασικά στοιχεία για το κεφάλαιο αυτό βρίσκονται στις ενότητες 9.1 έως και 9.3 οι οποίες πρέπει να ολοκληρωθούν από τον αναγνώστη. Οι ενότητες περιλαμβάνουν την βασική παρουσίαση της άρνησης στην Prolog, τον έλεγχο του μηχανισμού οπισθοδρόμησης και την διαχείριση των ρευμάτων εισόδου εξόδου . Η ενότητες 9.4 και 9.5 παρουσιάζουν δύο πιο ολοκληρωμένα παραδείγματα πάνω στις έννοιες που πραγματεύεται το κεφάλαιο. Η ενότητα 9.6 περιέχει επιπρόσθετα στοιχεία που δεν είναι όμως απαραίτητα, τουλάχιστον σε μια πρώτη ανάγνωση. Τέλος η ενότητα 9.7 περιέχει κάποιες σημαντικές παρατηρήσεις που αφορούν το κατηγόρημα της αποκοπής και θα πρέπει να περιληφθεί στην πρώτη ανάγνωση.

9.1 Άρνηση ως Αποτυχία

Μια από τις σημαντικότερες διαφορές της Prolog, από την κλασική κατηγορηματική λογική είναι η σημασία που αποδίδεται στην άρνηση, δηλαδή στην απόδειξη των αρνητικών προτάσεων. Η Prolog υιοθετεί την "**υπόθεση του κλειστού κόσμου**", που απλά διατυπώνεται ως "ότι δεν είναι δυνατό να αποδειχθεί αληθές, θεωρείται ψευδές". Έτσι αν μια προς απόδειξη πρόταση δεν μπορεί να αποδειχθεί βάσει των λογικών προτάσεων του Prolog προγράμματος (γεγονότα και κανόνες), τότε αυτόματα συνάγεται το συμπέρασμα ότι είναι ψευδής.

Για παράδειγμα, έστω τα ακόλουθα γεγονότα, από το παράδειγμα του κοινωνικού δικτύου του Κεφαλαίου 4:

```
male(petros).
male(ilias).
male(demos).
male(nikos).
```

Στην ερώτηση male(alex) η Prolog θα απαντήσει με false, δηλαδή ότι η προς απόδειξη πρόταση είναι ψευδής, καθώς δεν μπορεί να αποδείξει την αλήθεια της πρότασης, ταιριάζοντας την με κάποια από τα γεγονότα του προγράμματος.

Στην Prolog, η απόδειξη της άρνησης μιας πρότασης που περιγράφηκε παραπάνω γίνεται μέσω του ειδικού κατηγορήματος not/1, το οποίο δέχεται ως όρισμα μια οποιοδήποτε Prolog ερώτηση. Η σημασιολογία του είναι απλή: αν η ερώτηση που εμφανίζεται στο όρισμα του κατηγορήματος μπορεί να αποδειχθεί, τότε το not/1 αποτυγχάνει, αν δεν μπορεί να αποδειχθεί τότε το not/1 επιτυγχάνει. Για παράδειγμα:

```
?-male(alex)
false
?-not(male(alex))
true
```

Θα πρέπει να τονίσουμε εδώ ότι στο όρισμα του κατηγορήματος, μπορεί να εμφανίζεται οποιαδήποτε ερώτηση. Για παράδειγμα, η ακόλουθη ερώτηση επιτυγχάνει καθώς το 15 δεν είναι μέλος της λίστας.

```
?- not(member(15,[10,20,30])).
true.
```

Ιδιαίτερη προσοχή θα πρέπει να δίνεται όταν το όρισμα του not/1 περιέχει μη δεσμευμένες (ελεύθερες) μεταβλητές. Για παράδειγμα, ένα συνηθισμένο "λάθος" εντοπίζεται στην ακόλουθη ερώτηση:

```
?- not(X = 10)
false
```

Είναι σύνηθες οι προγραμματιστές που έρχονται πρώτη φορά σε επαφή με την Prolog, να αναμένουν η παραπάνω ερώτηση να αποτιμηθεί ως αληθής. Αυτό οφείλεται στην λανθασμένη θεώρηση της σημασίας του τελεστή "=", ο οποίος είναι ο τελεστής ενοποίησης. Κατά συνέπεια, η ερώτηση X=10, διατυπώνει το ερώτημα "αν η μεταβλητή X μπορεί να ενοποιηθεί με τη σταθερά 10", στη οποία η απάντηση είναι προφανώς θετική (και η πλήρης ερώτηση αποτιμάται ως αληθής). Παρόμοιο "λάθος", προκύπτει και στην ερώτηση:

```
?-not(male(X)).
false
```

στην οποία η απάντηση είναι αρνητική, καθώς υπάρχει κάποια τιμή για την μεταβλητή X, η οποία να κάνει την ερώτηση male(X) αληθή, για παράδειγμα η σταθερά petros, και κατά συνέπεια η άρνησή της είναι ψευδής. Προσοχή θα πρέπει να δοθεί εδώ στο γεγονός ότι η μεταβλητή δεν δεσμεύεται σε κάποια τιμή, αλλά παραμένει ελεύθερη. Αυτή η ιδιότητα θα χρησιμοποιηθεί στην τεχνική της διπλής άρνησης που αναφέρεται στο τέλος του κεφαλαίου, στην ενότητα 9.6.

Με χρήση του κατηγορήματος not/1, είναι προφανές ότι η σχέση recommend_new_friend/2 γράφεται:

```
recommend_new_friend(User,Followee):-
  follows(Followee,User),
  not(follows(User,Followee)).
```

Ο δεύτερος υποστόχος, ο οποίος διατυπώνεται χρησιμοποιώντας την άρνηση, εξασφαλίζει ότι δεν θα προτείνονται στο χρήστη άλλοι χρήστες οι οποίοι είναι ήδη φίλοι του. Έτσι η συμπεριφορά του κατηγορήματος είναι:

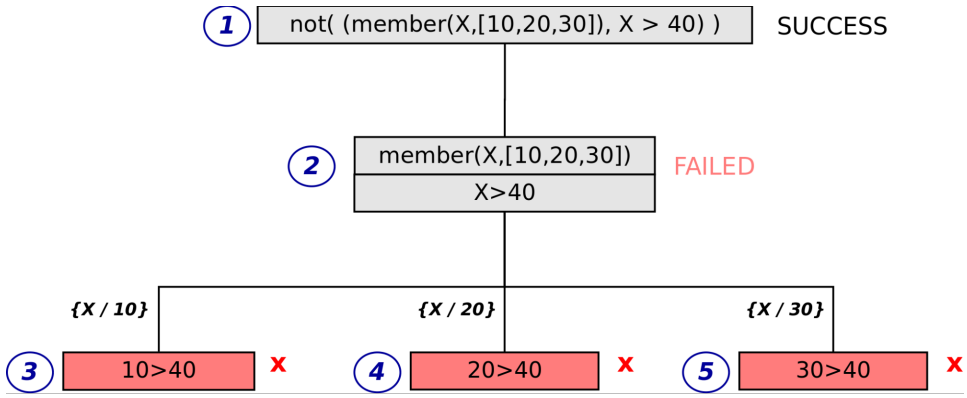
```
?- recommend_new_friend(ilias,Followee).
Followee = nikos.
```

Ενδιαφέρον παρουσιάζουν και σύνθετες ερωτήσεις, δηλαδή ερωτήσεις που είναι συζεύξεις στόχων, οι οποίες όμως καθώς το not/1 δέχεται μόνο ένα όρισμα, θα πρέπει να περικλείονται σε παρενθέσεις. Για παράδειγμα, η ακόλουθη ερώτηση:

```
?- not( (member(X,[10,20,30]),X > 40) ).
true
```

είναι αληθής, καθώς δεν υπάρχει στοιχείο που να είναι μέλος της λίστας και ταυτόχρονα (σύζευξη) να είναι μεγαλύτερο του 40. Διατυπώνοντας το παραπάνω λίγο διαφορετικά, η ερώτηση εξασφαλίζει ότι όλα τα στοιχεία της λίστας είναι μικρότερα του 40! Πώς όμως εξασφαλίζεται ότι κάθε στοιχείο της λίστας ικανοποιεί την συνθήκη; Η απάντηση βρίσκεται στο μηχανισμό οπισθοδρόμησης της Prolog.

Όπως είδαμε στο Κεφάλαιο 7, όταν το πρώτο όρισμα του κατηγορήματος member/2 είναι ελεύθερη μεταβλητή, τότε ο μηχανισμός εκτέλεσης δεσμεύει διαδοχικά την μεταβλητή στα στοιχεία της λίστας κατά την οπισθοδρόμηση. Η τελευταία προκαλείται λόγω της αποτυχίας της συνθήκης X>40. Περισσότερο αναλυτικά, η μεταβλητή X παίρνει αρχικά την τιμή 10 και ο στόχος 10>40, αποτυγχάνει προκαλώντας οπισθοδρόμηση (βήμα 3, Σχήμα 9.1). Η ύπαρξη εναλλακτικών λύσεων για το member/2, δίνει την τιμή 20 στη μεταβλητή, που επίσης οδηγεί σε αποτυχία του στόχου 20>40 (βήμα 4), κοκ. Όταν όλες οι εναλλακτικές λύσεις του member(X,[10,20,30]) εξαντληθούν, τότε η σύζευξη αποτυγχάνει, με αποτέλεσμα η ερώτηση να αποτιμάται ως αληθής λόγω του not/1. Το δένδρο αναζήτησης του συγκεκριμένου παραδείγματος φαίνεται στο Σχήμα 9.1. Οι αριθμοί στο σχήμα δηλώνουν τη σειρά δημιουργίας των κόμβων.



Σχήμα 9.1: *Δένδρο αναζήτησης της ερώτησης not((member(X,[10,20,30]),X>40)).*

Μέγιστο Στοιχείο Λίστας

Η χρήση του not/1 είναι ένα αρκετά ισχυρό εργαλείο σε αρκετές περιπτώσεις. Για παράδειγμα, έστω το κατηγόρημα list_max/2, που εξετάστηκε στο Κεφάλαιο 7, το οποίο επιστρέφει το μέγιστο στοιχείο μιας λίστας αριθμών. Ίσως ο πλέον δηλωτικός ορισμός του μέγιστου μιας λίστας να είναι ο ακόλουθος:

- Το μέγιστο στοιχείο μια λίστας, είναι ένα μέλος της λίστας τέτοιο ώστε δεν υπάρχει (άλλο) μέλος της το οποίο να είναι μεγαλύτερο.

Η υλοποίηση του παραπάνω ορισμού σε Prolog είναι:

```
list_max_n(Max,List):-
  member(Max,List),
  not((member(X,List), X > Max)).
```

Ο πρώτος υποστόχος member(Max,List) εξασφαλίσει ότι η μεταβλητή Max θα πάρει μια τιμή από τη λίστα των αριθμών. Ο δεύτερος υποστόχος, επιβάλει απλά να μην υπάρχει στοιχείο της λίστας το οποίο να είναι μεγαλύτερο. Για παράδειγμα

```
?- list_max_n(X,[2,3,1]).
X = 3 .
```

Όπως στην σύνθετη ερώτηση της προηγούμενης ενότητας, στην περίπτωση αυτή η λειτουργία του κατηγορήματος βασίζεται στον μηχανισμό οπισθοδρόμησης, ο οποίος σε περίπτωση αποτυχίας εξετάζει τις εναλλακτικές λύσεις. Έτσι, ο πρώτος υποστόχος δεσμεύει αρχικά την μεταβλητή Max στην πρώτη τιμή της λίστας και ο δεύτερος ελέγχει αν υπάρχει κάποια τιμή στη λίστα αυστηρά μεγαλύτερη αυτής. Αν υπάρχει τέτοια τιμή στη λίστα, τότε ο δεύτερος υποστόχος αποτυγχάνει. Η αποτυχία αυτή οδηγεί σε οπισθοδρόμηση, με την μεταβλητή Max να δεσμεύεται στην επόμενη τιμή και να γίνεται ξανά ο έλεγχος. Η διαδικασία επαναλαμβάνεται μέχρι η Max να πάρει μια τιμή τέτοια ώστε δεν υπάρχει (αυστηρά) μεγαλύτερη αυτής στη λίστα, άρα τη μέγιστη τιμή.

Το not/1 ανήκει στα μετα-λογικά κατηγορήματα ανώτερης τάξης, μια ειδική κατηγορία κατηγορημάτων, τα οποία θα εξεταστούν διεξοδικότερα στο Κεφάλαιο 10, στο οποίο θα δοθεί και η υλοποίηση του συγκεκριμένου κατηγορήματος.

9.2 Έλεγχος Οπισθοδρόμησης

Στο Κεφάλαιο 5 παρουσιάστηκε ο μηχανισμός εκτέλεσης της Prolog, ο οποίος είναι ουσιαστικά αναζήτηση κατά βάθος στο δένδρο απόδειξης, της απάντησης στην ερώτηση που τέθηκε. Καθώς το δένδρο αυτό μπορεί να γίνει εξαιρετικά σημαντικό σε μέγεθος, ο παραπάνω μηχανισμός εκτέλεσης σε περιπτώσεις μεγάλων υπολογισμών μπορεί να απαιτήσει σημαντικό χρόνο εκτέλεσης για την εύρεση της απάντησης, είτε εξερευνώντας κλαδιά τα οποία δεν μπορούν να οδηγήσουν σε λύση, είτε επιστρέφοντας λύσεις οι οποίες δεν ενδιαφέρουν. Για παράδειγμα, έστω ότι θέλουμε να εξετάσουμε αν ένα συγκεκριμένο στοιχείο είναι μέλος

μιας λίστας, η οποία περιέχει πολλαπλές εμφανίσεις των στοιχείων της. Σύμφωνα με όσα έχουμε πει στο αντίστοιχο κεφάλαιο (Κεφάλαιο 7), η ερώτηση που εξασφαλίζει την απάντηση στο παραπάνω είναι:

```
?- member(a,[a,b,c,a,a,b,b,c]).
true ;
true ;
true ;
false.
```

Όπως φαίνεται παραπάνω, κατά την οπισθοδρόμηση το κατηγόρημα member/2 θα επιτύχει τρεις φορές, όσες και οι εμφανίσεις του στοιχείου a στην λίστα του δεύτερου ορίσματος, πριν εξαντλήσει το δένδρο αναζήτησης και επιστρέψει false, δηλαδή ότι δεν υπάρχουν άλλες λύσεις. Σε κάποιες περιπτώσεις μια τέτοια συμπεριφορά μπορεί να αποτελέσει πρόβλημα. Έστω το κατηγόρημα commom_element/3, το οποίο δεσμεύει το πρώτο όρισμά του, σε ένα κοινό στοιχείο μεταξύ δύο λιστών (Κεφάλαιο 8). Ο ορισμός του κατηγορήματος βασίζεται στο κατηγόρημα member/2 και είναι ο ακόλουθος:

```
common_element(X,ListA,ListB):-
  member(X,ListA),
  member(X,ListB).
```

Στην ερώτηση που ακολουθεί, η Prolog επιστρέφει περισσότερες λύσεις από ότι θα ήταν επιθυμητό:

```
?- common_element(X,[a,b,c],[a,a,b,b,c,c,c]).
X = a ;
X = a ;
X = b ;
X = b ;
X = c ;
X = c ;
X = c.
```

Άρα σε κάποιες περιπτώσεις είναι απαραίτητο να υπάρχει μεγαλύτερος έλεγχος στον μηχανισμό οπισθοδρόμησης. Αυτό επιτυγχάνεται με τα κατηγορήματα ελέγχου εκτέλεσης, δηλαδή τα κατηγορήματα της αποκοπής, επιτυχίας και αποτυχίας που θα εξεταστούν στις επόμενες ενότητες.

Το κατηγόρημα της Αποκοπής

Το πλέον ισχυρό ενσωματωμένο κατηγόρημα ελέγχου της οπισθοδρόμησης στην Prolog, είναι το κατηγόρημα της **αποκοπής** (cut). Η λειτουργία του είναι να μειώνει το δένδρο αναζήτησης κλαδεύοντας δυναμικά, δηλαδή κατά την ώρα της εκτέλεσης, κλαδιά (μονοπάτια αναζήτησης) τα οποία γνωρίζουμε ότι δεν οδηγούν σε λύση.

Το κατηγόρημα συμβολίζεται ένα θαυμαστικό (!) και εμφανίζεται σαν κανονικός υποστόχος στο σώμα ενός κανόνα, ο οποίος πετυχαίνει πάντα. Η επίδραση της αποκοπής στο μηχανισμό εκτέλεσης είναι να "παγώνει" (freeze) όσες επιλογές έχουν γίνει πριν την εκτέλεση της, για το κατηγόρημα μέσα στο οποίο εμφανίζεται, δηλαδή υποχρεώνει το μηχανισμό οπισθοδρόμησης να αγνοήσει τις όποιες εναλλακτικές λύσεις υπήρχαν για το κατηγόρημα και βρίσκονται αριστερά της αποκοπής.

Στη γενική περίπτωση, έστω το κατηγόρημα p, του οποίου οι εναλλακτικοί κανόνες p_1, p_2, .. p_n φαίνονται παρακάτω, με b_{ij} τους υποστόχους κάθε κανόνα:

```
p1:-b11,b12,...b1k.
...
pq:-bq1,bq2,...bqj,!,bq(j+1)...bql.
p(q+1):- b(q+1)1,...b(q+1)p.

...
pn:-bn1,bn2,..bnm.
```

Αν η εκτέλεση φτάσει να αποδείξει τον υποστόχο b_{qj}, θα εκτελεστεί η αποκοπή και η εκτέλεση δεσμεύεται σε όσες επιλογές (choices) έγιναν για τα b_{q1}, b_{q2}, έως και b_{qj} και το p, δηλαδή όσα σημεία επιλογής (choice points) υπάρχουν στα αριστερά του κατηγορήματος της αποκοπής απορρίπτονται από τον μηχανισμό εκτέλεσης. Αυτό σημαίνει ότι οι πιθανές εναλλακτικές λύσεις οι οποίες θα προέκυπταν από τα b_{q1}, b_{q2}, έως και b_{qj} και οι εναλλακτικές λύσεις για το p, δηλαδή οι λύσεις που θα προέκυπταν από τα p_{q+1}, … p_n αγνοούνται.

Είναι σημαντικό να γίνει κατανοητό ότι η επίδραση της αποκοπής στο μηχανισμό εκτέλεσης γίνεται δυναμικά, δηλαδή όταν και εάν "εκτελεστεί" το κατηγόρημα της αποκοπής.

Έστω ένα απλό κατηγόρημα a/1 το οποίο αποτελείται από δύο γεγονότα της μορφής:

```
a(1).
a(2).
```

και το κατηγόρημα b/1:

```
b(X):-a(X).
b(3).
```

Είναι προφανές ότι στην ερώτηση b(X) η Prolog θα επιστρέψει τρεις λύσεις:

```
?- b(X).
X = 1 ;
X = 2 ;
X = 3.
```

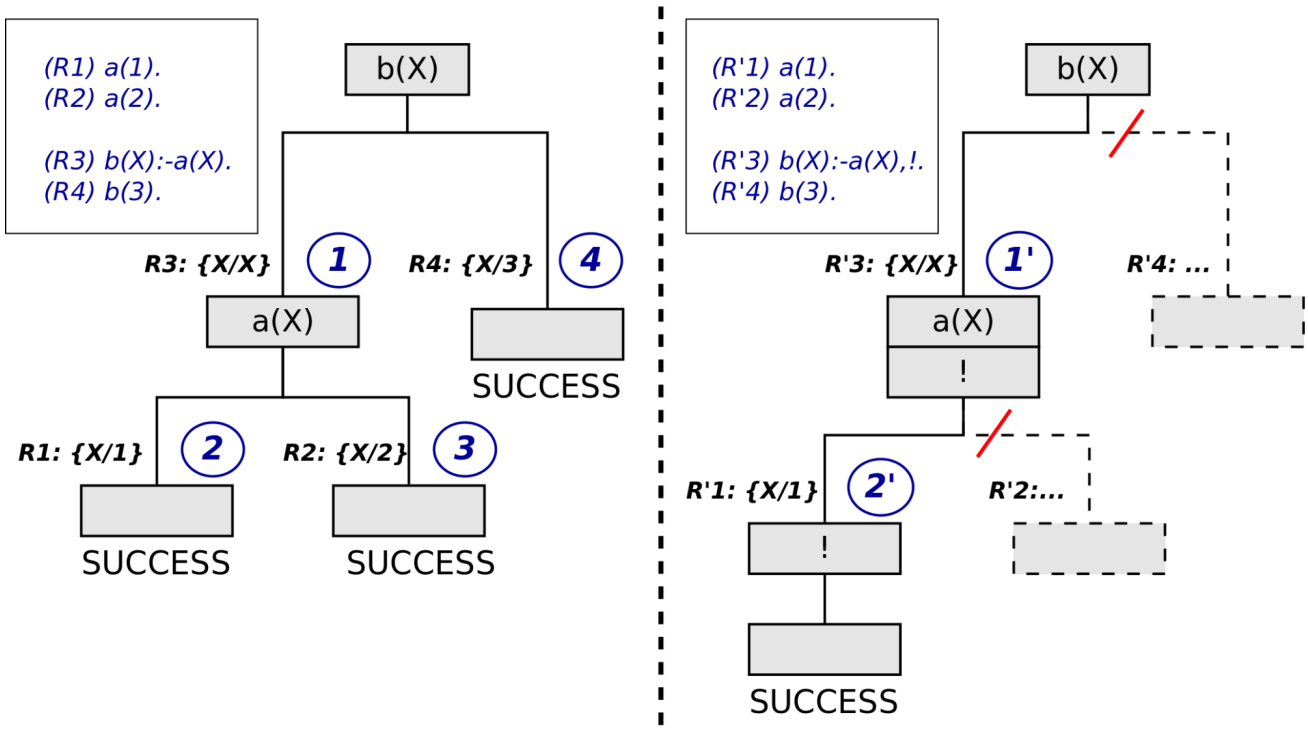
Αν στην πρώτη πρόταση του κατηγορήματος b/1 εισάγουμε σαν τελευταίο υποστόχο το κατηγόρημα της αποκοπής:

```
b(X):-a(X),!.
```

Η ερώτηση b(X) στην δεύτερη αυτή περίπτωση επιστρέφει μια και μοναδική λύση:

```
?- b(X).
X = 1.
```

Αυτό συμβαίνει γιατί ο μηχανισμός εκτέλεσης της Prolog αποδεικνύει τον υποστόχο a(X) για X = 1, και έπειτα εκτελεί το κατηγόρημα της αποκοπής, με αποτέλεσμα οι εναλλακτικές λύσεις για το κατηγόρημα a/1 αλλά και για το b/1 μέσα στο οποίο εμφανίζεται η αποκοπή να μην λαμβάνονται υπόψη. Στο Σχήμα 9.2 παρουσιάζεται η επίδραση της αποκοπής στο δένδρο αναζήτησης για την συγκεκριμένη ερώτηση.



Σχήμα 9.2: *Η επίδραση της αποκοπής στο δένδρο αναζήτησης.*

Όπως φαίνεται στο Σχήμα 9.2, μετά την απόδειξη του υποστόχου a(X) στο βήμα 1', εκτελείται η αποκοπή, η οποία πετυχαίνει πάντα (βήμα 2') και κλαδεύει τα εναλλακτικά μονοπάτια αναζήτησης, δηλαδή λύσεις που θα προέκυπταν από τις προτάσεις R'2 και R'4.

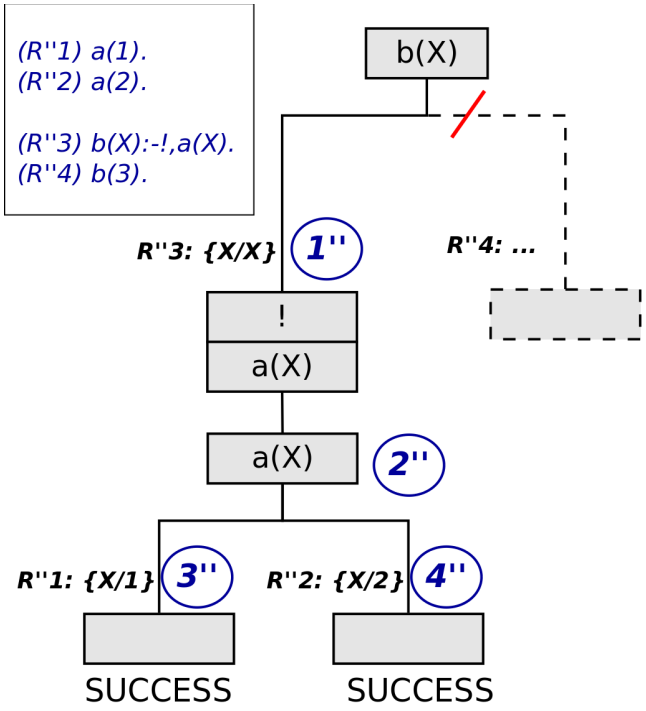
Έστω τώρα ότι εισάγουμε την αποκοπή στην πρώτη πρόταση του κατηγορήματος πριν από τον υποστόχο a(X), δηλαδή έχουμε την πρόταση:

```
b(X):-!,a(X).
```

Στην ερώτηση b(X), η Prolog θα επιστρέψει τις ακόλουθες λύσεις:

```
?- b(X).
X = 1 ;
X = 2.
```

Εφόσον η αποκοπή εκτελέστηκε πριν την απόδειξη του υποστόχου a(X), δεν "κλάδεψε" τις εναλλακτικές λύσεις οι οποίες προκύπτουν από αυτόν, όπως φαίνεται στο Σχήμα 9.3, αλλά μόνο την λύση που προέκυπτε από την δεύτερη πρόταση του κατηγορήματος b(X) (βήμα 1'', σχήμα 9.3). Από τα παραδείγματα είναι προφανές ότι η αποκοπή επηρεάζει τα εναλλακτικά μονοπάτια αναζήτησης που βρίσκονται πριν (στα αριστερά) της, και όχι τα μονοπάτια τα οποία παράγονται από υποστόχους που βρίσκονται μετά την αποκοπή.



Σχήμα 9.3: *Η επίδραση της αποκοπής στους στόχους που βρίσκονται στα δεξιά της.*

Ας εξετάσουμε ένα περισσότερο πολύπλοκο παράδειγμα. Έστω, τα ακόλουθα γεγονότα της Prolog:

```
letter(a).
letter(b).

alpha(c).
alpha(d).

num(1).
num(2).

roman(i).
roman(ii).
```

και το κατηγόρημα p(X,Y):

```
p(X,Y):- letter(X),num(Y).
p(X,Y):- alpha(X),roman(Y).
p(e,1).
```

Για την ερώτηση p(X,Y), το παραπάνω κατηγόρημα επιστρέφει 9 λύσεις (Πίνακας 9.1), που προκύπτουν από τις τρεις παραπάνω προτάσεις, ως ακολούθως:

- Η πρώτη πρόταση επιστρέφει 4 λύσεις, από τον συνδυασμό των κατηγορημάτων letter/1 και num/1, οι οποίες προκύπτουν από το συνδυασμό των δύο λύσεων του letter/1 και των δύο λύσεων του num/1 (λύσεις 1,2,3 και 4 στον πίνακα 9.1).
- Η δεύτερη πρόταση επιστρέφει 4 λύσεις, από τον συνδυασμό των κατηγορημάτων alpha/1 και roman/1, με παρόμοιο τρόπο (λύσεις 5,6,7 και 8 του πίνακα 9.1).
- Τέλος, μία λύση επιστρέφεται από την τελευταία πρόταση (λύση 9).

Το κατηγόρημα p_with_cut(X,Y) είναι πανομοιότυπο με το p(X,Y), με την διαφορά ότι στο σώμα του δεύτερου κανόνα εμφανίζεται το κατηγόρημα της αποκοπής.

```
p_with_cut(X,Y):- letter(X),num(Y).
p_with_cut(X,Y):- alpha(X),!,roman(Y).
p_with_cut(e,1).
```

Ποια είναι όμως η διαφορά που κάνει το κατηγόρημα της αποκοπής; Ο πίνακας 9.1 δίνει τις λύσεις για τα αντίστοιχα ερωτήματα p(X,Y) και p_with_cut(X,Y).

Πίνακας 9.1: *Λύσεις στα ερωτήματα p(X,Y) και p_with_cut(X,Y).*

	`?- p(X,Y).`	`?- p_with_cut(X,Y).`
(1)	`X = a, Y = 1 ;`	`X = a, Y = 1 ;`
(2)	`X = a, Y = 2 ;`	`X = a, Y = 2 ;`
(3)	`X = b, Y = 1 ;`	`X = b, Y = 1 ;`
(4)	`X = b, Y = 2 ;`	`X = b, Y = 2 ;`
(5)	`X = c, Y = i ;`	`X = c, Y = i ;`
(6)	`X = c, Y = ii ;`	`X = c, Y = ii.`
(7)	`X = d, Y = i ;`	
(8)	`X = d, Y = ii ;`	
(9)	`X = e, Y = 1.`	

Όπως φαίνεται στον πίνακα 9.1, η εναλλακτική λύση του υποστόχου alpha/1 με δέσμευση της μεταβλητής X στην τιμή d καθώς και η εναλλακτική λύση που υπήρχε για το κατηγόρημα p_with_cut/2 από την τρίτη πρόταση, δεν επιστρέφονται από την Prolog. Έτσι οι λύσεις είναι έξι συνολικά:

- τέσσερις από την πρώτη πρόταση, όπως και στο κατηγόρημα p/2,
- δύο από την δεύτερη πρόταση, καθώς ο υποστόχος alpha/1, πετυχαίνει μόνο για X ίσο με c, (αριστερά της αποκοπής), ενώ ο υποστόχος roman/1, εφόσον είναι δεξιά της αποκοπής, διατηρεί τις δύο λύσεις,
- καμία λύση από την τρίτη πρόταση, καθώς έχει εκτελεστεί η αποκοπή.

Συχνό λάθος είναι να θεωρηθεί, από την παραπάνω ανάλυση ότι η τρίτη πρόταση του κατηγορήματος είναι άχρηστη, καθώς δεν μπορεί ποτέ να επιστρέψει λύση. Η ακόλουθη ερώτηση αποδεικνύει το αντίθετο:

```
?- p_with_cut(e,Y).
Y = 1.
```

Η παραπάνω λύση προκύπτει από την τρίτη πρόταση του κατηγορήματος. Στην περίπτωση αυτή, η αποκοπή της δεύτερης πρότασης, δεν εκτελέστηκε, καθώς ο υποστόχος alpha(e) απέτυχε πριν να φτάσει η εκτέλεση

στο σημείο της αποκοπής, άρα η εναλλακτική λύση που προκύπτει από τον τρίτο κανόνα του κατηγορήματος διατηρήθηκε.

Η χρήση της αποκοπής έχει σημαντικά πλεονεκτήματα, με κύριο το ότι αυξάνει την εκφραστική ικανότητα της γλώσσας, επιτρέποντας την ανάπτυξη προγραμμάτων τα οποία δεν θα μπορούσαν να αναπτυχθούν χωρίς αυτή, όπως για παράδειγμα η υλοποίηση της άρνησης που θα εξεταστεί στο Κεφάλαιο 10 και ο εξαναγκασμός του συστήματος σε αποτυχία. Επίσης, η σωστή χρήση της επιτρέπει να υλοποιηθεί κώδικας ο οποίος είναι αποδοτικότερος τόσο σε χρόνο εκτέλεσης όσο και σε απαιτήσεις σε μνήμη, καθώς επιτρέπει την αποφυγή ατέρμονων αναζητήσεων, περικοπή λύσεων που πλεονάζουν και η παράλειψη άσκοπων ελέγχων. Στα επόμενα εξετάζονται αυτές περιπτώσεις.

Περικοπή λύσεων που πλεονάζουν

Θα εξετάσουμε την περίπτωση της περικοπής λύσεων οι οποίες πλεονάζουν, μέσω του κατηγορήματος common_element/3 που παρουσιάστηκε ως παράδειγμα στην αρχή της ενότητας, και επιστρέφει τα κοινά στοιχεία μεταξύ δύο λιστών. Μια πρώτη (λανθασμένη) προσέγγιση στο κατηγόρημα είναι να εισαχθεί σαν τελευταίος υποστόχος του κατηγορήματος η αποκοπή:

```
common_element_a(X,ListA,ListB):-
  member(X,ListA),
  member(X,ListB),!.
```

Στην ερώτηση που ακολουθεί, η Prolog επιστρέφει μια και μοναδική λύση, καθώς όλες οι εναλλακτικές λύσεις των δύο υποστόχων member/2, αποκλείονται εφόσον εμφανίζονται πριν (στα αριστερά) της αποκοπής:

```
?- common_element_a(X,[a,b,c],[a,a,b,b,c,c,c]).
X = a.
```

Η κατάσταση ελάχιστα βελτιώνεται αν εισάγουμε την αποκοπή "ανάμεσα" στους δύο στόχους:

```
common_element_b(X,ListA,ListB):-
  member(X,ListA),
  !,
  member(X,ListB).
```

όπου για την ίδια ερώτηση, η Prolog επιστρέφει δύο φορές την πρώτη λύση και αγνοεί τις υπόλοιπες, καθώς αποκλείστηκαν οι εναλλακτικές λύσεις του πρώτου υποστόχου member/2, αλλά διατηρήθηκαν οι λύσεις για τον δεύτερο υποστόχο member/2:

```
?- common_element_b(X,[a,b,c],[a,a,b,b,c,c,c]).
X = a ;
X = a ;
false.
```

Είναι προφανές ότι αυτό το οποίο είναι απαραίτητο είναι ένα κατηγόρημα που θα επιστρέφει το μέλος μιας λίστας μόνο μια φορά. Αυτό υλοποιείται από το κατηγόρημα member_check/2:

```
member_check(X,[X|_]):-!.
member_check(X,[_|Rest]):-
  member_check(X,Rest).
```

Η αποκοπή που εμφανίζεται ως μοναδικός υποστόχος του πρώτου κανόνα, αποκλείει την επιτυχία του κατηγορήματος πολλαπλές φορές για συγκεκριμένο X, δηλαδή αν κληθεί το κατηγόρημα με την μεταβλητή X δεσμευμένη σε μια τιμή τότε το κατηγόρημα θα επιτύχει μόνο μια φορά, όσες φορές και αν εμφανίζεται η τιμή αυτή στην λίστα του δεύτερου ορίσματος:

```
?- member_check(a,[a,a,b,b,c,c,c,c]).
true.
?- member_check(c,[a,a,b,b,c,c,c,c]).
true.
```

Τι γίνεται όμως αν η μεταβλητή X είναι ελεύθερη; Στην περίπτωση αυτή επιστρέφεται μόνο το πρώτο στοιχείο της λίστας, όπως φαίνεται από την ακόλουθη ερώτηση:

```
?- member_check(X,[a,a,b,b,c,c,c,c]).
X = a.
```

```
?- member_check(X,[a,b,c]).
X = a.
```

Κατά συνέπεια, η εισαγωγή της αποκοπής αλλάζει σημαντικά την συμπεριφορά του κατηγορήματος, καθώς είτε επιστρέφει το πρώτο στοιχείο είτε πετυχαίνει μόνο μια φορά. Εκτός από την αποφυγή περιττών λύσεων, το κατηγόρημα member_check/2 έχει και σημαντικά καλύτερη απόδοση σε χρόνο, καθώς η εκτέλεση σταματά στην πρώτη εμφάνιση του στοιχείου στη λίστα.

Έτσι, η τελική - σωστή έκδοση του προγράμματος είναι η ακόλουθη:

```
common_element_c(X,ListA,ListB):-
  member(X,ListA),
  member_check(X,ListB).
```

Ο πρώτος υποστόχος member/2 διατηρείται ώστε να μπορέσει να διατρέξει την πρώτη λίστα, ενώ ο δεύτερος αλλάζει σε member_check/2, ώστε να πετυχαίνει μια φορά για κάθε στοιχείο της λίστας του πρώτου ορίσματος. Το κατηγόρημα έχει την ακόλουθη συμπεριφορά:

```
?- common_element_c(X,[a,b,c],[a,a,b,b,c,c,c]).
X = a ;
X = b ;
X = c.
```

Παράλειψη Άσκοπων Ελέγχων

Μια από τις πλέον διαδεδομένες χρήσεις της αποκοπής, είναι η απαλοιφή ελέγχων που πλεονάζουν, όταν είναι γνωστό εκ των προτέρων ότι μόνο μια από τις εναλλακτικές προτάσεις του κατηγορήματος μπορεί να επιτύχει. Έστω το κατηγόρημα expression_sign/2, το οποίο ενοποιεί το δεύτερο όρισμά του σε μια λεκτική περιγραφή του πρόσημου της τιμής του πρώτου ορίσματος. Με άλλα λόγια “επιστρέφει" στο δεύτερο όρισμα positive αν το πρώτο όρισμα είναι θετικός αριθμός, negative αν είναι αρνητικός, ή zero αν είναι μηδέν. Το κατηγόρημα υλοποιείται απλά με την μορφή τριών αμοιβαία αποκλειόμενων κανόνων:

```
expression_sign(X,positive):- X > 0.
expression_sign(X,zero):- X =:=0.
expression_sign(X,negative):-X < 0.
```

Η συμπεριφορά του κατηγορήματος είναι η ακόλουθη:

```
?- expression_sign(2,S).
S = positive ;
false.
?- expression_sign(0,S).
S = zero ;
false.
?- expression_sign(-5,S).
S = negative.
```

Η συμπεριφορά του κατηγορήματος είναι απόλυτα σωστή. Όμως, στην πρώτη περίπτωση, έγιναν δύο περισσότεροι έλεγχοι της έκφρασης από ότι είναι απαραίτητο, καθώς όταν αναζητήθηκαν εναλλακτικές λύσεις, δοκιμάστηκαν τα εναλλακτικά μονοπάτια των δύο επόμενων κανόνων πριν ο μηχανισμός εκτέλεσης καταλήξει στην απάντηση false. Όμοια συμπεριφορά παρατηρείται και στον δεύτερο κανόνα.

Εφόσον οι τρεις κανόνες είναι αμοιβαία αποκλειόμενοι, μπορούμε να αποφύγουμε την παραπάνω συμπεριφορά εισάγοντας το κατηγόρημα της αποκοπής στο τέλος κάθε κανόνα:

```
expression_sign(X,positive):- X > 0,!.
expression_sign(X,zero):- X =:=0,!.
expression_sign(X,negative):-X < 0,!.
```

και οι απαντήσεις του κατηγορήματος για τις ίδιες ερωτήσεις είναι:

```
?- expression_sign(2+2,S).
S = positive.
?- expression_sign(2-2,S).
S = zero.
?- expression_sign(2-5,S).
```

```
S = negative.
```

Όπως φαίνεται παραπάνω, αποκλείονται τα εναλλακτικά μονοπάτια, από την στιγμή που μια από τις συνθήκες επιτύχει, άρα μειώνεται ο αριθμός των ελέγχων κατά την οπισθοδρόμηση.

Μια ενδιαφέρουσα παρατήρηση, η οποία οδηγεί στην απαλοιφή ακόμη περισσότερων ελέγχων, είναι ότι καθώς η Prolog εξετάζει τους κανόνες με την σειρά που εμφανίζονται στον κώδικα, ο έλεγχος της τελευταίας πρότασης πλεονάζει, καθώς έχουν προηγηθεί οι δύο προηγούμενοι έλεγχοι. Η παρατήρηση οδηγεί στην απαλοιφή του τελευταίου έλεγχου, δίνοντας την τελική έκδοση του κατηγορήματος:

```
expression_sign(X,positive):- X > 0,!.
expression_sign(X,zero):- X =:=0,!.
expression_sign(_,negative).
```

Κατά συνέπεια, η αποκοπή οδηγεί στην δυνατότητα να παραλείψουμε ελέγχους, άρα οδηγεί σε αποδοτικότερες υλοποιήσεις των κατηγορημάτων. Θα μπορούσε να ισχυριστεί κάποιος ότι θα μπορούσε να υλοποιήσει μια παρόμοια έκδοση χωρίς την χρήση της αποκοπής, η οποία εκμεταλλεύεται την σειριακότητα της Prolog:

```
expression_sign_incorrect(X,positive):- X > 0.
expression_sign_incorrect(X,zero):- X =:=0.
expression_sign_incorrect(_,negative).
```

Το πρόβλημα δημιουργείται όταν ο μηχανισμός οπισθοδρόμησης παράγει τις εναλλακτικές λύσεις. Για παράδειγμα:

```
?- expression_sign_incorrect(2,S).
S = positive ;
S = negative.
```

Η δεύτερη (λανθασμένη) λύση, παράγεται από τον τρίτο κανόνα, καθώς η έλλειψη οποιουδήποτε ελέγχου επιτρέπει την επιτυχία για οποιαδήποτε έκφραση.

Είδη Αποκοπής

Η αποκοπή είναι δυνατό να αλλάζει τη δηλωτική ερμηνεία του προγράμματος, το οποίο σε πρακτικούς όρους σημαίνει ότι με την χρήση της αποκοπής μπορεί να αλλάξει το σύνολο των λύσεων - συμπερασμάτων, οι οποίες προκύπτουν από ένα πρόγραμμα. Έτσι διακρίνονται δύο είδη αποκοπής:

- οι **πράσινες αποκοπές** (green cuts), οι οποίες δεν μεταβάλλουν το σύνολο των λύσεων, δηλαδή δεν μεταβάλλουν τη δηλωτική σημασία του προγράμματος, αλλά βελτιώνουν την απόδοσή του,
- οι **κόκκινες αποκοπές** (red cuts), οι οποίες μεταβάλλουν τη δηλωτική ερμηνεία του προγράμματος, με άλλα λόγια η εισαγωγή τους στο πρόγραμμα αλλάζει το σύνολο λύσεων.

Το παράδειγμα του κατηγορήματος expression_sign/2 που εξετάστηκε στην προηγούμενη παράγραφο είναι χαρακτηριστικό. Στις δύο πρώτες εκδόσεις του κατηγορήματος:

(α' έκδοση):

```
expression_sign(X,positive):- X > 0.
expression_sign(X,zero):- X =:=0.
expression_sign(X,negative):-X < 0.
```

(β' έκδοση) - Πράσινες αποκοπές:

```
expression_sign(X,positive):- X > 0,!.
expression_sign(X,zero):- X =:=0,!.
expression_sign(X,negative):-X < 0,!.
```

η αποκοπή δεν επηρεάζει το σύνολο των λύσεων, αλλά μόνο την απόδοση του προγράμματος, άρα ανήκουν την περίπτωση των πράσινων αποκοπών. Αντίθετα στις δύο επόμενες εκδόσεις:

(γ' έκδοση) - Κόκκινες αποκοπές:

```
expression_sign(X,positive):- X > 0,!.
expression_sign(X,zero):- X =:=0,!.
expression_sign(_,negative).
```

(δ' έκδοση) - Έκδοση γ' χωρίς αποκοπές:

```
expression_sign_incorrect(X,positive):- X > 0.
expression_sign_incorrect(X,zero):- X =:=0.
expression_sign_incorrect(_,negative).
```

η εισαγωγή της αποκοπής διαφοροποιεί τις λύσεις που δημιουργούνται από τις εκδόσεις γ' και δ' του κατηγόρηματος, άρα οι δύο εμφανίσεις της αποκοπής ανήκουν στην κατηγορία των κόκκινων αποκοπών.

Μια εναλλακτική διατύπωση της διάκρισης μεταξύ των πράσινων και κόκκινων αποκοπών είναι ότι η αφαίρεση των πρώτων (green cuts) δεν αλλάζει το σύνολο των λύσεων, ενώ η αφαίρεση των κόκκινων αποκοπών μεταβάλλει τις παραγόμενες λύσεις.

Αναδρομή και Αποκοπή: Τομή δύο Λιστών

Η τομή δύο λιστών είναι μια λίστα η οποία περιέχει τα στοιχεία που ανήκουν και στις δύο λίστες. Για χάρη απλότητας, θα θεωρήσουμε ότι οι λίστες είναι σύνολα, δηλαδή κάθε στοιχείο τους εμφανίζεται μόνο μία φορά. Ας ορίσουμε το κατηγόρημα intersect/3, το οποίο δέχεται στα πρώτα δύο του ορίσματα δύο λίστες, και ενοποιεί το τρίτο όρισμα με τα κοινά στοιχεία των δύο λιστών. Για παράδειγμα, αν η πρώτη λίστα είναι η L1 = [a,b,c] και η λίστα του δεύτερου ορίσματος L2 = [b,c,d,e], τότε η τομή τους είναι η λίστα L3 = [b,c]. Καθώς το κατηγόρημα αφορά διαχείριση λιστών, η καταλληλότερη τεχνική είναι η αναδρομή. Η αναδρομική - δηλωτική περιγραφή του κατηγορήματος είναι η ακόλουθη:

- Αν η L1 είναι η κενή λίστα, τότε η τομή της με οποιαδήποτε λίστα είναι η κενή λίστα.
- Αν η κεφαλή μιας μη κενής λίστας L1 είναι μέλος της λίστας L2, τότε η τομή τους είναι μια λίστα που έχει ως κεφαλή την κεφαλή της λίστας L1 και ουρά την τομή της ουράς της λίστας L1 με την λίστα L2.
- Αν η κεφαλή μιας μη κενής λίστας L1 *δεν* είναι μέλος της λίστας L2, τότε η τομή τους είναι τομή της ουράς της λίστας L1 με την λίστα L2.

Ο παραπάνω δηλωτικός ορισμός υλοποιείται στην Prolog με το ακόλουθο κατηγόρημα:

```
intersect([],_,[]).
intersect([X|Rest],List,[X|IntList]):-
  member(X,List),
  intersect(Rest,List,IntList).
intersect([X|Rest],List,IntList):-
  not(member(X,List)),
  intersect(Rest,List,IntList).
```

Η συμπεριφορά του κατηγορήματος είναι η ακόλουθη:

```
?- intersect([a,b,c],[b,c,d,e],L).
L = [b, c] .

?- intersect([a],[b,c,d,e],L).
L = [].
```

Όπως και στο παράδειγμα του προσήμου των αριθμητικών εκφράσεων, είναι δυνατό να επιταχύνουμε την εκτέλεση του κατηγορήματος εισάγοντας την αποκοπή μετά τον έλεγχο member/2 του πρώτου κανόνα, καθώς οι δύο συνθήκες των κανόνων είναι αμοιβαία αποκλειόμενες:

```
intersect([],_,[]).
intersect([X|Rest],List,[X|IntList]):-
  member(X,List), !,
  intersect(Rest,List,IntList).
intersect([X|Rest],List,IntList):-
  not(member(X,List)),
  intersect(Rest,List,IntList).
```

Στην περίπτωση αυτή η αποκοπή είναι πράσινη καθώς η μόνη της επίδραση είναι η αύξηση της απόδοσης. Εφόσον ο έλεγχος μέλους γίνεται στην δεύτερη πρόταση, αν η εκτέλεση φτάσει στην τρίτη πρόταση έχει

εξασφαλιστεί ότι η κεφαλή δεν ανήκει στην λίστα του δεύτερου ορίσματος (List). Άρα ο έλεγχος not(member(X,List)) είναι περιττός και μπορεί να παραληφθεί:

```
intersect([],_,[]).
intersect([X|Rest],List,[X|IntList]):-
  member(X,List), !,
  intersect(Rest,List,IntList).
intersect([X|Rest],List,IntList):-
  intersect(Rest,List,IntList).
```

Αφαιρώντας τον περιττό έλεγχο η αποκοπή της δεύτερη πρότασης μετατράπηκε σε κόκκινη αποκοπή. Χωρίς αυτή, η παραπάνω υλοποίηση θα επιστρέψει λανθασμένες λύσεις κατά την οπισθοδρόμηση. Η απόδοση όμως της τελικής αυτής έκδοσης έχει αυξηθεί σημαντικά, καθώς ο έλεγχος μέσω του κατηγορήματος member/2 μπορεί να απαιτήσει σημαντικό χρόνο σε λίστες με μεγάλο μέγεθος.

Κατηγορήματα Αποτυχίας και Επιτυχίας

Όπως φάνηκε από τις προηγούμενες παραγράφους, ο μηχανισμός οπισθοδρόμησης αποτελεί ένα ισχυρό εργαλείο για την υλοποίηση προγραμμάτων. Εκτός από το κατηγόρημα της αποκοπής, ο έλεγχος του μηχανισμού οπισθοδρόμησης στην Prolog περιλαμβάνει τα ακόλουθα κατηγορήματα:

- fail, το οποίο αποτυγχάνει πάντα,
- true, το οποίο πετυχαίνει μόνο μια φορά,
- repeat, το οποίο πετυχαίνει όσες φορές και αν κληθεί, δηλαδή δημιουργεί άπειρες επιτυχείς εναλλακτικές λύσεις στο σημείο που εμφανίζεται.

Η συμπεριφορά των κατηγορημάτων φαίνεται στις ακόλουθες ερωτήσεις:

```
?- fail.
false.

?- true.
true.

?- repeat.
true ;
true ;
true ;
true ;
true ;
true
...
```

Προσοχή πρέπει να δοθεί στην διαφορά του κατηγορήματος true και repeat. Όσες εναλλακτικές λύσεις και αν ζητήσουμε στην τρίτη ερώτηση, η Prolog θα απαντήσει πάντα θετικά.

Εξαναγκασμός σε Αποτυχία

Αν και η λειτουργία των παραπάνω κατηγορημάτων δεν φαίνεται να έχει μεγάλο νόημα, η ύπαρξη τους επιτρέπει την υλοποίηση πλήθους τεχνικών λογικού προγραμματισμού. Από τις πλέον σημαντικές τεχνικές είναι ο εξαναγκασμός σε αποτυχία με την χρήση του συνδυασμού αποκοπής και αποτυχίας (cut-fail). Κλασικό παράδειγμα αποτελεί η υλοποίηση της κατηγορήματος μη-μέλος μιας λίστας (not_member/2), το οποίο αποτυγχάνει αν το πρώτο του όρισμα είναι μέλος της λίστας του δεύτερου ορίσματος. Η απλή προσέγγιση είναι να χρησιμοποιηθεί άρνηση:

```
not_member(X,List):-not(member(X,List)).
```

που λειτουργεί θαυμάσια. Το ζητούμενο είναι να υλοποιηθεί το κατηγόρημα, χωρίς την χρήση του not/1. Μια περιγραφή του κατηγορήματος είναι:

1. αν το πρώτο όρισμα X είναι μέλος της λίστας List, τότε το κατηγόρημα αποτυγχάνει,

2. αλλιώς, το κατηγόρημα επιτυγχάνει.

Η πρώτη (λανθασμένη) προσέγγιση στην υλοποίηση της περιγραφής είναι για τον ορισμό της αποτυχίας να χρησιμοποιηθεί το κατηγόρημα fail:

```
not_member(X,List):-member(X,List),fail.
not_member(_X,_List).
```

όπου κάθε πρόταση του κατηγορήματος αντιστοιχεί σε μια περίπτωση της περιγραφής. Οι ακόλουθες ερωτήσεις δείχνουν την συμπεριφορά του κατηγορήματος:

```
?- not_member(4,[1,2,3]).
true.

?- not_member(3,[1,2,3]).
true.

?- not_member(X,[1,2,3]).
true.
```

Η απάντηση στην πρώτη ερώτηση είναι ορθή, όμως στις επόμενες δύο ερωτήσεις είναι λανθασμένες. Μια προσεκτικότερη ανάλυση δείχνει ότι το κατηγόρημα πετυχαίνει πάντα. Αν ο υποστόχος member(X,List) της πρώτης πρότασης αποτύχει, δηλαδή το πρώτο όρισμα X δεν είναι μέλος της λίστας, τότε ο μηχανισμός οπισθοδρόμησης επιλέγει την δεύτερη πρόταση και καθώς αυτή δεν θέτει περιορισμούς στα ορίσματα, η ερώτηση επιτυγχάνει. Αυτή είναι η επιδιωκόμενη συμπεριφορά και με αυτόν τον τρόπο η πρώτη ερώτηση στο παράδειγμα επιτυγχάνει.

Αν ο υποστόχος member(X, List) πετύχει, δηλαδή το X είναι μέλος της λίστας, τότε η Prolog εκτελεί το κατηγόρημα fail, το οποίο αποτυγχάνει πάντα, και άρα ο μηχανισμός οπισθοδρόμησης ακολουθεί την ίδια διαδικασία με την προηγούμενη περίπτωση, η ερώτηση ενοποιείται με την δεύτερη εναλλακτική πρόταση και επιτυγχάνει. Με τον διαδικασία αυτή παράγονται οι λανθασμένες λύσεις της δεύτερης και τρίτης ερώτησης.

Είναι προφανές ότι θα πρέπει μετά την επιτυχία του υποστόχου member/2 να αποκόπτεται η εναλλακτική λύση της δεύτερης πρότασης, πριν την εκτέλεση του fail. Αυτό επιτυγχάνεται με την εισαγωγή του κατηγορήματος της αποκοπής:

```
not_member(X,List):-member(X,List),!,fail.
not_member(_X,_List).
```

Το κατηγόρημα επιστρέφει σωστές απαντήσεις και στις τρεις παραπάνω ερωτήσεις:

```
?- not_member(4,[1,2,3]).
true.

?- not_member(3,[1,2,3]).
false.

?- not_member(X,[1,2,3]).
false.
```

Ο συνδυασμός αποκοπής και αποτυχίας (cut-fail) χρησιμοποιείται οποτεδήποτε θέλουμε να εξαναγκάσουμε την εκτέλεση σε αποτυχία. Ιδιαίτερο ενδιαφέρον παρουσιάζει η τελευταία ερώτηση η οποία αποτυγχάνει καθώς το ερώτημα που διατυπώνει είναι "δεν υπάρχει X, το οποίο να είναι μέλος της λίστας" και όχι το ερώτημα "υπάρχει X, τέτοιο ώστε να μην είναι μέλος της λίστας". Αυτό φαίνεται και από την ακόλουθη ερώτηση:

```
?- not_member(X,[]).
true.
```

η οποία ορθώς επιτυγχάνει καθώς δεν υπάρχει τιμή η οποία να ανήκει στην κενή λίστα.

Στο παράδειγμα του κοινωνικού δικτύου, η σχέση does_not_follow(User,Followee) του εισαγωγικού παραδείγματος, μπορεί να γραφεί:

```
does_not_follow(User,Followee):-
  follows(User,Followee),
```

```
  !,fail.

does_not_follow(_User,_Followee).
```

Και στην περίπτωση αυτή, το κατηγόρημα λειτουργεί σωστά όταν καλείται και με τα δύο ορίσματα δεσμευμένα σε κάποια τιμή. Για παράδειγμα:

```
?- does_not_follow(ilias,demos).
true.
?- does_not_follow(ilias,petros).
false.
?- does_not_follow(ilias,X).
false.
```

Από την τελευταία ερώτηση προκύπτει ότι το κατηγόρημα δεν μπορεί να χρησιμοποιηθεί για να παράξουμε τους χρήστες που δεν ακολουθούν τον ilias. Αυτό θα μπορούσε να γίνει αν το ερώτημα που διατυπώνονταν ήταν, "υπάρχει χρήστης X, τον οποίο δεν ακολουθεί ο χρήστης ilias" το οποίο εκφράζεται από την σύνθετη ερώτηση:

```
?- user(X,_),does_not_follow(ilias,X).
X = ilias ;
X = demos ;
X = nick ;
X = sofia ;
X = helen ;
false.
```

Η πρώτη απάντηση αντιστοιχεί στο απόλυτα σωστό συμπέρασμα από την άποψη της λογικής πρώτης τάξης, ότι ένας χρήστης δεν ακολουθεί τον εαυτό του.

Όπως θα εξεταστεί στο Κεφάλαιο 10, η παραπάνω τεχνική σε συνδυασμό με την μεταβλητή κλήση χρησιμοποιείται στην υλοποίηση του κατηγορήματος της άρνησης.

Επανάληψη μέσω οπισθοδρόμησης

Η Prolog στερείται των κλασσικών δομών ελέγχου επανάληψης που υπάρχουν στις γλώσσες προστατικού προγραμματισμού, όπως για παράδειγμα των εντολών while και for. Η βασική τεχνική εκτέλεσης του ίδιου συνόλου ενεργειών σε πολλά δεδομένα (π.χ. στα στοιχεία μιας λίστας) που έχουμε εξετάσει μέχρι τώρα είναι η αναδρομή. Για παράδειγμα, για να τυπώσουμε στην οθόνη τα στοιχεία μιας λίστας (χρησιμοποιώντας το κατηγόρημα write/1 και nl/0 (newline), βλέπε ενότητα 9.3), τότε ο απλούστερος τρόπος θα ήταν να χρησιμοποιήσουμε τον ακόλουθο αναδρομικό ορισμό:

```
write_list([]).
write_list([X|Tail]) :-
  write(X), nl,
  write_list(Tail).
```

Το κατηγόρημα διατρέχει τα στοιχεία της λίστας χρησιμοποιώντας την κλασική σημειογραφία διαχωρισμού κεφαλής - ουράς, τυπώνοντας κάθε φορά την κεφαλή και καλώντας αναδρομικά τον εαυτό του με όρισμα την ουρά:

```
?- write_list([a,b,c,d]).
a
b
c
d
true.
```

Όμως, το κατηγόρημα member/2 είναι δυνατό να χρησιμοποιηθεί για να δεσμεύσουμε διαδοχικά το πρώτο του όρισμα στα στοιχεία μιας λίστας κατά την οπισθοδρόμηση. Άρα, θα μπορούσε να χρησιμοποιηθεί για να τυπώσουμε τα στοιχεία της λίστας:

```
write_list_f(List):-
  member(X,List),
  write(X), nl.
```

Το πρόβλημα στο παραπάνω κατηγόρημα είναι ότι αφού τυπώσει ένα στοιχείο της λίστας, πετυχαίνει και απαιτείται να ζητήσουμε ρητά (πατώντας το πλήκτρο ";") να εκτελεστεί και για τα υπόλοιπα στοιχεία της λίστας, δηλαδή:

```
?- write_list_f([a,b,c,d]).
a
true ;
b
true ;
c
true ;
d
true.
```

Για να επιτύχουμε την εκτύπωση όλων των στοιχείων σε μια κλήση του κατηγορήματος, θα πρέπει να "προκαλέσουμε" την οπισθοδρόμηση εισάγοντας ως τελευταίο στόχο το κατηγόρημα fail:

```
write_list_f(List):-
  member(X,List),
  write(X), nl,
  fail.
```

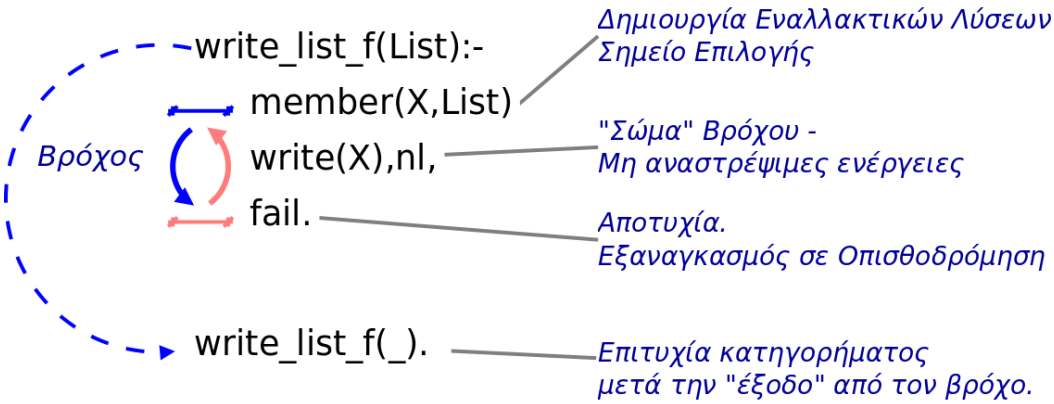
Ο μηχανισμός εκτέλεσης στην παραπάνω πρόταση ενοποιεί την μεταβλητή X με το πρώτο στοιχείο της λίστας List, έπειτα εμφανίζει το στοιχείο και όταν συναντήσει το κατηγόρημα fail, οπισθοδρομεί στο τελευταίο σημείο επιλογής, δηλαδή στον υποστόχο member/2, ο οποίος ενοποιεί την μεταβλητή X στο επόμενο στοιχείο της λίστας, το εμφανίζει στη οθόνη, αποτυγχάνει κ.ο.κ. Η διαδικασία παλινδρομεί μεταξύ του υποστόχου fail και του member/2 όσο το τελευταίο έχει εναλλακτικές λύσεις:

```
?- write_list([a,b,c,d]).
a
b
c
d
false.
```

Όπως φαίνεται παραπάνω, όταν τυπωθεί και το τελευταίο στοιχείο το κατηγόρημα αποτυγχάνει. Για να επιτύχουμε πλήρως την συμπεριφορά της αναδρομικής λύσης που παρουσιάστηκε παραπάνω, δηλαδή το κατηγόρημα να επιτυγχάνει μετά την εμφάνιση των στοιχείων της λίστας θα πρέπει να προστεθεί μια ακόμη πρόταση στο κατηγόρημα, η οποία πετυχαίνει πάντα, όπως και στην υλοποίηση των κατηγορημάτων not_member/2 και does_not_follow/2 που παρουσιάστηκαν παραπάνω:

```
write_list_f(List):-
  member(X,List),
  write(X), nl,
  fail.
write_list_f(_).
```

Η παραπάνω υλοποίηση είναι αντίστοιχη ενός βρόχου, όπου η συνθήκη διατυπώνεται στην αρχή του βρόχου, ο οποίος περικλείεται ανάμεσα σε ένα υποστόχο με πολλαπλές λύσεις και στο κατηγόρημα fail. Ο βρόχος εκτελείται όσο υπάρχουν εναλλακτικές λύσεις στο κατηγόρημα-συνθήκη και για να έχει νόημα θα πρέπει να περιλαμβάνει στο σώμα του, υποστόχους με κατηγορήματα εισόδου/εξόδου (read/write) ή κατηγορήματα προσθαφαίρεσης προτάσεων (assert/retract - Κεφάλαιο 10), δηλαδή κατηγορήματα με μη-αναστρέψιμες κατά την οπισθοδρόμηση ενέργειες (Σχήμα 9.4). Τέτοιοι βρόχοι ονομάζονται βρόχοι οδηγούμενοι από αποτυχία (failure-driven loops).



Σχήμα 9.4: *Βρόγχος οδηγούμενος από αποτυχία*

Ιδιαίτερη προσοχή θα πρέπει να δοθεί όταν στο σώμα ενός τέτοιου βρόχου περιέχονται περισσότερα του ενός κατηγορήματα τα οποία έχουν πολλές λύσεις. Στην περίπτωση αυτή ο μηχανισμός οπισθοδρόμησης θα εξερευνήσει συστηματικά όλους τους δυνατούς συνδυασμούς λύσεων που προκύπτουν. Το κατηγόρημα print_cartesian_product/2, δέχεται δύο λίστες σαν ορίσματα και τυπώνει το καρτεσιανό γινόμενο των στοιχείων τους:

```
print_cartesian_product(L1,L2):-
  member(X,L1),
  member(Y,L2),
  write(X),write("-"),write(Y), nl,
  fail.

print_cartesian_product(_,_).
```

Η βασική τεχνική είναι παρόμοια με εκείνη που ακολουθήθηκε παραπάνω, όμως καθώς υπάρχουν δύο σημεία επιλογής (choice points) πριν από το fail, ο μηχανισμός εκτέλεσης εξετάζει συστηματικά και συνδυάζει τις εναλλακτικές λύσεις, όπως φαίνεται από την ερώτηση που ακολουθεί:

```
?- print_cartesian_product([a,b],[1,2,3]).
a-1
a-2
a-3
b-1
b-2
b-3
true.
```

Συχνά είναι απαραίτητο να εκτελείται ένα βρόχος μέχρι να αληθεύει μια συνθήκη, της οποίας όμως η τιμή εξαρτάται από “εξωγενείς" προς το κατηγόρημα παράγοντες, για παράδειγμα, μια τιμή που εισάγει ο χρήστης ή την τρέχουσα ώρα. Στο παράδειγμα που ακολουθεί, θα θεωρήσουμε το κατηγόρημα wait/1, το οποίο πετυχαίνει αφού έχουν περάσει N δευτερόλεπτα από την στιγμή που κλήθηκε. Στην SWI-Prolog, η τρέχουσα ώρα δίνεται από το κατηγόρημα get_time/1, το οποίο ενοποιεί το όρισμά του με τον αριθμό των δευτερολέπτων που έχουν περάσει από την 1/1/1970:

```
?- get_time(Time).
Time = 1441201554.51853.
```

Το κατηγόρημα πετυχαίνει μόνο μια φορά. Η πρώτη προσέγγιση στη λύση είναι να διαβαστεί η ώρα δύο φορές, και να απαιτηθεί οι δύο χρονικές στιγμές να απέχουν τουλάχιστον κατά N δευτερόλεπτα:

```
wait(N):-
  get_time(Start),
  get_time(End),
  End > Start + N.
```

Η παραπάνω λύση όμως αποτυγχάνει καθώς το κατηγόρημα get_time/1 πετυχαίνει μόνο μια φορά σε κάθε κλήση του:

```
?- wait(2).
false.
```

Η ορθή προσέγγιση θα ήταν αφού ζητηθεί ο χρόνος έναρξης του κατηγορήματος (πρώτος υποστόχος get_time/1) να επαναλαμβάνεται ο υποστόχος get_time(End), μέχρις ότου η χρονική στιγμή End ικανοποιεί την συνθήκη, δηλαδή η υλοποίηση ενός repeat-until βρόχου. Αυτό επιτυγχάνεται εισάγοντας ανάμεσα στους δύο υποστόχους get_time/1 το κατηγόρημα repeat:

```
wait(N):-
  get_time(Start),
  repeat,
  get_time(End),
  End > Start + N.
```

Καθώς το repeat πετυχαίνει όσες φορές και αν κληθεί κατά την οπισθοδρόμηση, αποτελεί σημείο επιλογής με άπειρες λύσεις, στο οποίο πάντα "σταματά" η οπισθοδρόμηση, με αποτέλεσμα ο μηχανισμός εκτέλεσης να καλεί εκ νέου τον υποστόχο get_time(End), δεσμεύοντας την μεταβλητή End στην νέα τρέχουσα τιμή του χρόνου, μέχρι ότου η συνθήκη πετύχει:

```
?- wait(2).
...(μετά από δύο δευτερόλεπτα)
true ;
```

Μια τελευταία παρατήρηση είναι ότι μετά την πάροδο των N δευτερολέπτων από την αρχική κλήση του κατηγορήματος, αυτό πετυχαίνει όσες φορές και αν κληθεί, λόγω της ύπαρξης του repeat:

```
?- wait(2).
...(μετά από δύο δευτερόλεπτα)
true ;
true ;
true ;
```

Για να αποφύγουμε μια τέτοια συμπεριφορά, δηλαδή το κατηγόρημα να πετυχαίνει μόνο μια φορά, εισάγουμε το κατηγόρημα της αποκοπής μετά την συνθήκη, το οποίο κλαδεύει τις εναλλακτικές λύσεις του repeat:

```
wait(N):-
  get_time(Start),
  repeat,
  get_time(End),
  End > Start + N,
  !.
```

Η τελική συμπεριφορά του κατηγορήματος είναι:

```
?- wait(2).
true.
```

Στην κατηγόρημα που εξετάστηκε, η συνθήκη του βρόχου βρίσκεται στο τέλος του, άρα αποτελεί αντίστοιχο των δομών επανάληψης repeat-until που υπάρχουν στις κλασικές διαδικαστικές γλώσσες προγραμματισμού. Η σημαντική διαφορά είναι ότι λόγω της μοναδικής ανάθεσης των μεταβλητών, συνθήκη θα πρέπει να εξαρτάται από "εξωτερικούς" ως προς το κατηγόρημα παράγοντες, όπως η τρέχουσα ώρα στην περίπτωση που εξετάστηκε.

Τέλος, προσοχή θα πρέπει να δοθεί στον συνδυασμό των κατηγορημάτων repeat-fail, η οποία οδηγεί σε ατέρμονα βρόχο:

```
write_for_ever(Text):-
  repeat,
  write(Text),nl,
  fail.
```

Η ακόλουθη ερώτηση τυπώνει το μήνυμα του ορίσματος άπειρες φορές στην οθόνη:

```
?- write_for_ever('Prolog Rules!').
Prolog Rules!
Prolog Rules!
Prolog Rules!
```

```
...
```

Αν και ο συνδυασμός repeat-fail φαίνεται να μην έχει ιδιαίτερη χρησιμότητα, εντούτοις σε κάποιες εφαρμογές είναι ο μοναδική προσέγγιση. Για παράδειγμα η υλοποίηση ενός Εξυπηρετητή Ιστού (Web server), ο οποίος θεωρητικά θα πρέπει να εκτελείται επ' άπειρον, αποτελεί μια τέτοια περίπτωση, καθώς η χρήση της αναδρομής θα εξαντλούσε σε σύντομο χρονικό διάστημα τη διαθέσιμη μνήμη.

Το σημαντικό πλεονέκτημα που προσφέρει η χρήση βρόχων που βασίζονται στην οπισθοδρόμηση, είναι ότι επιτρέπουν την υλοποίηση αποδοτικότερων προγραμμάτων:

- Καθώς η οπισθοδρόμηση ακυρώνει τις αναθέσεις τιμών προηγούμενων μεταβλητών και δεν δεσμεύει νέο χώρο στην μνήμη για το περιβάλλον κλήσης κάθε κατηγορήματος. Έτσι, οι απαιτήσεις σε μνήμη κατηγορημάτων που περιέχουν ιδιαίτερα πολλές επαναλήψεις είναι πολύ μικρές.
- Η οπισθοδρόμηση είναι σημαντικά γρηγορότερη της αναδρομής, άρα στην περίπτωση πολλών επαναλήψεων, τα κατηγορήματα που ακολουθούν την τεχνική αυτή είναι σημαντικά ταχύτερα σε εκτέλεση.

9.3 Ρεύματα Εισόδου Εξόδου στην Prolog

Η συνηθισμένη αλληλεπίδραση μεταξύ του χρήστη και της γλώσσας Prolog, γίνεται μέσω ερωτήσεων που πληκτρολογεί ο χρήστης και απαντήσεων που εμφανίζονται από το σύστημα στην οθόνη. Προφανώς, μια πλήρης γλώσσα προγραμματισμού γενικού σκοπού απαιτεί την δυνατότητα επικοινωνίας με αρχεία, συνδέσεις TCP/IP κοκ. Όπως και στις συμβατικές γλώσσες προγραμματισμού, η επικοινωνία αυτή βασίζεται στην έννοια των ρευμάτων εισόδου/εξόδου. Η απλούστερη μορφή χειρισμού των ρευμάτων εισόδου/εξόδου είναι μέσω των ενσωματωμένων κατηγορημάτων read/1, write/1 και nl.

Το κατηγόρημα read/1, διαβάζει όρους τους οποίους ενοποιεί με το όρισμά του. Ιδιαίτερη προσοχή πρέπει να δοθεί στο γεγονός ότι το κατηγόρημα αναμένει από τον χρήστη να πληκτρολογήσει ένα οποιονδήποτε έγκυρο Prolog όρο ο οποίος να τελειώνει με την τελεία, αλλιώς εμφανίζει μήνυμα λάθους. Για παράδειγμα:

```
?- read(X).
|:   prolog.
X = prolog.

?- read(X).
|:  [1,2,3].
X = [1, 2, 3].

?- read(X).
|:   hello world.
ERROR: ...
```

Το σύμβολο "|:" είναι το προτρεπτικό σήμα (prompt) της Prolog. Στην τρίτη ερώτηση προέκυψε σφάλμα, καθώς ο χρήστης πληκτρολόγησε ένα μη έγκυρο Prolog όρο. Η απαίτηση για την εισαγωγή έγκυρων όρων προκύπτει από το γεγονός πως οτιδήποτε πληκτρολογήσει ο χρήστης θα πρέπει να ενοποιηθεί με μια Prolog μεταβλητή.

Όπως εξετάστηκε στην προηγούμενη ενότητα, το κατηγόρημα write/1 εμφανίζει τον όρο του ορίσματός του στην οθόνη, ενώ το nl (newline) εισάγει τον χαρακτήρα αλλαγής γραμμής:

```
?- write(hello),write(world).
helloworld
true.

?- write(hello),nl,write(world).
hello
world
true.
```

Και εδώ το όρισμα μπορεί να είναι οποιοσδήποτε έγκυρος Prolog όρος:

```
?- write(name(john)).
name(john)
true.

?- write(X).
_G16187
true.

?- write('hello world').
hello world
true.

?- write(hello world).
ERROR: ...
```

Περισσότερο προσοχή απαιτεί η διαφορά των δύο τελευταίων ερωτήσεων. Στην προτελευταία ερώτηση, η φράση hello world περικλείεται σε μονά εισαγωγικά, άρα είναι όρος (δες Κεφάλαιο 4), ενώ στην τελευταία το όρισμα που δόθηκε δεν αποτελεί όρο.

Το κατηγόρημα user_input_yn/1 το οποίο υλοποιεί την κλασσική ερώτηση χρήστη y/n, μπορεί να υλοποιηθεί εύκολα χρησιμοποιώντας βρόχους οδηγούμενους από αποτυχία:

```
user_input_yn(Ans):-
  repeat,
  write('Please answer y/n '),nl,
  read(Ans),
  (Ans = y ; Ans = n),
  !.
```

Το παραπάνω κατηγόρημα είναι μία από τις λίγες περιπτώσεις χρήσης του τελεστή διάζευξης (“;”). Η εκτέλεση του κατηγορήματος φαίνεται στις ακόλουθες ερωτήσεις:

```
?- user_input_yn(Ans).
Please answer y/n
|: other.
Please answer y/n
|: y.

Ans = y.

?- user_input_yn(Ans).
Please answer y/n
|: foo.
Please answer y/n
|: n.

Ans = n.
```

Μέχρι στιγμής τα κατηγορήματα read και write αφορούσαν τα εξορισμού ρεύματα εξόδου και εισόδου της Prolog που είναι η οθόνη και το πληκτρολόγιο. Τα δύο αυτά ρεύματα έχουν κοινό όνομα user. Το τρέχον ρεύμα εξόδου δίνεται από το ενσωματωμένο κατηγόρημα telling/1, ενώ το τρέχον ρεύμα εισόδου από το κατηγόρημα seeing/1. Έτσι, αν δεν έχει γίνει ανακατεύθυνση των ρευμάτων, ισχύει:

```
?- seeing(InStream).
InStream = user.

?- telling(OutStream).
OutStream = user.
```

Ανακατεύθυνση των ρευμάτων γίνεται μέσω των κατηγορημάτων see/1 και tell/1, τα οποία δέχονται σαν όρισμα το όνομα ενός αρχείου εισόδου και εξόδου αντίστοιχα. Το τρέχον ρεύμα, επανέρχεται στις εξορισμού ρυθμίσεις του (user), με τα κατηγορήματα seen/0 και told/0. Το κατηγόρημα print_to_file/2, δημιουργεί ένα

αρχείο με όνομα που εμφανίζεται στο πρώτο του όρισμα, στο οποίο γράφει το περιεχόμενο του δευτέρου ορίσματος:

```
print_to_file(FileName,Content):-
  tell(FileName),
  write(Content),nl,
  told.
```

Για παράδειγμα η ακόλουθη ερώτηση:

```
?- print_to_file('testFile.txt','Hello World').
true.
```

δημιουργεί ένα αρχείο με το όνομα 'testFile.txt' στο οποίο εισάγει την φράση 'Hello World'. Ιδιαίτερη προσοχή θα πρέπει να δοθεί στο γεγονός ότι τα δύο ορίσματα περικλείονται σε απλά εισαγωγικά (όροι). Αν απαιτείται να προσθέσουμε κάτι στο τέλος ενός αρχείου το οποίο ήδη υπάρχει, τότε η ανακατεύθυνση του ρεύματος εξόδου γίνεται με το κατηγόρημα append/1.

Αντίστοιχα η ανάγνωση του αρχείου μπορεί να γίνει με το κατηγόρημα read/1. Όμως, καθώς το τελευταίο "διαβάζει" ένα Prolog όρο, θα πρέπει το αρχείο να απαρτίζεται από έγκυρους Prolog όρους, οι οποίοι τερματίζονται με την τελεία. Έτσι, έστω ένα αρχείο testReadFile με περιεχόμενα:

```
name(john).
nick.
12.
[2,3,4].
```

Η ερώτηση ενοποιεί τη μεταβλητή X στον πρώτο όρο που περιέχεται στο αρχείο:

```
?- see(testReadFile), read(X).
X = name(john).
```

Διαδοχικές κλήσεις του κατηγορήματος read/1, θα ενοποιήσουν την μεταβλητή του ορίσματός του με τους υπόλοιπους όρους του αρχείου:

```
?- read(X).
X = nick.
?- read(X).
X = 12.
?- read(X).
X = [2, 3, 4].
?- read(X).
X = end_of_file.
```

Κατά τα συνηθισμένα και σε άλλες γλώσσες προγραμματισμού, όταν έχουν αναγνωσθεί όλα τα περιεχόμενα του αρχείου, τότε το κατηγόρημα read/1, δεσμεύει την μεταβλητή με τον ειδικό όρο end_of_file. Η παραπάνω συμπεριφορά παραπέμπει στη χρήση ενός βρόχου repeat, με συνθήκη τερματισμού την ανάγνωση του τέλους του αρχείου. Το κατηγόρημα που ακολουθεί εμφανίζει στην οθόνη τους όρους ενός αρχείου:

```
print_terms_of_file(FileName):-
  see(FileName),
  repeat,
  read(Term),
  write(Term),nl,
  Term = end_of_file,
  !,
  seen.
```

Η συμπεριφορά του για το αρχείο testReadFile είναι:

```
?- print_terms_of_file(testReadFile).
name(john)
nick
12
[2,3,4]
end_of_file
true.
```

Το κατηγόρημα λειτουργεί σωστά, όμως εμφανίζει και το end_of_file, κάτι το οποίο δεν είναι επιθυμητό. Υπάρχουν διάφορες προσεγγίσεις στην επίλυση του παραπάνω προβλήματος. Η απλούστερη είναι οι όροι να εμφανίζονται με την βοήθεια ενός βοηθητικού κατηγορήματος handle_term/1, το οποίο αν το όρισμά του είναι ο ειδικός όρος end_of_file, τότε απλά πετυχαίνει, χωρίς καμιά άλλη "ενέργεια", ενώ σε οποιαδήποτε άλλη περίπτωση εμφανίζει το όρισμά του και αποτυγχάνει, προκαλώντας οπισθοδρόμηση:

```
handle_file_term(end_of_file):- !.

handle_file_term(Term):-
  write(Term),nl,
  fail.
```

Στο παραπάνω κατηγόρημα η αποκοπή στην πρώτη πρόταση εξασφαλίζει ότι σε καμιά περίπτωση δεν θα τυπωθεί ο όρος end_of_file. Με την χρήση του νέου αυτού κατηγορήματος, το print_terms_of_file/1 γίνεται:

```
print_terms_of_file(FileName):-
  see(FileName),
  repeat,
  read(Term),
  handle_file_term(Term),
  !,
  seen.
```

Η αποκοπή παραπάνω αποκλείει τα (άπειρα) εναλλακτικά μονοπάτια που εισάγονται από το repeat. Η συμπεριφορά του κατηγορήματος είναι:

```
?- print_terms_of_file(testReadFile).
name(john)
nick
12
[2,3,4]
true.
```

Ανάγνωση Μεμονωμένων Χαρακτήρων και Μετατροπή σε άτομα

Τα κατηγορήματα της προηγούμενης παραγράφου είναι ιδιαίτερα για τη γλώσσα, με την έννοια ότι διαβάζουν Prolog όρους από το ρεύμα εισόδου. Μια τέτοια προσέγγιση, αν και ταιριάζει στην φιλοσοφία της γλώσσας είναι αρκετά περιοριστική όσον αφορά τη δομή των αρχείων. Κατά συνέπεια, η Prolog προσφέρει κατηγορήματα ανάγνωσης/εγγραφής μεμονωμένων χαρακτήρων σε ρεύματα. Τα βασικά κατηγορήματα είναι:

- put_code/1, το οποίο στέλνει το όρισμα του στο τρέχον ρεύμα εξόδου. Το όρισμα είναι ένας κωδικός χαρακτήρα (πχ ASCII).
- put_char/1, το οποίο είναι παρόμοιο με το put_code/1, με την διαφορά ότι το όρισμα είναι ένας χαρακτήρας.
- get_code/1, το οποίο ενοποιεί το όρισμά του με τον κωδικό του επόμενου χαρακτήρα του ρεύματος εισόδου. Το κατηγόρημα ενοποιεί το όρισμά του με το -1 όταν η ανάγνωση φτάσει στο τέλος του αρχείου.
- get_char/1, με παρόμοια συμπεριφορά με το get_code/1, αλλά ενοποιεί το όρισμά του με τον επόμενο χαρακτήρα στο ρεύμα εισόδου. Στο τέλος του αρχείου το όρισμα ενοποιείται με το end_of_file.

Στα επόμενα φαίνεται η χρήση των παραπάνω κατηγορημάτων:

```
 ?- get_code(L).
|: A
L = 65.
?- get_code(L).
| a
L = 97.
?- get_char(C).
|: a
C = a.
```

```
?- put_code(65).
A
true.
?- put_char(a).
a
true.
?- put_char('A').
A
true.
```

Στην απλή περίπτωση, η χρήση των κατηγορημάτων μπορεί να γίνει σε βρόχους οδηγούμενους από αποτυχία, όπως και στα προηγούμενα παραδείγματα. Για παράδειγμα το ακόλουθο κατηγόρημα εμφανίζει στην οθόνη το περιεχόμενο του αρχείου:

```
print_contents_of_file(FileName):-
  see(FileName),
  repeat,
  get_code(CharCode),
  (
   (CharCode = -1,!,seen)
     ;
   (put_code(CharCode),fail)
  ).
```

Προσοχή θα πρέπει να δοθεί στο γεγονός ότι η συνθήκη τερματισμού είναι η ανάγνωση από το ρεύμα εισόδου της τιμής -1. Στο παραπάνω κατηγόρημα χρησιμοποιήσαμε μια διαφορετική προσέγγιση στην υλοποίηση του βρόχου οδηγούμενου από αποτυχία, που χρησιμοποιεί τον τελεστή διάζευξης. Η συμπεριφορά του κατηγορήματος φαίνεται παρακάτω:

```
?- print_contents_of_file(testReadfile).
name(john).
nick.
12.
[2,3,4].
true.
```

Το κατηγόρημα τυπώνει όλους τους χαρακτήρες του αρχείου (και τις τελείες), αντίθετα με την περίπτωση του κατηγορήματος print_terms_of_file/1, όπου τυπώνονταν μόνο οι όροι.

Η ανάγνωση χαρακτήρων από το ρεύμα εισόδου απαιτεί σχεδόν πάντα την δυνατότητα να μετατραπούν ακολουθίες (λίστες) τέτοιων χαρακτήρων σε όρους και αντίστροφα. Η λειτουργικότητα αυτή προσφέρεται από τα **κατηγορήματα δόμησης/αποδόμησης ατόμων**. Χαρακτηριστικά παραδείγματα είναι:

- name/2, name(Atomic,List), το οποίο πετυχαίνει αν το πρώτο του όρισμα είναι ένας ατομικός όρος (άτομο ή αριθμός) και το δεύτερο η λίστα των κωδικών χαρακτήρων που το απαρτίζουν,
- atom_chars(Atom, List) και number_chars(Number,List), τα οποία πετυχαίνουν όταν η λίστα List περιέχει τους χαρακτήρες από τους οποίους αποτελείται το πρώτο όρισμα, το οποίο πρέπει στην πρώτη περίπτωση να είναι απαραίτητα άτομο και στην δεύτερη αριθμός,
- atom_codes(Atom,ListCodes) και number_codes(Number,ListCodes) που πετυχαίνουν όταν η λίστα List περιέχει τους κωδικούς χαρακτήρων από τους οποίους αποτελείται το πρώτο όρισμα, το οποίο πρέπει στην πρώτη περίπτωση να είναι απαραίτητα άτομο και στην δεύτερη αριθμός. Τα δύο αυτά κατηγορήματα είναι εναλλακτικά του name/2, και ακολουθούν το πρότυπο ISO.

Οι ακόλουθες ερωτήσεις είναι απλά παραδείγματα της λειτουργίας των παραπάνω κατηγορημάτων:

```
?- name(prolog,L).
L = [112, 114, 111, 108, 111, 103].

?- name(L,[108,111,103,105,99]).
L = logic.
```

```
?- atom_codes(abc,L).
L = [97, 98, 99].

?- atom_codes(123,L).
L = [49, 50, 51].

?- atom_codes(L,[49,50,51]).
L = '123'.

?- number_codes(L,[49,50,51]).
L = 123.
```

Οι τελευταίες δύο ερωτήσεις δείχνουν τη διαφορά μεταξύ του atom_codes/2 και του number_codes/2, όπου για την ίδια λίστα κωδικών, το πρώτο επιστρέφει άτομο (‘123’) ενώ το δεύτερο αριθμό (123).

Καθώς με τα παραπάνω κατηγορήματα τα άτομα είναι δυνατό να μετατραπούν σε λίστες χαρακτήρων ή κωδικών χαρακτήρων, είναι δυνατό να χρησιμοποιηθούν τα κλασικά κατηγορήματα διαχείρισης λιστών. Για παράδειγμα, το κατηγόρημα concatenate_atoms/3 πετυχαίνει όταν το τρίτο του όρισμα είναι ένα νέο άτομο που προκύπτει από την παράθεση των ατόμων στα δύο πρώτα ορίσματά του:

```
concatenate_atoms(Atom1,Atom2,Atom):-
  atom_codes(Atom1,L1),
  atom_codes(Atom2,L2),
  append(L1,L2,L),
  atom_codes(Atom,L).
```

και η συμπεριφορά του είναι:

```
?- concatenate_atoms(logic,programming,Atom).
Atom = logicprogramming.
```

Σημείωση: Υπάρχει πλήθος κατηγορημάτων τα οποία αφορούν είσοδο-έξοδο και την δόμηση-αποδόμηση ατόμων, όπως για παράδειγμα get_byte/1, get_code/1, put_byte/1, κλπ. Καθώς η χρήση τους όσον αφορά τις τεχνικές λογικού προγραμματισμού δεν διαφέρει από εκείνη που αναφέρθηκε στα παραπάνω κατηγορήματα, η παρουσίασή τους παραλήφθηκε στο παρόν σύγγραμμα. Ο αναγνώστης μπορεί να αναζητήσει αυτά τα κατηγορήματα στο εγχειρίδιο της έκδοσης της γλώσσας Prolog που θα επιλέξει.

Κατηγορήματα διαχείρισης ρευμάτων κατά το πρότυπο ISO

Τα παραπάνω κατηγορήματα είναι γνωστά σαν Edinburgh-style και ιστορικά είναι εκείνα που χρησιμοποιήθηκαν στις παλαιότερες εκδόσεις της γλώσσας. Το τρέχον ISO πρότυπο της γλώσσας, επιτρέπει τον χειρισμό των ρευμάτων με κατηγορήματα αντίστοιχα με εκείνα που υπάρχουν σε συμβατικές γλώσσες. Τα βασικά κατηγορήματα είναι τα ακόλουθα:

- open(FileName,Mode,StreamID), όπου:
    - FileName είναι το όνομα του αρχείου στο οποίο θα δημιουργηθεί το ρεύμα,
    - Mode είναι ο τρόπος χρήσης του ρεύματος, και είναι ένα από τα read (ανάγνωση), write (εγγραφή/δημιουργία αρχείου), append (προσθήκη στο τέλος ενός ήδη υπάρχοντος αρχείου) ή update (άνοιγμα του αρχείου για εγγραφή και τοποθέτηση του αντίστοιχου δείκτη στην αρχή του αρχείου), και
    - StreamID είναι αναγνωριστικό (identifier) του συγκεκριμένου ρεύματος που δημιουργείται και παίρνει σαν τιμή ένα ειδικό Prolog όρο ο οποίος κατασκευάζεται αυτόματα από την γλώσσα.
- close(StreamID), κλείνει το συγκεκριμένο ρεύμα εισόδου.

Ο χειρισμός των ρευμάτων μέσω των αναγνωριστικών απαιτεί την ύπαρξη κατηγορημάτων τα οποία διαβάζουν και γράφουν σε ένα συγκεκριμένο ρεύμα. Έτσι τα κατηγορήματα read/1, write/1, get_code/1 και put_code/1 έχουν τις αντίστοιχες εκδόσεις τους, με δύο ορίσματα, όπου το πρώτο όρισμα είναι το αναγνωριστικό του ρεύματος εισόδου που έχει δημιουργηθεί με την κλήση του κατηγορήματος open/3.

Έτσι, το κατηγόρημα copy_file/2 που παρουσιάζεται παρακάτω, αντιγράφει χαρακτήρα προς χαρακτήρα τα περιεχόμενα του αρχείου του πρώτου του ορίσματος (InFile), σε ένα (νέο) αρχείο το όνομα του οποίου δίνεται στο δεύτερό του όρισμα (OutFile).

```
copy_file(InFile,OutFile):-
  open(InFile,read,InStream),
  open(OutFile,write,OutStream),
  repeat,
  get_code(InStream,CharCode),
  handle_char_code(OutStream,CharCode),
  !,
  close(InStream),
  close(OutStream).
```

Το κατηγόρημα βασίζεται στο βοηθητικό κατηγόρημα handle_char_code/2, το οποίο όταν ο χαρακτήρας που διαβάστηκε από την είσοδο είναι το τέλος του αρχείου (-1), τότε απλώς πετυχαίνει, ενώ κάθε άλλος χαρακτήρας αποστέλλεται στο ρεύμα εξόδου:

```
handle_char_code(_,-1):-!.
handle_char_code(OutStream,CharCode):-
  put_code(OutStream,CharCode),
  fail.
```

Το ISO πρότυπο υποστηρίζει αρκετά ακόμη κατηγορήματα (open/4, seek/4, set_stream κλπ.), η παροθσίαση των οποίων όμως ξεφεύγει από το πλαίσιο του τρέχοντος συγγράμματος.

9.4 Παράδειγμα Εφαρμογής: Το κοινωνικό Δίκτυο

Στο παράδειγμα του κοινωνικού δικτύου που εξετάστηκε στην αρχή του κεφαλαίου, είδαμε την υλοποίηση των κατηγορημάτων:

- recommend_new_friend(User,Followee), το οποίο προτείνει στο χρήστη User, ένα νέο φίλο Followee
- does_not_follow(User,Followee), το οποίο πετυχαίνει όταν ο χρήστης User, δεν ακολουθεί τον Followee.

Με την χρήση των τεχνικών και κατηγορημάτων τα οποία παρουσιάστηκαν στο παρόν κεφάλαιο, είναι δυνατό να υλοποιηθεί ένα πλήθος ακόμη σχέσεων που αφορούν το συγκεκριμένο πεδίο, αλλά και να διορθωθεί και η συμπεριφορά κάποιων άλλων.

Για παράδειγμα, η σχέση recommend_common_friends/2, που εξετάστηκε στο κεφάλαιο 4, βάσει της οποίας ένας χρήστης Y προτείνεται ως φίλος σε ένα χρήστη X, αν ο Y είναι φίλος κάποιου φίλου Z του X, είναι:

```
recommend_common_friends(X,Y):-
  friends(X,Z),
  friends(Z,Y),
  X \= Y.
```

Ο παραπάνω ορισμός δεν εξετάζει αν ο χρήστης X ήδη ακολουθεί τον Y, ώστε να μην τον προτείνει, δηλαδή δεν εξασφαλίζει ότι προτείνονται νέοι φίλοι στον X. Προσθέτοντας απλά σαν τελευταίο υποστόχο το does_not_follow/2 στο παραπάνω, εξασφαλίζεται η σωστή συμπεριφορά:

```
recommend_common_friends(X,Y):-
  friends(X,Z),
  friends(Z,Y),
  X \= Y,
  does_not_follow(X,Y).
```

Η εύρεση του νεότερου χρήστη του κοινωνικού δικτύου, είναι δυνατό να υλοποιηθεί εύκολα, ακολουθώντας την ίδια τεχνική, που εφαρμόστηκε στην εύρεση του μέγιστου της λίστας. Έτσι, η δηλωτική περιγραφή του κατηγορήματος είναι: "ο χρήστης User του κοινωνικού δικτύου είναι ο νεότερος, αν δεν υπάρχει άλλος χρήστης με χρονολογία γέννησης μεγαλύτερη από εκείνη του User". Για να υλοποιήσουμε το προηγούμενο

ευκολότερα, δημιουργούμε το κατηγόρημα birth_year/2, όπου πετυχαίνει όταν το πρώτο του όρισμα είναι ένας χρήστης του κοινωνικού δικτύου και το δεύτερο είναι η χρονολογία γέννησής του:

```
birth_year(User,Year):-
  user(User,info(_,birthday(_,_,Year))).
```

Για να λειτουργήσει σωστά ένα οποιοδήποτε κατηγόρημα με αυτήν την τεχνική θα πρέπει να μπορεί να επιστρέψει όλες τις δυνατές λύσεις κατά την οπισθοδρόμηση. Το παραπάνω κατηγόρημα ικανοποιεί την συνθήκη αυτή, καθώς μπορεί να επιστρέψει όλους του συνδυασμούς χρηστών-χρονολογιών:

```
?- birth_year(User,Year).
User = ilias,
Year = 1985 ;

User = petros,
Year = 1980 ;

User = demos,
Year = 1980 ;
...
```

Η σχέση νεότερος χρήστης υλοποιείται πλέον από το ακόλουθο κατηγόρημα:

```
younger_user(User):-
  birth_year(User,Year),
  not( (birth_year(_,YearX), YearX > Year) ).
```

Η συμπεριφορά του φαίνεται στην ακόλουθη ερώτηση:

```
?- younger_user(U).
U = sofia .
```

Αν αξιοποιήσουμε τους βρόχους οδηγούμενους από αποτυχία είναι δυνατό να εμφανίσουμε στην οθόνη τα ονόματα των χρηστών και σε ποια ομάδα ανήκουν. Υπενθυμίζεται εδώ, ότι στο Κεφάλαιο 4, υλοποιήθηκε η σχέση belongs_to/2, με πρώτο όρισμα την ομάδα στην οποία ανήκει ο χρήστης του δεύτερου ορίσματος:

```
?- belongs_to(Group,User).
Group = a,
User = ilias ;
Group = a,
User = petros .
...
```

Καθώς επιθυμούμε να εμφανίζουμε τις ομάδες στις οποίες ανήκει ένας συγκεκριμένος χρήστης, πριν προχωρήσουμε στον επόμενο, θα πρέπει να καλούμε το κατηγόρημα με το δεύτερο όρισμα δεσμευμένο στο όνομα ενός χρήστη, με άλλα λόγια απαιτείται μια σύνθετη ερώτηση της μορφής:

```
?- user(User,_),belongs_to(Group,User).
User = ilias,
Group = a ;
User = ilias,
Group = b ;
User = petros,
Group = a ;
...
```

όπου ο πρώτος υποστόχος δεσμεύει τη μεταβλητή User στο όνομα ενός χρήστη. Η οπισθοδρόμηση θα εξαντλήσει όλες τις λύσεις για τον δεύτερο υποστόχο, επιστρέφοντας όλες τις ομάδες στις οποίες ανήκει ο συγκεκριμένος χρήστης που προέκυψε σαν λύση στον πρώτο υποστόχο, πριν προχωρήσει στον επόμενο, κοκ. Βασιζόμενοι στο παραπάνω και εφαρμόζοντας τις (κλασικές) τεχνικές επανάληψης μέσω οπισθοδρόμησης που αναφέρθηκαν στο παρόν κεφάλαιο, η υλοποίηση το κατηγορήματος θα είναι:

```
display_users_groups:-
  user(Name,_),
  nl, write(Name),write(" is in group(s): "),
  belongs_to(Group,Name),
```

```
    write(Group), write(' '),
    fail.

display_users_groups.
```

και η συμπεριφορά του:

```
?- display_users_groups.

ilias is in group(s):a b
petros is in group(s):a c
demos is in group(s):b
nick is in group(s):
...
```

Με τη χρήση κατάλληλων ρευμάτων ο παραπάνω κώδικας μπορεί να εξάγει την πληροφορία σε ένα αρχείο.

9.5 Ένας Απλός Λεκτικός Αναλυτής

Οι λεκτικοί αναλυτές (lexical analyser) αποτελούν το πρώτο στάδιο σε εφαρμογές επεξεργασίας γλωσσών, όπως είναι οι εφαρμογές επεξεργασίας φυσική γλώσσας και οι μεταγλωττιστές (τεχνητές γλώσσες). Στόχος ενός λεκτικού αναλυτή είναι να αναγνωρίζει και να επιστρέφει τις λεκτικές μονάδες μιας γλώσσας από ένα ρεύμα εισόδου όπως για παράδειγμα ένα αρχείο. Οι λεκτικές μονάδες είναι τα βασικά στοιχεία της γλώσσας. Για παράδειγμα σε μια εφαρμογή επεξεργασίας λόγου θα ήταν οι λέξεις και τα σύμβολα στίξης.

Στο παράδειγμα που ακολουθεί ο απλός λεκτικός αναλυτής θα αναγνωρίζει από ένα αρχείο εισόδου:

- λέξεις, που είναι ακολουθίες χαρακτήρων που αποτελούνται από τα γράμματα της αγγλικής αλφαβήτου ή ψηφία (0-9),
- περιορισμένα σημεία στίξης, δηλαδή την τελεία (“.”) και το κόμμα (“,”).

Θεωρώντας ότι το πρόγραμμα διαβάζει κωδικούς χαρακτήρων χρησιμοποιώντας το αντίστοιχο κατηγόρημα get_code/2, θα πρέπει να υπάρχει ένα κατηγόρημα το οποίο αναγνωρίζει χαρακτήρες που ανήκουν σε μια λέξη και ένα κατηγόρημα που πετυχαίνει όταν ο κωδικός είναι σημείο στίξης. Για την πρώτη περίπτωση, το κατηγόρημα is_letter_digit/1 πετυχαίνει όταν το πρώτο όρισμά του είναι ένας κωδικός που αντιστοιχεί σε ένα ψηφίο (0-9) ή αγγλικό χαρακτήρα, κεφαλαίο (A-Z) ή πεζό (a-z):

```
%% digit
is_letter_digit(Code):- Code >= 48, Code =< 57,!.

%% uppercase
is_letter_digit(Code):- Code >= 65, Code =<90,!.

%% lowercase
is_letter_digit(Code):- Code >= 97, Code =<122.
```

Παραπάνω, η χρήση της αποκοπής επιτυγχάνει την αύξηση της απόδοσης του προγράμματος, καθώς δεν γίνονται περιττοί έλεγχοι. Με παρόμοιο τρόπο ορίζεται το κατηγόρημα is_punctuation_mark/1, το οποίο πρέπει να πετυχαίνει σε δύο περιπτώσεις, για τον χαρακτήρα 46 (“.”) και τον χαρακτήρα 44 (“,”), το οποίο υλοποιείται εύκολα με την χρήση του member/2:

```
is_punctuation_mark(Code):- member(Code,[46,44]).
```

Η συνηθισμένη προσέγγιση στην υλοποίηση λεκτικών αναλυτών είναι η ανάγνωση ενός χαρακτήρα από το αρχείο, βάσει του οποίου αποφασίζεται ποια θα είναι η επόμενη κίνηση του λεκτικού αναλυτή. Ο χαρακτήρας αυτός ονομάζεται συνήθως LookAhead. Έτσι, στο παράδειγμα που εξετάζεται αν η τιμή του LookAhead αντιστοιχεί:

- Στο τέλος του αρχείου, τότε η διαδικασία ανάγνωσης/αναγνώρισης σταματά.
- Σε σημείο στίξης, τότε το σημείο στίξης προστίθεται στην λίστα των λεκτικών μονάδων που “επιστρέφει” ο λεκτικός αναλυτής και διαβάζεται ο επόμενος χαρακτήρας.

- Σε γράμμα ή ψηφίο, τότε διαβάζονται οι επόμενοι χαρακτήρες μέχρις ότου ο αναλυτής διαβάσει από το αρχείο ένα χαρακτήρα που δεν είναι χαρακτήρας ή ψηφίο. Όσοι χαρακτήρες αναγνώστηκαν πλην του τελευταίου μετατρέπονται στην λέξη που είναι η λεκτική μονάδα.
- Σε κάθε άλλη περίπτωση, ο χαρακτήρας αγνοείται και διαβάζεται ο επόμενος χαρακτήρας.

Στην περίπτωση ανάγνωσης των λέξεων, καθώς έχει αναγνωσθεί ένας χαρακτήρας παραπάνω, αυτός αποτελεί την επόμενη λεκτική μονάδα (LookAhead) βάσει της οποίας θα συνεχιστεί η λεκτική ανάλυση. Ο παραπάνω ορισμός εύκολα υλοποιείται αναδρομικά, από το κατηγόρημα read_tokens_from/3, όπου το πρώτο του όρισμα είναι το χαρακτηριστικό (ID) του ρεύματος εισόδου, το δεύτερο ο κωδικός χαρακτήρα LookAhead, και το τρίτο η λίστα των λεκτικών μονάδων:

```
%% τέλος του αρχείου
read_tokens_from(_Stream,-1,[]):-!.

%% σύμβολο στίξης
read_tokens_from(Stream,LookAhead,[Punc|Tokens]):-
  is_punctuation_mark(LookAhead),
  !,
  atom_codes(Punc,[LookAhead]),
  get_code(Stream,NextLookAhead),
  read_tokens_from(Stream,NextLookAhead,Tokens).

%% αρχή μιας λέξης
read_tokens_from(Stream,LookAhead,[Word|Tokens]):-
  is_letter_digit(LookAhead),!,
  read_word(Stream,LookAhead,CharList,NextLookAhead),
  !,
  atom_codes(Word,CharList),
  read_tokens_from(Stream,NextLookAhead,Tokens).

%% οποιοσδήποτε άλλος χαρακτήρας
read_tokens_from(Stream,_,Tokens):-
  get_code(Stream,NextLookAhead),
  read_tokens_from(Stream,NextLookAhead,Tokens).
```

Στον δεύτερο και τρίτο κανόνα, οι κωδικοί χαρακτήρων που αναγνώστηκαν μετατρέπονται σε άτομα με την χρήση του κατηγορήματος atom_codes/2. Ενδιαφέρον παρουσιάζει το κατηγόρημα read_word/4, όπου συγκεντρώνει στο τρίτο του όρισμα όλους τους χαρακτήρες από το ρεύμα εισόδου (πρώτο όρισμα) μέχρι να συναντήσει κωδικό που δεν ανήκει στην λέξη (μη χαρακτήρα/αριθμό). Καθώς για να επιτευχθεί αυτό διαβάζεται ένας παραπάνω χαρακτήρας, αυτός επιστρέφεται στο τελευταίο όρισμα και αποτελεί το επόμενο LookAhead:

```
read_word(Stream,LookAhead,[LookAhead|Codes],LastLookAhead):-
  is_letter_digit(LookAhead),
  !,
  get_code(Stream,NextLookAhead),
  read_word(Stream,NextLookAhead,Codes,LastLookAhead).

read_word(_Stream,LookAhead,[],LookAhead).
```

Η όλη διαδικασία ξεκινά με το άνοιγμα του αρχείου εισόδου και την ανάγνωση του πρώτου κωδικού χαρακτήρα:

```
read_tokens(File,Tokens):-
 open(File,read,Stream),
 get_code(Stream,LookAhead),
 read_tokens_from(Stream,LookAhead,Tokens),
  !,
  close(Stream).
```

Έτσι για παράδειγμα, αν καλέσουμε το παραπάνω κατηγόρημα για το αρχείο testReadFile, θα έχει την ακόλουθη συμπεριφορά:

```
?- read_tokens(testReadFile,Tokens)
Tokens = [name,john,'.',nick,'.',12,'.',2,',',3,',',4,'.']
```

Η υλοποίηση που περιγράφηκε αποτελεί ένα απλό λεκτικό αναλυτή, ο οποίος εύκολα μπορεί να επεκταθεί για να καλύψει περισσότερο πολύπλοκες περιπτώσεις.

9.6 Προχωρημένα Θέματα

Η αποκοπή σε Αναδρομικά Κατηγορήματα (member_check/2)

Στη ενότητα 9.2 δόθηκε η υλοποίηση του κατηγορήματος member_check/2, στην οποία εμφανίζεται η αποκοπή στον πρώτο κανόνα:

```
member_check(X,[X|_]):-!.
member_check(X,[_|Rest]):-
      member_check(X,Rest).
```

Το ερώτημα που γεννάται είναι αν θα μπορούσε η αποκοπή να εμφανίζεται σε κάποιο άλλο σημείο του κατηγορήματος. Υπάρχουν δύο άλλες πιθανές θέσεις οι οποίες αναλύονται στα επόμενα.

Η πρώτη περίπτωση είναι να εμφανίζεται η αποκοπή στο τέλος του πρώτου κανόνα:

```
member_check_a(X,[X|_]).
member_check_a(X,[_|Rest]):-
      member_check_a(X,Rest),!.
```

Η ακόλουθη ερώτηση επιστρέφει δύο λύσεις:

```
?- member_check_a(X,[a,a,b,b]).
X = a ;
X = a.
```

Στην περίπτωση αυτή, η Prolog επιστρέφει δύο λύσεις, καθώς η πρώτη προκύπτει από την πρώτη πρόταση του κατηγορήματος, χωρίς να εκτελεστεί η αποκοπή, και η δεύτερη λύση από την δεύτερη πρόταση του κατηγορήματος. Κατά συνέπεια το κατηγόρημα πετυχαίνει δύο φορές, επιστρέφοντας τις αντίστοιχες λύσεις. Το ίδιο συμβαίνει και στην παρακάτω ερώτηση:

```
?- member_check_a(a,[a,a,b,b]).
true ;
true.
```

Ενδιαφέρον παρουσιάζει η ακόλουθη ερώτηση:

```
?- member_check_a(b,[a,a,b,b]).
true.
```

η οποία επιτυγχάνει μόνο μια φορά. Αυτό συμβαίνει γιατί η λύση προκύπτει μετά την αναδρομική κλήση του κατηγορήματος (δεύτερη πρόταση), μετά την επιτυχία της οποίας εκτελείται η αποκοπή, κλαδεύοντας τις όποιες εναλλακτικές λύσεις.

Η τελευταία θέση στην οποία θα μπορούσε να τοποθετηθεί η αποκοπή είναι στην αρχή της δεύτερης πρότασης:

```
member_check_b(X,[X|_]).
member_check_b(X,[_|Rest]):-
  !, member_check_b(X,Rest).
```

Στο παραπάνω κατηγόρημα η ίδια ερώτηση έχει τις ακόλουθες λύσεις:

```
?- member_check_b(X,[a,a,b,b]).
X = a ;
X = a ;
X = b ;
X = b ;
false.
```

Όπως φαίνεται παραπάνω, η αποκοπή δεν έχει καμία επίδραση στον αριθμό των λύσεων, καθώς η αναδρομική λύση βρίσκεται στα δεξιά την αποκοπής. Με άλλα λόγια, η αποκοπή δεν κλαδεύει κανένα μονοπάτι καθώς δεν υπάρχει ούτε υποστόχος στα αριστερά της, ούτε εναλλακτικός κανόνας για το κατηγόρημα, μετά από εκείνον στον οποίο εμφανίζεται η αποκοπή.

Υλοποίηση του κατηγορήματος repeat/0.

Η υλοποίηση ενός κατηγορήματος το οποίο πετυχαίνει πάντα, όσες φορές και αν "φτάσει" σε αυτό η οπισθοδρόμηση είναι σχετικά απλή. Εφόσον επιθυμούμε το κατηγόρημα να πετυχαίνει, θα πρέπει να υπάρχει ένα γεγονός το οποίο να ενοποιείται με το κατηγόρημα, άρα:

```
repeat.
```

Το ζητούμενο όμως είναι να πετυχαίνει το κατηγόρημα όσες φορές και αν επανακληθεί μέσω οπισθοδρόμησης. Η λύση βρίσκεται στην αναδρομή. Πιο συγκεκριμένα αν το κατηγόρημα καλεί αναδρομικά τον εαυτό του, χωρίς να υπάρχει κάποια συνθήκη τερματισμού, τότε κάθε φορά που καλείται θα πετυχαίνει, επεκτείνοντας το δένδρο αναζήτησης κατά ένα επίπεδο ακόμη. Άρα η υλοποίηση του κατηγορήματος repeat/0 είναι:

```
repeat.
repeat:-repeat.
```

Διπλή άρνηση

Σε κάποιες σπάνιες περιπτώσεις είναι δυνατό να απαιτείται να γνωρίζουμε αν δύο όροι είναι ενοποιήσιμοι, χωρίς να γίνονται οι οποιεσδήποτε δεσμεύσεις μεταβλητών. Για παράδειγμα, έστω ότι θα θέλαμε να διαπιστώσουμε ότι δύο όροι a(X,32,n(27)) και a(10, Y, n(Z)) είναι ενοποιήσιμοι, χωρίς όμως οι μεταβλητές X,Y και Z να πάρουν τιμές. Αν χρησιμοποιήσουμε τον τελεστή ενοποίησης για να διαπιστώσουμε το προηγούμενο, τότε ο μηχανισμός εκτέλεσης θα κάνει και τις αντίστοιχες δεσμεύσεις στις μεταβλητές:

```
?- a(X,32,t(27)) = a(10,Y,t(Z)).
X = 10,
Y = 32,
Z = 27.
```

Καθώς η άρνηση δεν δεσμεύει τις μεταβλητές των όρων, θα μπορούσαμε να διατυπώσουμε την παραπάνω ερώτηση ως not(a(X,32,n(27)) = a(10, Y, n(Z))), ώστε να αποφύγουμε τις δεσμεύσεις. Όμως, το πρόβλημα είναι ότι το παραπάνω επιτυγχάνει όταν οι όροι δεν είναι ενοποιήσιμοι και αποτυγχάνει όταν δεν είναι, δηλαδή έχει την ακριβώς αντίθετη συμπεριφορά από την επιθυμητή. Αυτό λύνεται με ένα ακόμη not/1, δηλαδή εφαρμόζοντας διπλή άρνηση στο ερώτημα:

```
?- not(not(a(X,32,t(27)) = a(10,Y,t(Z)))).
true.
```

Σε περίπτωση που οι όροι δεν είναι ενοποιήσιμοι, η διπλή άρνηση αποτυγχάνει:

```
?- not( not(a(X,32,t(27)) = a(10,Y,t(11)))).
false.
```

9.7 Σημαντικές Παρατηρήσεις

Ο μηχανισμός οπισθοδρόμησης και ο έλεγχός του αποτελούν ισχυρότατα εργαλεία ανάπτυξης τεχνικών. Ιδιαίτερη προσοχή θα πρέπει όμως να δίνεται στην απαλοιφή ελέγχων με την χρήση της αποκοπής. Για παράδειγμα, έστω ο ακόλουθος ορισμός για το μέγιστο δύο αριθμών:

```
max1(X,Y,X):- X>=Y.
max1(X,Y,Y):- X=<Y.
```

και ο αντίστοιχος ορισμός με τη χρήση της αποκοπής:

```
max2(X,Y,X):- X>=Y, !.
max2(X,Y,Y).
```

Αν το τρίτο όρισμα είναι μεταβλητή, τότε οι δύο παραπάνω εκδοχές λειτουργούν όπως είναι αναμενόμενο, για παράδειγμα:

```
?- max2(3,2,Max).
Max = 3.

?- max2(2,4,Max).
Max = 4.
```

Το πρόβλημα δημιουργείται στην δεύτερη εκδοχή όταν το τελευταίο όρισμα είναι δεσμευμένο σε μια τιμή:

```
?- max2(2,1,1).
true.
```

Αυτό συμβαίνει γιατί ο πρώτος κανόνας του max2/2 αποτυγχάνει κατά την ενοποίηση της κεφαλής, χωρίς ποτέ να φτάσει στον έλεγχο X >=Y. Έτσι καθώς ο δεύτερος κανόνας δεν περιέχει ελέγχους, πετυχαίνει δίνοντας λάθος αποτέλεσμα. Αντίθετα στο max1/2 η ύπαρξη ελέγχων δεν επιτρέπει κάτι τέτοιο:

```
?- max1(2,1,1).
false.
```

Η μη προσεκτική χρήση της αποκοπής μπορεί να δημιουργήσει σημαντικά σφάλματα στον κώδικα και κατά συνέπεια απαιτείται σημαντική προσοχή στην χρήση της. Σε κάθε περίπτωση ο έλεγχος των κατηγορημάτων θα πρέπει να περιλαμβάνει "ακραίες" περιπτώσεις όπως η παραπάνω.

Βιβλιογραφία

Τα εξωλογικά χαρακτηριστικά της γλώσσας αποτελούν αντικείμενο όλων των "κλασικών" βιβλίων αναφοράς για τη γλώσσα προγραμματισμού Prolog, όπως τα (Clocksin and Mellish, 2003), (Bratko, 2011), (Sterling and Shapiro, 1994) (O'Keefe, 1990) και (Covington et al., 1988).

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Clocksin, W. F. and Mellish C. S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

Covington, M. A. and Nute, D. and Vellino A. (1988). *Prolog Programming in Depth*. Scott Foresman & Co. ISBN 9780673186591.

O'Keefe, R. A. (1990), *The craft of Prolog*, Cambridge, Mass: MIT Press

Sterling, L. and Shapiro, E. Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass: MIT Press.

Άλυτες Ασκήσεις

9.1 Έστω το Prolog κατηγόρημα adjust_list(Limit,Inc,List,Rest) (adjust_list/4) το οποίο δοθείσας μιας λίστας ακεραίων List, αυξάνει κατά Inc όλα τα μέλη της λίστας που είναι μικρότερα του Limit. Για παράδειγμα:

```
?- adjust_list(5, 2, [1, 3, 5, 7, 9, 10, 11], R).
R = [3, 5, 5, 7, 9, 10, 11]
?- adjust_list(10, 20, [1, 3, 5, 7, 9, 10, 11], R).
R = [21, 23, 25, 27, 29, 10, 11]
```

Να δώσετε τον αναδρομικό ορισμό του κατηγορήματος κάνοντας όσο το δυνατό λιγότερους ελέγχους,

9.2 Χρησιμοποιώντας τον τελεστή ! να ορίσετε ένα κατηγόρημα set_diff/3 το οποίο υλοποιεί αφαίρεση συνόλων στα δύο πρώτα του κατηγορήματα τα οποία είναι λίστες. Για παράδειγμα:

```
?- set_diff([1,2,3,4],[1,2],L).
L = [3, 4]
true
?- set_diff([1,2,3,4],[1,2,5,6],L).
L = [3, 4]
true
```

```
?- set_diff([1,2,3,4],[1,2,5,3,6],L).
L = [4]
true
```

9.3 Να ορίσετε το κατηγόρημα set_union(List1,List2,List3) χρησιμοποιώντας την αποκοπή (!), το οποίο υλοποιεί ένωση συνόλων. Το κατηγόρημα επιτυγχάνει όταν η λίστα List3 είναι η ένωση των στοιχείων της List1 (σύνολο 1) που δεν υπάρχουν στην List2 (σύνολο 2), με την List2. Για παράδειγμα:

```
?- set_union([a, b, c], [d, e, f, a], L).
L = [b, c, d, e, f, a]

?- set_union([a, b], [d, e, f, b, a], L).
L = [d, e, f, b, a]

?- set_union([a, b, c], [d, e, f, b], L).
L = [a, c, d, e, f, b]
```

9.4 Να ορίσετε ένα κατηγόρημα unique_element(X,List), το οποίο επιτυγχάνει όταν το X είναι στοιχείο της λίστας List και εμφανίζεται μόνο μια φορά. Να χρησιμοποιήσετε το κατηγόρημα not/1. Για παράδειγμα

```
?- unique_element(X, [a, b, c, c, b, d]).
X = a ;
X = d ;
false

?- unique_element(X, [a, a, b, c, c, b]).
false
```

9.5 Να ορίσετε ένα κατηγόρημα proper_set(List), το οποίο επιτυγχάνει όταν η λίστα List περιέχει μόνο μοναδικά στοιχεία, είναι δηλαδή σύνολο. Να χρησιμοποιήσετε τον ορισμό του unique_element/2 που δώσατε στην άσκηση 9.4 και το not/1. Για παράδειγμα:

```
?- proper_set([1, 2, 1, 3]).
false
?- proper_set([1, 2, 3]).
true
?- proper_set([a, b, c, d]).
true
?- proper_set([a, b, c, d, d]).
false
```

9.6 Έστω το Prolog κατηγόρημα exclude_range(Low,High,List,NewList) το οποίο επιτυγχάνει όταν δοθείσας μιας λίστας ακεραίων List (3ο όρισμα), η λίστα NewList περιέχει όλους τους ακεραίους που ΔΕΝ ανήκουν στο διάστημα που ορίζουν τα πρώτα δύο ορίσματα Low και High. Για παράδειγμα:

```
?- exclude_range(2, 10, [1, 2, 3, 5, 10, 11, 12, 15], List).
List = [1, 11, 12, 15]
?- exclude_range(2, 10, [2, 3, 8, 10], List).
List = []
```

Να δώσετε τον αναδρομικό ορισμό του κατηγορήματος κάνοντας όσο το δυνατό λιγότερους ελέγχους,

9.7 Να υλοποιήσετε στην Prolog ένα κατηγόρημα το οποίο υλοποιεί ένα πρόγραμμα αλληλεπίδρασης με τον χρήστη που ακολουθεί τον παρακάτω διάλογο:

```
?- dialogue.
Give a number (or exit to finish): 4.
```

```
1. Calculate square
2. Calculate factorial
3. Calculate the sum
Choice: 2.
The factorial is 4! = 24
Give a number(or exit to finish): exit.
true
```

Να δώσετε δύο υλοποιήσεις

α) με αναδρομή

β) με την χρήση βρόχου οδηγούμενου από αποτυχία.

9.8 Να υλοποιήσετε ένα κατηγόρημα count_lines/2, count_lines(FileName,N), το οποίο ενοποιεί το δεύτερο του όρισμα Ν με τον αριθμό γραμμών του αρχείου FileName. Για διευκόλυνσή σας, θεωρείστε ότι ο χαρακτήρας αλλαγής γραμμής είναι το 10 (Line Feed).

ΚΕΦΑΛΑΙΟ 10: Μετα-Λογικός Προγραμματισμός - Κατηγορήματα ανώτερης τάξης

Λέξεις Κλειδιά:
Διαχείριση όρων, Μεταβλητή κλήση, Διαχείριση προτάσεων, Διαχείριση Συνόλου Λύσεων, Δυναμικός Μετα-προγραμματισμός

Περίληψη
Τα μετα-λογικά χαρακτηριστικά της γλώσσας Prolog αυξάνουν σημαντικά τη δυνατότητα ανάπτυξης πρακτικών προγραμμάτων. Στην κατηγορία αυτή ανήκουν τα κατηγορήματα της διαχείρισης των όρων της Prolog κατά την διάρκεια της εκτέλεσης, όπως είναι αυτά της εξέτασης του τύπου των όρων, της σύγκρισης των όρων για ισότητα ή ανισότητα, και της σύνθεσης και διάσπασης όρων, τα οποία και παρουσιάζονται στο παρόν κεφάλαιο με την χρήση απλών παραδειγμάτων αλλά και περισσότερο πολύπλοκων λογικών προγραμμάτων. Το κεφάλαιο, παρουσιάζει επίσης μια επέκταση της Prolog που επιτρέπει την κωδικοποίηση λογικών πέραν της κατηγορηματικής λογικής πρώτης τάξης, την μεταβλητή κλήση, η οποία εκμεταλλευόμενη την συντακτική ισομορφία των κατηγορημάτων και των σύνθετων όρων, επιτρέπει την δημιουργία κατά την εκτέλεση ερωτημάτων προς απόδειξη καθώς και την κλήση τους. Η μετατροπή του προγράμματος κατά τη διάρκεια της εκτέλεσης είναι το επόμενο αντικείμενο το οποίο πραγματεύεται το κεφάλαιο, η οποία προσφέρει ένα απλό τρόπο μεταβολής του ίδιου του εκτελέσιμου κώδικα. Τέλος, στο κεφάλαιο παρουσιάζονται τα κατηγορήματα συλλογής όλων των εναλλακτικών λύσεων μιας κλήσης σε μία λίστα. Το κεφάλαιο κλείνει με ένα ολοκληρωμένο παράδειγμα χρήσης όλων των μετα-λογικών χαρακτηριστικών σε ένα πρόγραμμα διαχείρισης βάσης δεδομένων. Επίσης παρουσιάζεται ένα σύντομο πλην όμως ενδεικτικό παράδειγμα της προχωρημένης δυνατότητας της γλώσσας Prolog για την ανάπτυξη διερμηνέων άλλων γλωσσών προγραμματισμού, ακόμα και του εαυτού της!

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός να:

- κατανοεί τις έννοιες μετα-λογική, λογική ανώτερης τάξης και μετα-προγραμματισμός,
- ελέγξει τον τύπο των όρων μέσα σε ένα πρόγραμμα Prolog και να τους συγκρίνει μεταξύ τους,
- διασπάσει και να συνθέσει σύνθετους όρους,
- χρησιμοποιεί τη μεταβλητή κλήση για την υλοποίηση δυναμικών λογικών προγραμμάτων,
- μεταβάλει δυναμικά τις λογικές προτάσεις ενός Prolog προγράμματος,
- συλλέγει σε λίστες τις πολλαπλές απαντήσεις σε ένα ερώτημα / κλήση.

Παράδειγμα κίνητρο

Καθώς τα κατηγορήματα μετα-λογικής κατατάσσονται σε πολλές κατηγορίες, κάθε κατηγορία έχει δικά της παραδείγματα που μπορούν να καταδείξουν την αναγκαιότητά τους. Συνεπώς, σε κάθε ενότητα του τρέχοντος κεφαλαίου παρουσιάζεται και το αντίστοιχο παράδειγμα - κίνητρο. Αυτό που μπορεί να ειπωθεί σε γενικές γραμμές είναι ότι τα μετα-λογικά κατηγορήματα είναι απαραίτητα στις περιπτώσεις που χρειάζεται ο προγραμματιστής να μεταχειριστεί δυναμικά (at run time) τα βασικά στοιχεία της γλώσσας της λογικής ως δεδομένα. Αυτό περιλαμβάνει τους όρους και τα ορίσματα των κατηγορημάτων, τις προτάσεις του προγράμματος, τις κλήσεις και τα αποτελέσματα των κλήσεων. Κάθε μία από τις προηγούμενες περιπτώσεις δημιουργεί και την αντίστοιχη κατηγορία κατηγορημάτων, όπως παρουσιάστηκαν στην περίληψη.

Μονοπάτι Μάθησης

Στην πραγματικότητα, κάθε ενότητα αυτού του κεφαλαίου υλοποιεί μια διαφορετική δυνατότητα μετα-λογικής της γλώσσας Prolog, συνεπώς δεν υφίσταται κάποια αυστηρή σειρά διαδοχής των ενοτήτων κατά την ανάγνωση. Παρόλα αυτά κάποιες ενότητες παρουσιάζουν κάποια παραδείγματα κώδικα με χρήση ενσωματωμένων κατηγορημάτων, τα οποία βασίζονται σε παραδείγματα προηγούμενων ενοτήτων. Συνεπώς, για την καλύτερη κατανόηση αυτών των παραδειγμάτων θα ήταν προτιμότερο (όχι όμως εντελώς απαραίτητο), η ενότητα 10.3 να αναγνωσθεί πριν την 10.4, η 10.4 και 10.9 πριν την 10.5, η 10.6 πριν την 10.7, και η 10.8 μετά από όλες τις προηγούμενες. Η δυνατότητα ανάπτυξης διερμηνέων άλλων γλωσσών στην Prolog αποτελεί προχωρημένο χαρακτηριστικό της γλώσσας, το οποίο θα μπορούσε να παραληφθεί σε μια πρώτη ανάγνωση.

10.1 Εισαγωγή στα μετα-λογικά χαρακτηριστικά της Prolog

Τα μετα-λογικά χαρακτηριστικά της γλώσσας Prolog αυξάνουν σημαντικά την δυνατότητα ανάπτυξης πρακτικών προγραμμάτων, με έμφαση στην δυναμικότητα και προσαρμοστικότητα του κώδικα στις παραμέτρους εισόδου του (context-aware) και εν τέλει στην ανάπτυξη ευφυών εφαρμογών. Στην κατηγορία αυτή ανήκουν διάφορα είδη ενσωματωμένων κατηγορημάτων που προσδίδουν διάφορες εξαιρετικές προγραμματιστικές δυνατότητες, μερικές από τις οποίες δεν υπάρχουν σε καμία άλλη γλώσσα.

Η πρώτη ομάδα τέτοιων κατηγορημάτων είναι τα κατηγορήματα διαχείρισης όρων, τα οποία δίνουν την δυνατότητα του δυναμικού (at run-time) ελέγχου τύπου και σύγκρισης όρων, αυξάνοντας την ευρωστία (robustness) των προγραμμάτων σε μη-προσδοκώμενα δεδομένα, καθώς και την προσαρμοστικότητά τους σε διαφορετικά είδη δεδομένων εισόδου. Επίσης, υπάρχει η δυνατότητα σύνθεσης (και διάσπασης) σύνθετων όρων, κάτι που χρησιμοποιείται κυρίως ως προεργασία για την εκμετάλλευση των υπολοίπων μετα-λογικών χαρακτηριστικών της Prolog, όπως για παράδειγμα η μεταβλητή κλήση.

Η μεταβλητή κλήση αποτελεί μία εξαιρετικά προηγμένη δυνατότητα της Prolog, σύμφωνα με την οποία η κλήση στο σώμα ενός κανόνα ή στις αρχικές κλήσεις ενός σύνθετου ερωτήματος μπορεί να είναι μία μεταβλητή αντί ενός ατομικού τύπου. Κατά την στιγμή της εκτέλεσης η μεταβλητή αυτή θα πρέπει να έχει ενοποιηθεί με έναν σύνθετο όρο, ο οποίος τελικά θα πάρει την θέση και θα παίξει τον ρόλο του ατομικού τύπου, εκμεταλλευόμενος την συντακτική ισομορφία των σύνθετων όρων με τους ατομικούς τύπους. Η μεταβλητή κλήση, σε συνδυασμό με τα κατηγορήματα σύνθεσης όρων, επιτρέπουν την κωδικοποίηση λογικών ανώτερης τάξης, καθώς είναι πλέον δυνατή η δημιουργία και η κλήση ερωτημάτων προς απόδειξη κατά την εκτέλεση του προγράμματος.

Η τρίτη κατηγορία κατηγορημάτων είναι αυτά της δυναμικής διαχείρισης λογικών προτάσεων, δηλαδή της προσθήκης ή της διαγραφής γεγονότων και κανόνων σε ένα λογικό πρόγραμμα κατά την διάρκεια της εκτέλεσής του! Με τον τρόπο αυτό τα προγράμματα μπορούν να τροποποιούν τον εαυτό τους και να προσαρμόζονται ανοίγοντας δρόμους για την ανάπτυξη ευφυών εφαρμογών.

Τέλος, μια άλλη κατηγορία κατηγορημάτων του είδους είναι τα κατηγορήματα “διαχείρισης συνόλου λύσεων” ή πιο απλά “συλλογής λύσεων”, τα οποία δίνουν την δυνατότητα συλλογής όλων των ενοποιήσεων των μεταβλητών που υπάρχουν σε ένα ερώτημα, δηλαδή όλων των απαντήσεων, σε μία λίστα όρων. Στην συνέχεια η λίστα αυτή μπορεί να υποστεί επεξεργασία μέσω “συμβατικών” κατηγορημάτων χειρισμού λιστών που έχουν παρουσιαστεί σε όλα τα προηγούμενα κεφάλαια, αλλά κυρίως στο Κεφάλαιο 7. Με την χρήση αυτών των κατηγορημάτων γίνεται εφικτή στην Prolog η υλοποίηση προγραμμάτων για την δημιουργία ερωτημάτων και αλγορίθμων συνόλων δεδομένων, όπως στα συστήματα βάσεων δεδομένων.

Σημειώνεται ότι σε πολλά βιβλία σχετικά με την γλώσσα Prolog κάποια από τα κατηγορήματα που εμφανίζονται ως μετα-λογικά στο παρόν κεφάλαιο, αντιμετωπίζονται ως έξτρα-λογικά, όπως είναι για παράδειγμα τα κατηγορήματα προσθήκης - διαγραφής λογικών προτάσεων. Προφανώς κάποια από τα κατηγορήματα μπορεί να έχουν και μετα-λογικές και έξτρα-λογικές θεωρήσεις, όμως στο παρόν βιβλίο επιλέξαμε να παρουσιάσουμε συγκεντρωμένα όλα τα κατηγορήματα που αντιμετωπίζουν και χειρίζονται τα

στοιχεία ενός λογικού προγράμματος (κώδικα) ως δεδομένα και αυτό συμπεριλαμβάνει σαφώς και την προσθαφαίρεση λογικών προτάσεων σε ένα λογικό πρόγραμμα.

10.2 Εξέταση τύπου όρων

Στην πρώτη ομάδα μετα-λογικών κατηγορημάτων εντάσσονται τα κατηγορήματα διαχείρισης όρων, τα οποία δίνουν την δυνατότητα του δυναμικού (at run-time) ελέγχου, σύγκρισης, σύνθεσης και διάσπασης όρων. Στην τρέχουσα ενότητα θα παρουσιαστούν τα κατηγορήματα εξέτασης τύπου όρων, στην επόμενη της σύγκρισης όρων, ενώ τα κατηγορήματα σύνθεσης - διάσπασης όρων στην μεθεπόμενη.

Τα κατηγορήματα εξέτασης τύπου όρων αυξάνουν την ευρωστία (robustness) των προγραμμάτων σε μη-προσδοκώμενα δεδομένα, καθώς και την προσαρμοστικότητά τους σε διαφορετικά είδη δεδομένων εισόδου. Ως παράδειγμα μπορεί να δοθεί το κατηγόρημα nth_member/3, το οποίο επιστρέφει το ν-οστό στοιχείο μιας λίστας. Υπάρχουν 2 βασικοί τρόποι χρήσης ενός τέτοιου κατηγορήματος:

Ο πρώτος αφορά στην περίπτωση να γνωρίζουμε ποια είναι η ν-οστή θέση και θέλουμε να επιστραφεί το αντίστοιχο στοιχείο της λίστας:

```
?- nth_member(3,A,[a,b,c,d,e,c]).
A = c
```

και ο δεύτερος να γνωρίζουμε ποιο στοιχείο ψάχνουμε και θέλουμε να επιστραφεί η θέση στην οποία βρίσκεται:

```
?- nth_member(X,c,[a,b,c,d,e,c]).
X = 3 ;
X = 6 ;
false
```

Ας θεωρήσουμε τον πρώτο τρόπο χρήσης. Ο παρακάτω κώδικας υλοποιεί την εύρεση του ν-οστού στοιχείου μιας λίστας, εφόσον είναι γνωστό το N, καθώς μειώνει τον αριθμό αυτό κατά μία μονάδα, αφαιρώντας ταυτόχρονα ένα στοιχείο από την κεφαλή της λίστας, έως ότου ο αριθμός γίνει **1**:

```
nth_member(1,E,[E|_]).
nth_member(N,E,[_|T]) :-
  N > 1,
  N1 is N - 1,
  nth_member(N1,E,T).
```

Ο παραπάνω κώδικας όμως δεν είναι δυνατόν να χρησιμοποιηθεί για τον δεύτερο τρόπο κλήσης, όταν δεν είναι γνωστό το N, καθώς τα ενσωματωμένα κατηγορήματα N>1 και N1 is N-1 δεν μπορούν να εκτελεστούν όταν η μεταβλητή N είναι ελεύθερη, όπως είδαμε στο κεφάλαιο 6. Στην περίπτωση αυτή ο αλγόριθμος πρέπει να αλλάξει, "ψάχνοντας" το ζητούμενο στοιχείο και όταν το βρει να "μετρήσει" πόσες φορές χρειάστηκε να αφαιρέσει στοιχεία από την κεφαλή της λίστας:

```
nth_member(1,E,[E|_]).
nth_member(N,E,[_|T]) :-
  nth_member(N1,E,T),
  N is N1 + 1.
```

Είναι δυνατόν όμως ο συγκεκριμένος κώδικας να καλύψει και τον πρώτο τρόπο κλήσης; Η απάντηση είναι πως ναι, καθώς δοκιμάζονται διαδοχικά να επιστραφούν όλα τα στοιχεία της λίστας, έως ότου επαληθευθεί η έκφραση N is N1 + 1. Στην περίπτωση αυτή όμως ο συγκεκριμένος κώδικας δεν είναι καθόλου αποδοτικός, καθώς μπορεί να κάνει πάρα πολλές δοκιμές και μάλιστα χρειάζεται να εκτελέσει τις δοκιμές αυτές αφού εκτελέσει την αναδρομική κλήση, με συνέπεια να "καταναλώνει" χώρο για τις τιμές των μεταβλητών αυτών στην μνήμη. Για πολύ μεγάλες λίστες, π.χ. 10000 στοιχείων, η μνήμη θα γεμίσει και η Prolog θα τερματίσει την εκτέλεση με μήνυμα λάθους. Συνεπώς, η πρώτη εκδοχή του κώδικα είναι πολύ πιο αποδοτική για τον πρώτο τρόπο κλήσης, ενώ η δεύτερη εκδοχή είναι "μονόδρομος" για τον δεύτερο τρόπο κλήσης. Η απάντηση στο ποιον κώδικα τελικά να χρησιμοποιήσουμε η απάντηση είναι "και τους δύο, ανάλογα με την περίπτωση". Η διάκριση γίνεται ανάλογα με το αν το όρισμα N έχει ή όχι τιμή κατά την διάρκεια της κλήσης (at run time). Για τον σκοπό αυτό πρέπει να χρησιμοποιηθούν τα ενσωματωμένα κατηγορήματα var/1 και nonvar/1 που

παρουσιάστηκαν Κεφάλαιο 4, προκειμένου να χρησιμοποιηθούν οι δύο διαφορετικές εκδοχές της αναδρομικής διαδικασίας, ανάλογα με το αν το όρισμα N είναι σταθερά (nonvar/1) ή μεταβλητή (var/1) κατά την στιγμή της εκτέλεσης:

```
nth_member(1,E,[E|_]).
nth_member(N,E,[_|T]) :-
  var(N),
  nth_member(N1,E,T),
  N is N1 + 1.
nth_member(N,E,[_|T]) :-
  nonvar(N),
  N > 1,
  N1 is N - 1,
  nth_member(N1,E,T).
```

Συνεπώς, τα κατηγορήματα var/1 και nonvar/1 χαρακτηρίζονται ως μετα-λογικά γιατί σχετίζονται με την κατάσταση της τρέχουσας απόδειξης και μεταχειρίζονται τις μεταβλητές ως να ήταν αντικείμενα της γλώσσας της λογικής. Στην ίδια κατηγορία μπορούμε να εντάξουμε και τα υπόλοιπα κατηγορήματα που εξετάζουν τον τύπο ενός όρου, όπως π.χ. integer/1, atom/1, compound/1, κλπ., τα οποία παρουσιάστηκαν στο Κεφάλαιο 4, καθώς και αυτά όταν το όρισμά τους είναι ελεύθερη μεταβλητή κατά την διάρκεια της εκτέλεσης αποτυγχάνουν.

10.3 Σύγκριση όρων

Στην ίδια ακριβώς περίπτωση εντάσσονται και τα κατηγορήματα σύγκρισης όρων, ιδιαίτερα όρων μη-πλήρως-ορισμένων, δηλαδή όρων που περιέχουν σε κάποιο όρισμά τους(ή είναι) ελεύθερες μεταβλητές. Στο Κεφάλαιο 6 παρουσιάστηκαν σύντομα τα συνηθισμένα κατηγορήματα >, <, >=, =< για τη σύγκριση αριθμών, πλην όμως αυτά ισχύουν μόνο για αριθμούς ή αριθμητικές παραστάσεις χωρίς μεταβλητές. Επίσης, στο Κεφάλαιο 4 παρουσιάστηκαν οι τελεστές = και \= οι οποίοι μπορούν να χρησιμοποιηθούν για σύγκριση όρων, όταν όμως οι όροι είναι πλήρως ορισμένοι (grounded). Τα κατηγορήματα που θα παρουσιαστούν εδώ ισχύουν για οποιουσδήποτε όρους, είτε είναι απλοί, συμπεριλαμβανομένων των αριθμών, είτε σύνθετοι είτε περιέχουν ελεύθερες μεταβλητές ή όχι. Σημειωτέον, τα κατηγορήματα αυτά θεωρούνται μετα-λογικά μόνο στην περίπτωση που περιέχονται στους όρους ελεύθερες μεταβλητές. Σε διαφορετική περίπτωση η σύγκριση όρων είναι μέσα στην εκφραστική ικανότητα της κατηγορηματικής λογικής πρώτης τάξης.

Ως πρώτο κατηγόρημα θα παρουσιαστεί αυτό του ελέγχου της ισότητας δύο όρων ==/2. Το κατηγόρημα =/2 μπορεί να χρησιμοποιηθεί για έλεγχο ισότητας όρων μόνο αν οι δύο όροι είναι πλήρως ορισμένοι:

```
?- f(1,2) = f(1,2).
true
?-f(1,2) = g(1,2).
false
```

Όταν για παράδειγμα ο ένας από τους δύο όρους είναι μεταβλητή (ή και οι δύο), τότε η μεταβλητή (ή μεταβλητές) ενοποιείται με τον όρο και φυσικά το κατηγόρημα επιτυγχάνει:

```
?- X = 2.
X = 2
?- X = Y.
X = Y
```

Ο λόγος είναι ότι το κατηγόρημα =/2 ελέγχει αν δύο όροι είναι ενοποιήσιμοι και πραγματοποιεί την ενοποίηση σε καταφατική απάντηση. Υπάρχει το ενσωματωμένο κατηγόρημα ==/2 το οποίο ελέγχει αν δύο όροι είναι ίσοι την στιγμή της εκτέλεσης και δεν ελέγχει ούτε πραγματοποιεί ενοποίηση:

```
?- X == 2.
false
?- X == Y.
false
```

Όπως φαίνεται και στα παραδείγματα, το κατηγόρημα ==/2 ελέγχει αν τη στιγμή της εκτέλεσής του οι δύο όροι είναι πανομοιότυποι και όχι αν εν δυνάμει μπορούν να ενοποιηθούν. Η συμπεριφορά του συγκεκριμένου κατηγορήματος διαφοροποιείται από του =/2 όταν μόνο ένας ή και οι δύο όροι είναι ή έχουν μέσα τους ελεύθερες μεταβλητές. Στην περίπτωση που και οι δύο όροι είναι πλήρως ορισμένοι (grounded) είτε επειδή είναι σταθερές, είτε επειδή περιέχουν μόνο σταθερές, είτε επειδή περιέχουν μεταβλητές που έχουν προηγουμένως ενοποιηθεί με σταθερές ή πλήρως ορισμένους όρους, τότε η συμπεριφορά των δύο κατηγορημάτων είναι ίδια:

```
?- f(1,2) == f(1,2).
true
?-f(1,2) == g(1,2).
false
?- X = 1, Y = 2, f(X,Y) == f(1,2).
X = 1, Y = 2
?- X = Y, X == Y.
X = Y
```

Αντίστοιχα με το \=, υπάρχει και το κατηγόρημα \==/2, το οποίο επιτυγχάνει αν οι δύο όροι δεν είναι πανομοιότυποι:

```
?- X \== 2.
true
?- X \== Y.
true
?- f(1,2) \== f(1,2).
false
```

Ένα παράδειγμα που αναδεικνύει την χρησιμότητα των παραπάνω είναι το κατηγόρημα count/3 που μετράει πόσες φορές ένα στοιχείο υπάρχει μέσα σε μία λίστα:

```
count(_,[],0).
count(A,[A|L],N) :-
  count(A,L,N1),
  N is N1 + 1.
count(A,[H|L],N) :-
  A \= H,
  count(A,L,N).
```

Στον δεύτερο κανόνα ελέγχεται αν το στοιχείο Α ενοποιείται με την κεφαλή της λίστας, και αν ναι τότε αυξάνεται ο μετρητής. Αν όλα τα στοιχεία της λίστας είναι σταθερές (ή πλήρως ορισμένοι όροι), τότε ο παραπάνω ορισμός συμπεριφέρεται όπως θα περιμέναμε:

```
?- count(a,[a,b,c,a,d],N).
N = 2
```

Αν όμως στην λίστα περιέχονται μεταβλητές, τότε επειδή ο παραπάνω ορισμός στηρίζεται στην ενοποίηση, τα αποτελέσματα είναι λίγο διαφορετικά:

```
?- count(a,[a,X,b,a,Y,d],N).
X = a, Y = a, N = 4
```

Δηλαδή, ο δεύτερος κανόνας επιβάλλει την ενοποίηση των μεταβλητών οι οποίες περιέχονται στην λίστα με το αναζητούμενο στοιχείο και προφανώς προσμετρά και αυτά τα στοιχεία στο τελικό αποτέλεσμα. Το αποτέλεσμα αυτό δεν είναι λάθος, αλλά ίσως δεν είναι το επιδιωκόμενο. Για να μην πραγματοποιηθεί η ενοποίηση θα πρέπει να χρησιμοποιηθούν τα δύο κατηγορήματα σύγκρισης όρων αυτής της ενότητας:

```
count(_,[],0).
count(A,[H|L],N) :-
  A == H,
  count(A,L,N1),
  N is N1 + 1.
count(A,[H|L],N) :-
  A \== H,
  count(A,L,N).
```

Έτσι, δεν πραγματοποιείται ενοποίηση των μεταβλητών στοιχείων της λίστας και προσμετρώνται μόνο τα πραγματικά ίδια στοιχεία:

```
?- count(a,[a,X,b,a,Y,d],N).
N = 2
```

Εναλλακτικά, το ίδιο ακριβώς αποτέλεσμα θα μπορούσε να επιτευχθεί με την αλλαγή της κλήσης στο κατηγόρημα ==/2 του δεύτερου κανόνα, με την ακόλουθη γραμμή:

```
count(A,[H|L],N) :-
  nonvar(A), nonvar(H), A = H,
  ...
```

Αυτό βέβαια δεν σημαίνει πως το κατηγόρημα ==/2 μπορεί σε οποιαδήποτε περίπτωση να αντικατασταθεί με τις παραπάνω τρεις κλήσεις. Για παράδειγμα:

```
?- X == X.
true
?- nonvar(X), nonvar(X), X = X.
false
```

Διάταξη όρων

Τέλος, υπάρχουν και τα κατηγορήματα σύγκρισης - διάταξης όρων: @>, @>=, @=<, @< με την ίδια σημασία με τα αντίστοιχα κατηγορήματα σύγκρισης αριθμών. Η διαφορά έγκειται στο γεγονός ότι οι συγκρινόμενοι όροι μπορεί να είναι οποιασδήποτε μορφής, απλοί ή σύνθετοι, με ή χωρίς μεταβλητές, αριθμοί ή σύμβολα, και όχι κατ' ανάγκην ίδιοι μεταξύ τους στο ίδιο κατηγόρημα σύγκρισης. Έτσι τίθεται το θέμα διάταξης των όρων, η οποία έχει ως εξής:

Μεταβλητές < *Αριθμοί* < *Συμβολοσειρές* < *Άτομα* < *Σύνθετοι όροι*

Για παράδειγμα:

```
?- 2 @> X.
true
?- f(1,2) @> nick.
true
```

Στο σημείο αυτό, η μοναδική πρακτικά χρήσιμη λεπτομέρεια είναι η σειρά διάταξης των σύνθετων όρων, η οποία συγκρίνει πρώτα (αριθμητικά) την τάξη των δύο όρων, στη συνέχεια λεξικογραφικά τα συναρτησιακά τους σύμβολα, και τέλος συγκρίνει ένα-προς-ένα τα ορίσματα των δύο όρων από αριστερά προς τα δεξιά, όπως φαίνεται και από τα παρακάτω παραδείγματα:

```
?- a(2,2) @> f(3).
true
?- f(1,2) @> a(2,2).
true
?- f(2,2) @> f(1,2).
true
?- f(2,2) @> f(2,1).
true
```

10.4 Σύνθεση - Διάσπαση Σύνθετων Όρων

Η Prolog παρέχει τη δυνατότητα σύνθεσης και διάσπασης σύνθετων όρων. Με αυτόν τον τρόπο οι σύνθετοι όροι μπορούν να κατασκευαστούν ή να αναλυθούν δυναμικά, δηλαδή κατά την διάρκεια εκτέλεσης ενός λογικού προγράμματος, αντί να δίνονται μόνο στατικά ως είσοδος κατά την κλήση ή κατά την διάρκεια συγγραφής των λογικών προτάσεων του προγράμματος.

Με την δυνατότητα διάσπασης, οι σύνθετοι όροι μπορούν να μετατραπούν σε λίστες και στη συνέχεια μπορούμε να τους χειριστούμε με αναδρομικές διαδικασίες χειρισμού λιστών. Για παράδειγμα, μπορούμε να αναζητήσουμε αν ο ακόλουθος όρος έχει μέσα του ένα σύμβολο (π.χ. nick), χωρίς να ξέρουμε ακριβώς σε ποια θέση:

```
student(name(nick),aem(1453),year(3))
```

Με την δυνατότητα σύνθεσης, μπορούμε να "κατασκευάσουμε" μέσω προγραμματισμού έναν σύνθετο όρο για διάφορες χρήσεις. Για παράδειγμα, ο ακόλουθος σύνθετος όρος είναι συντακτικά όμοιος με μία κλήση του κατηγορήματος member/2 και όπως θα δούμε σε επόμενη ενότητα μπορεί μέσω της μεταβλητής κλήσης να κληθεί δυναμικά:

```
member(c,[a,b,c,d])
```

Ένα πιο πολύπλοκο παράδειγμα της αναγκαιότητας της διάσπασης και σύνθεσης σύνθετων όρων είναι το ακόλουθο. Στο Κεφάλαιο 6 δόθηκε το παράδειγμα ενός κατηγορήματος derivative/3 για την συμβολική παραγώγιση μαθηματικών συναρτήσεων. Στο συγκεκριμένο κατηγόρημα, κάθε κανόνας παραγώγισης από τα μαθηματικά αντιστοιχίζεται σε έναν λογικό κανόνα στην γλώσσα Prolog. Υπάρχουν όμως και σύνθετοι κανόνες παραγώγισης οι οποίοι δεν είναι εύκολο να αντιστοιχιστούν σε κανόνες Prolog. Για παράδειγμα, ο κανόνας της παραγώγισης σύνθετων συναρτήσεων, δηλαδή συναρτήσεων της μορφής $f(g(x))$, ο οποίος επιβάλει να παραγωγιστεί η εξωτερική συνάρτηση f θεωρώντας την εσωτερική ποσότητα $g(x)$ ως μία μεταβλητή, και στη συνέχεια να πολλαπλασιαστεί με την παράγωγο της εσωτερικής ποσότητας ως προς x:

$$df(g(x))/dx = df(g(x))/dg(x) * dg(x)/dx$$

Η μετατροπή του παραπάνω κανόνα στην Prolog θα μπορούσε να γίνει άμεσα ως εξής, αν επιτρεπόταν στην Prolog η χρήση μεταβλητών στην θέση συναρτησιακού συμβόλου:

```
derivative(F(G),X,DF*DG) :-
  derivative(F(G),G,DF),
  derivative(G,X,DG).
```

Επειδή όμως η Prolog αποτελεί υποσύνολο της κατηγορηματικής λογικής πρώτης τάξης, και κατά συνέπεια οι μεταβλητές επιτρέπονται μόνο ως ορίσματα όρων ή ατομικών τύπων και όχι ως συναρτησιακά σύμβολα ή κατηγορήματα (δηλαδή μόνο εντός και όχι εκτός παρενθέσεων), ο παραπάνω ορισμός του κανόνα παραγώγισης σύνθετων συναρτήσεων δεν είναι εφικτός. Όπως θα δούμε στην συνέχεια, η ύπαρξη των κατηγορημάτων σύνθεσης/διάσπασης όρων καθιστά εφικτό τον ορισμό, παρακάμπτοντας τον περιορισμό των μεταβλητών πρώτης τάξης στην Prolog.

Κατηγόρημα univ

Το ενσωματωμένο κατηγόρημα =../2, το οποίο ονομάζεται *univ* (προφέρεται *γιουνίβ*), χρησιμοποιείται και για τη σύνθεση και για τη διάσπαση σύνθετων όρων. Η μορφή του κατηγορήματος είναι:

```
Term =.. [Functor | Arguments]
```

Το κατηγόρημα univ πετυχαίνει όταν:

- το πρώτο όρισμά του (Term) είναι ένας σύνθετος όρος,
- το δεύτερο όρισμά του είναι μια λίστα, και
- το πρώτο στοιχείο της λίστας είναι το συναρτησιακό σύμβολο (Functot) του σύνθετου όρου (Term), ενώ τα υπόλοιπα στοιχεία της λίστας (Arguments) ενοποιούνται ένα-προς-ένα με τα αντίστοιχα ορίσματα του σύνθετου όρου.

Για παράδειγμα, η ακόλουθη κλήση κάνει αποσύνθεση ενός σύνθετου όρου σε λίστα:

```
?- parent(john,mary) =.. List.
List = [parent,john,mary]
```

Αντίθετα, η ακόλουθη κλήση κάνει σύνθεση ενός όρου από τα στοιχεία μιας λίστας:

```
?- P =.. [point,3,2].
P = point(3,2)
```

Άρα το κατηγόρημα univ μπορεί να χρησιμοποιηθεί με δύο τρόπους, ανάλογα με το ποιο όρισμά του είναι μεταβλητή. Υπάρχει όμως περίπτωση κανένα από τα δύο ορίσματα να μην είναι μεταβλητή. Στην περίπτωση αυτή, το κατηγόρημα univ ελέγχει αν τα δύο ορίσματά του μπορούν να ενοποιηθούν και πραγματοποιεί την ενοποίηση:

```
?- point(X,Y) =.. [point,3,2].
```

```
X=3, Y=2
?- parent(john,mary) =.. [parent,nick,mary].
false
```

Με την χρήση του κατηγορήματος univ είναι δυνατός ο ορισμός του κανόνα παραγώγισης σύνθετων συναρτήσεων ως εξής:

```
derivative(F_G_X,X,DF*DG) :-
  F_G_X =.. [_F,G_X],
  derivative(F_G_X,G_X,DF),
  derivative(G_X,X,DG).
```

Συγκεκριμένα, η σύνθετη συνάρτηση F_G_X αντιμετωπίζεται ως σύνθετος όρος, ο οποίος μπορεί να διασπαστεί με την βοήθεια του κατηγορήματος univ και να έχουμε πλέον πρόσβαση στο συναρτησιακό σύμβολο και στην εσωτερική ποσότητα G_X της συνάρτησης, χωρίς να παραβιάζεται η σύνταξη της κατηγορηματικής λογικής πρώτης τάξης. Συνεπώς, για να βρούμε για παράδειγμα την παράγωγο της συνάρτησης $ημ(χ+συν(χ))$ χρησιμοποιούμε την εξής κλήση:

```
?- derivative(sin(x+cos(x)),x,A).
A = cos(x+cos(x))* (1+ -sin(x))
```

Κατηγορήματα functor/3 και arg/3

Εναλλακτικά του κατηγορήματος univ, υπάρχει ένα ζεύγος ενσωματωμένων κατηγορημάτων functor/3 και arg/3, τα οποία συνεργατικά μπορούν να αναλύσουν και να κατασκευάσουν σύνθετους όρους, όπως το univ. Συγκεκριμένα, το κατηγόρημα functor/3 μπορεί να χρησιμοποιηθεί για την ανάλυση ενός σύνθετου όρου. Η σύνταξή του είναι:

```
functor(Term, Functor, No_of_arguments)
```

όπου Term είναι κάποιος σύνθετος όρος, Functor είναι το συναρτησιακό σύμβολο του όρου, και No_of_arguments είναι ο αριθμός των ορισμάτων του. Μία από τις πιο συνηθισμένες χρήσεις είναι για να εξάγουμε το συναρτησιακό σύμβολο και την τάξη (arity) ενός σύνθετου όρου:

```
?- functor(student(name(john),id(324)),Fun,Args).
Fun = student
Args = 2
```

Μία δεύτερη χρήση του functor/3 είναι για τη δημιουργία όρων με συγκεκριμένο συναρτησιακό σύμβολο και αριθμό ορισμάτων, τα οποία ορίσματα είναι ελεύθερες μεταβλητές.

```
?- functor(Term,driver,2).
Term = driver(_G1454,_G1455)
```

Για την πρόσβαση στα ορίσματα του σύνθετου όρου χρησιμοποιείται το κατηγόρημα arg/3, το οποίο ενοποιεί το ν-οστό όρισμα ενός σύνθετου όρου. Η σύνταξή του είναι:

```
arg(N, Term, Argument)
```

όπου N είναι η (ακέραια) θέση του ορίσματος Argument που θέλουμε να ενοποιήσουμε από το σύνθετο όρο Term. Μία από τις χρήσεις του κατηγορήματος είναι η "επιστροφή" της τιμής του ν-οστού ορίσματος ενός σύνθετου όρου, όταν το όρισμα Argument είναι μεταβλητή:

```
?- arg(2,student(name(john),class(b)),X).
X = class(b)
```

Αν το όρισμα Argument δεν είναι μεταβλητή, τότε απλά ελέγχεται αν το ν-οστό όρισμα του όρου μπορεί να ενοποιηθεί με την τιμή του ορίσματος Argument:

```
?- arg(1,think(positive),positive).
true
```

Αυτός ο τελευταίος τρόπος χρήσης του κατηγορήματος arg/3 αποδεικνύεται ιδιαίτερα χρήσιμος όταν θέλουμε να "δώσουμε" τιμή σε κάποιο όρισμα ενός σύνθετου όρου χωρίς να προχωρήσουμε σε αποσύνθεση και επανασύνθεση του όρου:

```
?- arg(1,student(X,aem(1234)),name(john)).
X = name(john)
```

Σε συνδυασμό δε με το functor/3 μπορούμε να συνθέσουμε ένα σύνθετο όρο και με πολλαπλές κλήσεις του arg/3 να ενοποιήσουμε τα (μεταβλητά) ορίσματά του με συγκεκριμένες τιμές:

```
?- functor(Term,driver,2), arg(1,Term,nick), arg(2,Term,ner8495).
Term = driver(nick,ner8495).
```

Το παραπάνω αποτέλεσμα θα μπορούσε να είχε επιτευχθεί και με την ακόλουθη κλήση του univ:

```
?- Term =.. [driver,nick,ner8495].
Term = driver(nick,ner8495)
```

Από τα παραπάνω είναι φανερό πως η συνύπαρξη και των τριών κατηγορημάτων είναι πλεονασμός, καθώς είτε μόνο του το univ, είτε ο συνδυασμός των functor και arg, μπορούν να επιτύχουν τα ίδια αποτελέσματα. Στη συνέχεια δίνονται δύο ολοκληρωμένα παραδείγματα "κατασκευής" των κατηγορημάτων functor και arg από το κατηγόρημα univ και αντιστρόφως.

Κατασκευή functor/arg από univ

Για την κατασκευή του κατηγορήματος my_functor/3 (αντί του ενσωματωμένου functor/3) χρειάζεται το ενσωματωμένο κατηγόρημα length/2, το οποίο επιστρέφει τον αριθμό των στοιχείων μιας λίστας, και συγκεκριμένα της ουράς Arguments της λίστας List που προκύπτει από τη διάσπαση του όρου Term με την χρήση του univ. Η κεφαλή (Functor) της λίστας είναι αντίστοιχα το συναρτησιακό σύμβολο του σύνθετου όρου.

```
my_functor(Term,Functor,Arity) :-
  Term =.. List,
  List = [Functor|Arguments],
  length(Arguments,Arity).
```

Για την κατασκευή του κατηγορήματος my_arg/3 (αντί του ενσωματωμένου arg/3) χρειάζεται το κατηγόρημα nth_member/3, το οποίο επιστρέφει το ν-οστό στοιχείο μιας λίστας, εν προκειμένω της λίστας Arguments των ορισμάτων, και το οποίο παρουσιάστηκε στην προηγούμενη ενότητα.

```
my_arg(N,Term,Arg) :-
  Term =.. List,
  List = [Functor|Arguments],
  nth_member(N,Arg,Arguments).
```

Κατασκευή univ από functor/arg

Για την κατασκευή του κατηγορήματος univ/2 (αντί του ενσωματωμένου =../2) πρέπει να διακρίνουμε δύο περιπτώσεις, ανάλογα με τις δύο χρήσεις του univ: διάσπαση ή σύνθεση όρων. Για την διάσπαση όρων, όταν πρώτο όρισμα Term δεν είναι μεταβλητή, χρησιμοποιείται το κατηγόρημα functor/3 για να προσδιορίσει το συναρτησιακό σύμβολο Functor και τον αριθμό των ορισμάτων Arity. Στην συνέχεια ένα βοηθητικό κατηγόρημα univ_aux/4, το οποίο παρουσιάζεται παρακάτω, "κατασκευάζει" μία λίστα Arguments με τα ορίσματα του Term, μήκους Arity ξεκινώντας από το πρώτο. Τελικά, κατασκευάζεται η λίστα List με κεφαλή Functor και ουρά Arguments.

```
univ(Term,List) :-
  nonvar(Term),
  functor(Term,Functor,Arity),
  univ_aux(Term,1,Arity,Arguments),
  List = [Functor| Arguments].
```

Αντίστροφα, για τη σύνθεση όρων το πρώτο όρισμα Term είναι μεταβλητή. Η λίστα List αποσυντίθεται σε κεφαλή Functor και ουρά Arguments, το μήκος της οποίας είναι η τάξη Arity του όρου. Με το κατηγόρημα functor/3 "κατασκευάζεται" ο σύνθετος όρος και στη συνέχεια με τη χρήση του βοηθητικού κατηγορήματος univ_aux/4 κάθε όρισμα του νέου όρου ενοποιείται με το αντίστοιχο στοιχείο της λίστας Arguments.

```
univ(Term,List) :-
  var(Term),
  List = [Functor| Arguments],
  length(Arguments,Arity),
  functor(Term,Functor,Arity),
```

```
    univ_aux(Term,1,Arity,Arguments).
```

Το βοηθητικό κατηγόρημα univ_aux/4 διατρέχει την λίστα των ορισμάτων και ταυτόχρονα χρησιμοποιεί έναν μετρητή N (με αρχική τιμή 1 και τελική τιμή Arity) για να μετρήσει ποιο στοιχείο της λίστας είναι. Ο μετρητής χρησιμοποιείται από το κατηγόρημα arg/3 για να ενοποιήσει το ν-οστό στοιχείο του όρου Term με το ν-οστό στοιχείο της λίστας των ορισμάτων. Το κατηγόρημα είναι αναδρομικό και τερματίζει όταν ο μετρητής ξεπεράσει την τελική τιμή Arity οπότε και η λίστα των ορισμάτων πρέπει να έχει μείνει κενή.

```
univ_aux(_,N,Arity,[]) :-
  N > Arity.
univ_aux(Term,N,Arity,[Arg|Rest]) :-
  N =< Arity,
  arg(N,Term,Arg),
  N1 is N + 1,
  univ_aux(Term,N1,Arity,Rest).
```

Πλήρως ορισμένοι όροι

Στις προηγούμενες υπο-ενότητες είδαμε πώς μπορούν να συνδυαστούν διάφορες μετα-λογικές προεκτάσεις της Prolog, όπως είναι η εξέταση τύπου όρων και η σύνθεση διάσπαση όρων για την ανάπτυξη πολύπλοκων κατηγορημάτων. Ένα τέτοιο παράδειγμα είναι και αυτό που εξετάζεται στην συγκεκριμένη υπο-ενότητα. Ως γνωστόν, υπάρχουν κατηγορήματα που εξετάζουν αν ένας όρος είναι μεταβλητή ή όχι, απλός ή σύνθετος, κλπ. Υπάρχουν όμως περιπτώσεις που θέλουμε ακόμα και αν ένας όρος είναι σύνθετος, όπως ο f(g(X)), να διαπιστώσουμε αν εμφανίζονται μέσα στον όρο μεταβλητές ή μόνο σταθερές. Οι όροι στους οποίους δεν εμφανίζονται μέσα τους μεταβλητές ονομάζονται *πλήρως ορισμένοι όροι* (*ground terms*). Όταν ένας όρος είναι απλός, τότε με την χρήση των κατηγορημάτων var/nonvar μπορεί να διαπιστωθεί εύκολα αν είναι πλήρως ορισμένος ή όχι. Όταν όμως ο όρος είναι σύνθετος, θα πρέπει να διασπαστεί (π.χ. με την χρήση του univ), προκειμένου να διαπιστωθεί αν όλα του τα ορίσματα είναι πλήρως ορισμένα και μάλιστα χωρίς να είναι γνωστό εκ των προτέρων η μορφή του σύνθετου όρου.

Ο πρώτος κανόνας του κατηγορήματος ground_term/1 επιτυγχάνει αν ο όρος Term είναι ατομικός (atomic/1), το οποίο συμβαίνει μόνο στις περιπτώσεις των ατόμων και των αριθμών. Ο δεύτερος κανόνας εξετάζει αν ο όρος δεν είναι μεταβλητή (άρα είναι σύνθετος) και στην συνέχεια τον διασπά με το univ και εξετάζει με την βοήθεια του κατηγορήματος ground_term_aux/1 αν όλα τα ορίσματα της λίστας Arguments είναι πλήρως ορισμένοι όροι. Το κατηγόρημα ground_aux/1 εξετάζει αν κάθε στοιχείο της λίστας είναι ένας πλήρως ορισμένος όρος με την (έμμεσα) αναδρομική κλήση του ground_term/1. Η υλοποίηση του κατηγορήματος δίνεται παρακάτω:

```
ground_term(Term) :-
  atomic(Term).
ground_term(Term):-
  nonvar(Term),
  Term =.. [_Functor|Arguments],
  ground_aux(Arguments).

ground_aux([]).
ground_aux([First|Rest]):-
  ground_term(First),
  ground_aux(Rest).
```

10.5 Μεταβλητή Κλήση

Ένα σημαντικό χαρακτηριστικό της Prolog είναι η συντακτική ισοδυναμία προγραμμάτων και δεδομένων, καθώς οι κανόνες σύνταξης των ατομικών τύπων και των σύνθετων όρων είναι οι ίδιοι. Όπως είδαμε στα Κεφάλαια 3 και 4, οι ατομικοί τύποι και οι σύνθετοι όροι έχουν διαφορετικούς ρόλους μέσα σε ένα λογικό πρόγραμμα και συγκεκριμένα τον ρόλο του κώδικα και των δεδομένων, αντίστοιχα. Η συντακτική τους ομοιότητα όμως επιτρέπει την αντιμετώπιση του κώδικα ως σύνολο δεδομένων και το αντίστροφο. Αυτό πραγματοποιείται μέσω της *μεταβλητής κλήσης*, της δυνατότητας δηλαδή χρήσης μιας μεταβλητής στην θέση

μιας κλήσης, π.χ. στο σώμα ενός κανόνα. Η μεταβλητή αυτή θα πρέπει κατά τη στιγμή της εκτέλεσης να έχει πάρει τιμή και μάλιστα να έχει ενοποιηθεί με έναν σύνθετο όρο, όμοιο συντακτικά με μία κλήση - ατομικό τύπο. Εάν η μεταβλητή δεν έχει πάρει τιμή κατά την στιγμή της εκτέλεσης, η εκτέλεση του προγράμματος θα σταματήσει με μήνυμα λάθους.

Για παράδειγμα, η ακόλουθη σύνθετη κλήση αποτελείται από μία εντολή ενοποίησης μιας μεταβλητής με έναν σύνθετο όρο, ο οποίος είναι όμοιος συντακτικά με έναν ατομικό τύπο, και στην συνέχεια χρήση της μεταβλητής κλήσης με την τιμή αυτή:

```
?- X = member(c,[a,b,c]), X.
X = member(c,[a,b,c])
```

Το αποτέλεσμα της κλήσης είναι επιτυχές, αφού δεν επιστρέφεται false, και επιστρέφεται η τιμή της μεταβλητής X για την οποία υπήρξε επιτυχής κατάληξη της κλήσης. Σε πρώτη ανάγνωση δεν φαίνεται αναγκαία η χρήση της μεταβλητής κλήσης για μια τόσο απλή περίπτωση, καθώς θα μπορούσε να μπει στην ερώτηση απευθείας ο ατομικός τύπος member(c,[a,b,c]). Όμως το παράδειγμα αυτό απλά αναδεικνύει την δυνατότητα χρήσης της μεταβλητής κλήσης. Η "κατασκευή" της τιμής του σύνθετου όρου που θα αποτελέσει την τιμή της μεταβλητής κλήσης θα μπορούσε να είναι περισσότερο πολύπλοκη από μια απλή ενοποίηση, ακόμα και το αποτέλεσμα ενός πολύπλοκου τμήματος προγράμματος.

Για παράδειγμα, συνήθως η κατασκευή του σύνθετου όρου της μεταβλητής κλήσης πραγματοποιείται με την βοήθεια των κατηγορημάτων σύνθεσης όρων που παρουσιάστηκαν στην προηγούμενη ενότητα.

```
?- X =.. [member,c,[a,b,c]], X.
X = member(c,[a,b,c])
?- functor(X,member,2), arg(1,X,c), arg(2,X,[a,b,c]), X.
X = member(c,[a,b,c])
```

Σε κάποιες εκδόσεις της γλώσσας Prolog, στις οποίες συντακτικά απαγορεύεται να χρησιμοποιηθεί μεταβλητή στην θέση ατομικού τύπου, υπάρχει εναλλακτικά το ενσωματωμένο κατηγόρημα call/1, του οποίου η παράμετρος είναι η μεταβλητή κλήση:

```
?- X = member(c,[a,b,c]), call(X).
X = member(c,[a,b,c])
```

Φυσικά στις περισσότερες σύγχρονες εκδόσεις της Prolog, οι δύο εκδοχές της μεταβλητής κλήσης συνυπάρχουν.

Με τη χρήση της μεταβλητής κλήσης, σε συνδυασμό φυσικά και με άλλες μετα- και έξτρα-λογικές δυνατότητες της Prolog, είναι δυνατή η ανάπτυξη πολύ σημαντικών και χρήσιμων κατηγορημάτων που σχετίζονται με τον έλεγχο της εκτέλεσης του προγράμματος. Για παράδειγμα, η άρνηση - ως - αποτυχία της Prolog, που παρουσιάστηκε στο Κεφάλαιο 9, δεν είναι αναγκαίο να υπάρχει ως ενσωματωμένο κατηγόρημα not/1, καθώς μπορεί να κατασκευαστεί συνδυάζοντας την μεταβλητή κλήση, την αποκοπή και την αποτυχία, ως εξής:

```
not(Goal) :- Goal, !, fail.
not(_).
```

Η παράμετρος του not/1 είναι ένας ατομικός τύπος, που δίδεται ως όρισμα υπό τη μορφή σύνθετου όρου (δεδομένο). Στη συνέχεια, μέσω της μεταβλητής κλήσης μετατρέπεται σε κλήση, δηλαδή κώδικα. Αν επιτύχει η κλήση του Goal, τότε μέσω του συνδυασμού αποκοπής και αποτυχίας το κατηγόρημα not/1 αποτυγχάνει, δηλαδή επιστρέφει false, κάτι που είναι επιθυμητό στην περίπτωση της άρνησης. Η αποκοπή εξασφαλίζει ότι η αποτυχία (fail) δεν θα προκαλέσει στον μηχανισμό εκτέλεσης της Prolog ανεπιθύμητη οπισθοδρόμηση στην δεύτερη πρόταση του κατηγορήματος not/1, αλλά ούτε και σε τυχόν σημεία οπισθοδρόμησης που υπάρχουν μέσα στην κλήση Goal. Εάν δεν επιτύχει η κλήση Goal, τότε εκτελείται η δεύτερη πρόταση του κατηγορήματος not/1, η οποία απλώς επιτυγχάνει ανεξάρτητα από την τιμή της κλήσης.

Μία άλλη περίπτωση χρήσης της μεταβλητής κλήσης, σε συνδυασμό με την αποκοπή, είναι το κατηγόρημα *if-then-else*, με τη συνηθισμένη ερμηνεία των συμβατικών γλωσσών προγραμματισμού.

```
if_then_else(Condition,Then,_Else) :-
  Condition, !, Then.
if_then_else(_Condition,_Then,Else) :-
```

```
Else.
```

Και τα τρία ορίσματα του κατηγορήματος είναι ατομικοί τύποι που δίδονται ως σύνθετοι όροι και στη συνέχεια μέσω της μεταβλητής κλήσης μετατρέπονται σε κλήσεις. Στην πρώτη πρόταση, εκτελείται η κλήση Condition, η οποία είτε πετυχαίνει είτε αποτυγχάνει. Αν πετύχει, τότε συνεχίζεται η εκτέλεση του σώματος του πρώτου κανόνα, εκτελείται η αποκοπή, η οποία "κόβει" τις εναλλακτικές κλήσεις της Condition αλλά και την δεύτερη πρόταση του κατηγορήματος if_then_else/3 και εκτελεί την κλήση Then. Αν αποτύχει η Condition τότε εκτελείται η δεύτερη πρόταση του κατηγορήματος if_then_else/3 η οποία εκτελεί την κλήση Else. Στα παρακάτω παραδείγματα χρησιμοποιείται το κατηγόρημα if_then_else/3 για να συγκριθούν δύο αριθμοί και να βρεθεί ο μέγιστος από αυτούς.

```
?- X=5, Y=6, if_then_else(X>=Y, Max=X, Max=Y).
X = 5, Y = Max, Max = 6
?- X=15, Y=6, if_then_else(X>=Y, Max=X, Max=Y).
X = Max, Max = 15, Y = 6
```

Η εναλλαγή μεταξύ ατομικών τύπων και σύνθετων όρων καθώς και η μεταβλητή κλήση είναι πολύ σημαντικά στοιχεία για πολλές από τις προχωρημένες δυνατότητες της Prolog, όπως είναι η δυναμική διαχείριση λογικών προτάσεων και η διαχείριση συνόλου λύσεων, που θα παρουσιαστούν στις επόμενες ενότητες. Επίσης, με την μεταβλητή κλήση είναι δυνατή η ανάπτυξη εξειδικευμένων κελυφών (shells) της γλώσσας Prolog ή άλλων παραλλαγών της λογικής, δηλαδή διασυνδέσεων χρήστη (user interface) τα οποία διευκολύνουν την σύνταξη και την εκτέλεση προγραμμάτων. Επίσης, είναι δυνατή ανάπτυξη διερμηνέων (interpreters) άλλων γλωσσών προγραμματισμού στην γλώσσα Prolog, συνήθως παραλλαγών της λογικής. Το σύνολο αυτών των δυνατοτήτων συνήθως ονομάζεται *μετα-προγραμματισμός*, δηλαδή η δυνατότητα προγραμματισμού μιας άλλης γλώσσας προγραμματισμού.

10.6 Δυναμική διαχείριση λογικών προτάσεων

Ως τώρα τα λογικά προγράμματα τα αντιμετωπίσαμε ως ένα σύνολο στατικών λογικών προτάσεων (γεγονότα και κανόνες), τα οποία υπάρχουν μέσα σε ένα αρχείο και φορτώνονται στην μνήμη αρχικά, κατά την διαδικασία φόρτωσης προγράμματος με την εντολή consult/1. Η τελευταία συνήθως μεταγλωττίζει (compile) τις λογικές προτάσεις σε μία ενδιάμεση μορφή αναπαράστασης (WAM code) και στη συνέχεια κατά την διάρκεια της εκτέλεσης ο ενδιάμεσος αυτός κώδικας διερμηνεύεται (interpreted). Έτσι, η Prolog είναι εν μέρει μεταφραζόμενη γλώσσα (compiled) και εν μέρει διερμηνευόμενη (interpreted). Ο ενδιάμεσος κώδικας κατά τη μετάφραση βελτιστοποιείται ώστε η εκτέλεσή του να είναι όσο το δυνατόν αποδοτικότερη. Συνήθως τα κατηγορήματα που υπάρχουν μέσα σε ένα λογικό πρόγραμμα και μεταφράζονται με τον τρόπο αυτό αναφέρονται ως *στατικά*, καθώς δεν μπορούν να μεταβληθούν κατά την διάρκεια εκτέλεσης του προγράμματος, όπως γίνεται σχεδόν σε όλες τις γλώσσες προγραμματισμού που βασίζονται στην μετάφραση κώδικα (compilation).

Υπάρχει όμως η δυνατότητα στην Prolog κάποια κατηγορήματα να δηλωθούν ως *δυναμικά*. Μια τέτοια δήλωση επιτρέπει την μεταβολή των προτάσεων των κατηγορημάτων κατά την διάρκεια εκτέλεσης του προγράμματος, μεταλλάσσοντας ουσιαστικά τον κώδικα του ίδιου του προγράμματος. Σε κάποιες παλαιότερες εκδοχές της γλώσσας Prolog όλα τα κατηγορήματα αντιμετωπίζονταν ως δυναμικά, ενώ στις πιο σύγχρονες εκδόσεις για να είναι δυναμικό ένα κατηγόρημα πρέπει να δηλωθεί με μία κλήση dynamic/1, π.χ.:

```
?- dynamic father/2.
true
```

Η προσθήκη λογικών προτάσεων σε ένα δυναμικό κατηγόρημα γίνεται με την χρήση των ενσωματωμένων κατηγορημάτων asserta/1 και assertz/1:

- asserta(X): Προσθέτει στο λογικό πρόγραμμα την πρόταση X (γεγονός ή κανόνας) τοποθετώντας την πριν από τις ήδη υπάρχουσες προτάσεις του ίδιου κατηγορήματος.
- assertz(X): Προσθέτει την πρόταση X μετά από τις προτάσεις του ίδιου κατηγορήματος.

Όπως είναι προφανές, η τιμή του ορίσματος X είναι ένας σύνθετος όρος ο οποίος είναι συντακτικά όμοιος

είτε με ένα γεγονός (ατομική πρόταση), π.χ.

```
?- asserta(father(nick,anne)).
true
```

ή έναν κανόνα (σύνολο ατομικών προτάσεων που συνδέονται με τον τελεστή `:-`), π.χ.

```
?- assertz((father(X,Y) :- parent(X,Y), male(X))).
true
```

Η αφαίρεση λογικών προτάσεων γίνεται με την χρήση του ενσωματωμένου κατηγορήματος retract/1, όπου το όρισμα είναι και πάλι μια λογική πρόταση η οποία θέλουμε να αφαιρεθεί από το πρόγραμμα. Η λογική πρόταση μπορεί να είναι πλήρως ενοποιημένη, όταν π.χ. θέλουμε να αφαιρέσουμε ένα συγκεκριμένο γεγονός:

```
?- retract(father(nick,anne)).
true
```

ή μπορεί να περιέχει ελεύθερες μεταβλητές όταν είτε δεν γνωρίζουμε ακριβώς την πλήρως ενοποιημένη μορφή της πρότασης, ή μας ενδιαφέρει να σβήσουμε κάποιες ή και όλες τις εναλλακτικές προτάσεις που ενοποιούνται με την μερικώς ενοποιημένη πρόταση του ορίσματος του retract/1:

```
?- retract(father(nick,X)).
X=anne;
X=dimitra
```

Στο προηγούμενο παράδειγμα βλέπουμε ότι το κατηγόρημα retract/1 έχει εναλλακτικές λύσεις όταν η προς διαγραφή πρόταση περιέχει ελεύθερες μεταβλητές και μέσω οπισθοδρόμησης μπορούν να διαγραφούν όλες ή μερικές από τις προτάσεις που ενοποιούνται με το όρισμα.

Για να διαγραφεί κανόνας, χρειάζεται η μορφή του κανόνα να προδιαγραφεί στο όρισμα του κατηγορήματος retract/1:

```
?- retract((father(X,Y):-B)).
B = (parent(X,Y),male(X))
```

Προφανώς υπάρχει περίπτωση το κατηγόρημα retract/1 να αποτύχει όταν δεν υπάρχει πρόταση στο πρόγραμμα που να μπορεί να ενοποιηθεί με την πρόταση-όρισμα:

```
?- retract((father(X,Y):-B,C,D)).
false
```

Η χρήση των assert/1 και retract/1 μεταβάλλει δυναμικά τη σημασία/ερμηνεία των λογικών προγραμμάτων, καθώς οι ίδιες κλήσεις έχουν διαφορετικές απαντήσεις πριν και μετά την εκτέλεση των συγκεκριμένων κατηγορημάτων. Για παράδειγμα, έστω ότι στο λογικό πρόγραμμα υπάρχει μόνο το γεγονός:

```
father(nick,mary).
```

και κάνουμε την ερώτηση:

```
?- father(nick,X).
X=mary
```

Αν προστεθεί ένα γεγονός "πάνω" από τα υπόλοιπα:

```
?- asserta(father(nick,john)).
true
```

η ίδια ερώτηση θα έχει διαφορετική απάντηση (σύνολο όλων των απαντήσεων):

```
?- father(nick,X).
X = john;
X = mary
```

Αν προστεθεί ένα γεγονός "κάτω" από τα υπόλοιπα, η ίδια ερώτηση θα έχει και πάλι διαφορετικές απαντήσεις:

```
?- assertz(father(nick,anna)).
true
?- father(nick,X).
X=john;
X=mary;
X=anna
```

Τέλος, αν αφαιρεθεί ένα γεγονός και πάλι θα υπάρξουν διαφορετικές απαντήσεις:

```
?- retract(father(Y,mary)).
Y=nick
?- father(nick,X).
X=john;
X=anna
```

Η εκτεταμένη χρήση των κατηγορημάτων assert/retract μπορεί να οδηγήσει σε προγράμματα τα οποία είναι πολύ δύσκολο να αποσφαλματωθούν, καθώς η ερμηνεία τους αλλάζει δυναμικά κατά την διάρκεια εκτέλεσης, όπως αναλύθηκε παραπάνω. Για το λόγο αυτό θα πρέπει η χρήση τους να περιορίζεται μόνο στις απόλυτα αναγκαίες περιπτώσεις.

Απομνημόνευση λύσεων

Μία σχετικά ασφαλής και σημασιολογικά αποδεκτή χρήση του κατηγορήματος assert είναι η προσθήκη στο πρόγραμμα προτάσεων που ήδη αποτελούν λογικά του συμπεράσματα. Η σκοπιμότητα πίσω από μία τέτοια προσθήκη είναι η εξής: αν για την απόδειξη ενός συμπεράσματος (κλήση) από τις αρχικές προτάσεις του προγράμματος απαιτείται μεγάλος υπολογιστικός φόρτος, τότε η προσθήκη του συμπεράσματος στο λογικό πρόγραμμα αφενός δεν θα αλλάξει τη σημασιολογία του προγράμματος (αφού έτσι κι αλλιώς ήταν συμπέρασμα και πριν την προσθήκη) και αφετέρου θα επιταχύνει την απάντηση σε επόμενη κλήση της συγκεκριμένης ερώτησης αφού δεν θα ξαναχρειαστεί να εκτελεστεί η χρονοβόρα διαδικασία της απόδειξης. Η συγκεκριμένη χρήση του assert ονομάζεται *απομνημόνευση* (*memoing*).

Ως παράδειγμα παραθέτουμε το κατηγόρημα factorial/2 (παραγοντικό) το οποίο παρουσιάστηκε στο Κεφάλαιο 6. Ως γνωστόν, για να υπολογιστεί το παραγοντικό ενός αριθμού N θα πρέπει πρώτα να υπολογιστεί το παραγοντικό του N-1, κοκ.

```
factorial(0,1).
factorial(N,F):-
  N>0,
  N1 is N-1,
  factorial(N1,F),
  F is F*N.
```

Για τον υπολογισμό π.χ. του 5! θα γίνουν 5 αναδρομικές κλήσεις, ενώ για τον υπολογισμό του 6! θα γίνουν 6 αναδρομικές κλήσεις:

```
?- factorial(5,A).
A = 120
?- factorial(6,A).
A = 720
```

Όμως αφού το 6! χρειάζεται το 5! για να υπολογιστεί και το έχουμε ήδη υπολογίσει από την προηγούμενη κλήση, μια απομνημόνευση των ήδη υπολογισμένων παραγοντικών θα περιόριζε τις αναδρομικές κλήσεις για το 6! μόνο σε μία, αν φυσικά έχει προηγηθεί πιο πριν η κλήση του 5!:

```
factorial(0,1).
factorial(N,F) :-
  N1 is N - 1,
  factorial(N1,F1),
  F is N * F1,
  asserta(factorial(N,F)).
```

Μετά την κλήση

```
?- factorial(5,A).
A = 120
```

κάθε ένα παραγοντικό των αριθμών 1 έως και 5 θα έχει προστεθεί ως γεγονός στο λογικό πρόγραμμα πριν από την τερματική συνθήκη factorial(0,1) ως εξής:

```
factorial(5, 120).
factorial(4, 24).
factorial(3, 6).
factorial(2, 2).
```

```
factorial(1, 1).
factorial(0, 1).
...
```

Αυτό σημαίνει πως για τον υπολογισμό του 6! θα χρειαστεί μία μόνο αναδρομική κλήση, αφού όταν ζητηθεί η απόδειξη factorial(5,F1) η απάντηση μπορεί να δοθεί άμεσα από το πρώτο γεγονός factorial(5,120) στο πρόγραμμα. Το μοναδικό πρόβλημα με την παραπάνω λύση είναι ότι η ενοποίηση με το πρώτο γεγονός (στην ουσία με οποιοδήποτε γεγονός, πλην του factorial(0,1)) αφήνει ως εναλλακτική λύση πάντα τον αναδρομικό κανόνα, με αποτέλεσμα αν ζητηθεί εναλλακτική λύση να εκτελεστεί ξανά ο κανόνας. Αυτό θα έχει ως αποτέλεσμα να ξαναπροστεθεί το ίδιο γεγονός πολλές φορές στη μνήμη. Για να αποφύγουμε έναν τέτοιο βομβαρδισμό της μνήμης με πλεονάζουσα πληροφορία, καλό είναι αντί απλού γεγονότος να προστίθεται ένας κανόνας με μοναδική κλήση στο σώμα του την αποκοπή. Με τον τρόπο αυτό, αν μία ερώτηση για παραγοντικό απαντηθεί από κάποιο απομνημονευμένο γεγονός, ο αναδρομικός κανόνας δεν αποτελεί πλέον εναλλακτική λύση, λόγω της εκτέλεσης της αποκοπής. Η υλοποίηση φαίνεται παρακάτω:

```
factorial(N,F) :-
  N1 is N - 1,
  factorial(N1,F1),
  F is N * F1,
  asserta((factorial(N,F) :- !)).
```

Εξέταση περιεχομένων λογικού προγράμματος

Σημειώνεται ότι τα περιεχόμενα ενός λογικού προγράμματος που βρίσκεται φορτωμένο στη μνήμη μπορούμε να τα δούμε με τη χρήση του ενσωματωμένου κατηγορήματος listing/0, για όλα τα κατηγορήματα, ή του listing/1, για να δούμε όλες τις προτάσεις ενός μόνο κατηγορήματος:

```
?- listing(father).
:- dynamic father/2.
father(nick, anne).
father(A, B) :-
        parent(A, B),
        male(A).
true
```

Τέλος, υπάρχει και το ενσωματωμένο κατηγόρημα clause/2 με το οποίο μπορούμε να ανακαλέσουμε μία-μία τις προτάσεις του λογικού προγράμματος που μπορούν να ενοποιηθούν με τα δύο ορίσματα του κατηγορήματος. Το πρώτο όρισμα είναι η κεφαλή της πρότασης, ενώ το δεύτερο το σώμα της πρότασης. Μερικά χαρακτηριστικά παραδείγματα χρήσης του κατηγορήματος είναι τα ακόλουθα:

```
?- clause(father(A,B),Body).
A = nick,
B = anne,
Body = true ;
Body = (parent(A,B),male(A)).
?- clause(father(A,B),(Goal1,Goal2)).
Goal1 = parent(A,B),
Goal2 = male(A).
?- clause(father(A,anne),Body).
A = nick,
Body = true ;
Body = (parent(A,anne),male(A)).
?- clause(father(A,anne),true).
A = nick ;
false.
```

Είναι φανερό πως όταν θέλουμε να αναζητήσουμε αποκλειστικά γεγονότα πρέπει το δεύτερο όρισμα να έχει τιμή true, ενώ σε περίπτωση που κάποιες από τις μεταβλητές του πρώτου ορίσματος (κεφαλή της πρότασης) είναι ενοποιημένες, τότε η ενοποίηση αυτή προωθείται και στο σώμα της πρότασης. Το κατηγόρημα clause/2 θα χρησιμοποιηθεί στην υλοποίηση ενός μετα-διερμηνέα (meta-interpreter) στο τέλος αυτού του Κεφαλαίου.

Χρήση assert/retract για προσομοίωση καθολικών μεταβλητών

Ως γνωστόν, στην Prolog οι μεταβλητές εκτός του ότι δεν συμπεριφέρονται ως απλές θέσεις μνήμης όπως στις συμβατικές γλώσσες προγραμματισμού, έχουν αυστηρά τοπική εμβέλεια στον κανόνα που εμφανίζονται. Με την χρήση των κατηγορημάτων assert και retract μπορεί κάποιος να προσομοιώσει τις *καθολικές μεταβλητές* (*global variables*) που υπάρχουν σε άλλες γλώσσες προγραμματισμού, δηλαδή μεταβλητές οι τιμές των οποίων αποθηκεύονται ως γεγονότα με την βοήθεια του assert από έναν κανόνα και είναι προσβάσιμες από οποιονδήποτε άλλον κανόνα ενός λογικού προγράμματος. Επιπλέον, για αυτές τις μεταβλητές ισχύει η *καταστροφική ανάθεση τιμής* (*destructive assignment*).

Για παράδειγμα, έστω τα ακόλουθα γεγονότα τα οποία αναπαριστούν την γειτνίαση κρατών:

```
next(greece,turkey).
next(greece,albania).
next(greece,bulgaria).
next(greece,fyrom).
next(bulgaria,romania).
...
```

Με την χρήση των assert και retract θα κατασκευαστεί ένα κατηγόρημα το οποίο θα υπολογίζει με πόσες χώρες συνορεύει μια συγκεκριμένη χώρα. Παρακάτω παρατίθεται ο κώδικας, ο οποίος στη συνέχεια αναλύεται.

```
neighbours(_Country,_T) :-
  assert(counter(0)),
  fail.
neighbours(Country,T) :-
  next(Country,_X),
  retract(counter(T)),
  T1 is T + 1,
  assert(counter(T1)),
  fail.
neighbours(_Country,T) :-
  retract(counter(T)).
```

Ο παραπάνω κώδικας είναι χαρακτηριστικό παράδειγμα αυτού που αποκαλούμε “βρώμικος κώδικας”, ειδικά όσον αφορά στη γλώσσα Prolog. Οι τρεις κανόνες του κατηγορήματος neighbours/2 στην πραγματικότητα δεν αποτελούν εναλλακτικές λογικές προτάσεις απόδειξης κάποιου θεωρήματος αλλά στην πραγματικότητα είναι τα τρία τμήματα ενός διαδικαστικού, σειριακά εκτελέσιμου κώδικα. Αυτό γίνεται εφικτό χάρη στην πλήρη εκμετάλλευση των δυνατοτήτων ελέγχου της εκτέλεσης της αποδεικτικής διαδικασίας της Prolog και φυσικά στην χρήση των assert/1 και retract/1.

Συγκεκριμένα, ο πρώτος κανόνας εισάγει στο πρόγραμμα το γεγονός counter(0), δηλαδή αρχικοποιεί την καθολική τιμή counter με την τιμή 0. Στην συνέχεια το κατηγόρημα fail εξαναγκάζει την Prolog σε αποτυχία και ο έλεγχος μεταβαίνει στον δεύτερο κανόνα του κατηγορήματος neighbours/2. Καθώς δε το κατηγόρημα assert είναι μη-αναστρέψιμο (non-backtrackable) ο επόμενος κανόνας θα μπορεί να “δει” την τιμή της καθολικής μεταβλητής counter.

Στον δεύτερο κανόνα υπάρχει ένα βρόχος επανάληψης μέσω οπισθοδρόμησης (backtracking loops ή failure-driven loops), ο οποίος σαν πρώτη κλήση στο σώμα του έχει μία κλήση η οποία μπορεί να παράγει εναλλακτικές λύσεις και συγκεκριμένα βρίσκει μία χώρα με την οποία συνορεύει η χώρα Country. Στη συνέχεια, μέσω του retract/1, ανακαλείται και διαγράφεται η τρέχουσα τιμή της μεταβλητής counter (αρχικά 0), αυξάνεται κατά 1 και η νέα τιμή αποθηκεύεται ξανά ως γεγονός counter/1. Τέλος, μέσω του κατηγορήματος fail, ο μηχανισμός εκτέλεσης της Prolog εξαναγκάζεται σε αποτυχία και οπισθοδρόμηση στην κλήση next/2 η οποία έχει την δυνατότητα παραγωγής εναλλακτικών λύσεων, καθώς η ίδια χώρα Country μπορεί να συνορεύει με πολλές άλλες χώρες. Σημειώνεται ότι μεταξύ του fail και του next/2, όλες οι κλήσεις δεν αποτελούν σημεία οπισθοδρόμησης, ιδιαίτερα δε η κλήση του κατηγορήματος assert, συνεπώς η τρέχουσα τιμή της καθολικής μεταβλητής counter διατηρείται αναλλοίωτη και δεν χάνεται λόγω της αποτυχίας και της οπισθοδρόμησης, όπως θα γινόταν με μια κανονική μεταβλητή της Prolog. Έτσι κάθε φορά που η κλήση της next/2 δίνει μια νέα λύση για χώρα που συνορεύει με την Country, ο βρόχος αυξάνει την

τιμή της καθολικής μεταβλητής counter κατά 1. Όταν εξαντληθούν οι εναλλακτικές λύσεις της next/2, αποτυγχάνει ο δεύτερος κανόνας και ο έλεγχος μεταφέρεται στον τρίτο κανόνα.

Στον τρίτο κανόνα η κλήση του retract ανακαλεί την τελευταία τιμή της καθολικής μεταβλητής counter και ταυτόχρονα διαγράφει εντελώς το γεγονός counter/1 έτσι ώστε το κατηγόρημα neighbours/2 να μην αφήσει "σκουπίδια" στην μνήμη. Επιπλέον, έτσι εξασφαλίζεται ότι και την επόμενη φορά που θα τρέξει το κατηγόρημα, θα υπάρχει μόνο ένα γεγονός counter/1 στη μνήμη, κάτι που είναι σημαντικό για τη σωστή εκτέλεση του συγκεκριμένου αλγορίθμου. Αν υπάρχουν περισσότερα γεγονότα counter/1 υπάρχει περίπτωση να μην δοθεί σωστό αποτέλεσμα.

Ένα παράδειγμα κλήσης του κατηγορήματος neighbours/2 είναι το ακόλουθο:

```
?- neighbours(greece,N).
N = 4
```

10.7 Διαχείριση συνόλου λύσεων

Πολλές φορές είναι ιδιαίτερα χρήσιμο να μπορούμε επεξεργαστούμε όλες τις εναλλακτικές λύσεις σε μια κλήση. Μία τέτοια περίπτωση είναι το πρόβλημα μέτρησης του αριθμού των εναλλακτικών απαντήσεων σε ένα ερώτημα, όπως αυτό που παρουσιάστηκε στο τέλος της προηγούμενης ενότητας. Αν, για παράδειγμα, υπήρχε τρόπος να μαζέψουμε σε μία λίστα όλες τις εναλλακτικές απαντήσεις στο ερώτημα next(Country,X), τότε μετρώντας το μήκος της λίστας θα μπορούσαμε εύκολα να απαντήσουμε στο ερώτημα "*με πόσες χώρες συνορεύει η χώρα Country;*"

Η Prolog παρέχει τη δυνατότητα "συλλογής" σε λίστα όλων των εναλλακτικών λύσεων-απαντήσεων μιας κλήσης μέσω των τριών κατηγορημάτων bagof/3, setof/3 και findall/3, τα οποία έχουν όλα την ίδια σύνταξη αλλά διαφορετική ερμηνεία και λειτουργία. Η σύνταξη θα παρουσιαστεί με τη βοήθεια του πρώτου κατηγορήματος findall/3.

Κατηγόρημα findall/3

Το κατηγόρημα findall/3 συντάσσεται ως εξής:

```
findall(Term,Goal,List)
```

όπου Term είναι ένας όρος, συνήθως μια μεταβλητή, Goal μια κλήση σε κάποιο κατηγόρημα του προγράμματος, η οποία περιέχει την μεταβλητή Term ή έχει κοινές μεταβλητές με τον όρο Term, και List μια λίστα. Το findall/3 πετυχαίνει πάντα και ενοποιεί την λίστα List με όλες τις εναλλακτικές τιμές για το Term, για τις οποίες η κλήση Goal είναι αληθής. Για την ευκολότερη επεξήγηση του κατηγορήματος findall/3, θα θεωρήσουμε αρχικά ότι ο όρος Term είναι μια απλή μεταβλητή η οποία υπάρχει μέσα στην κλήση Goal. Στη περίπτωση αυτή, το ερώτημα που θέτει το findall/3 είναι "*ποιες είναι οι τιμές της μεταβλητής για τις οποίες η κλήση Goal είναι αληθής*". Για παράδειγμα έστω ότι στο λογικό πρόγραμμα υπάρχουν τα ακόλουθα τρία γεγονότα:

```
father(nick,anna).
father(nick,dimitra).
father(jim,nick).
```

Αν θελήσω να βρω όλα τα παιδιά του nick πρέπει να δώσω την ακόλουθη κλήση:

```
?- father(nick,X).
X = anna;
X = dimitra;
false
```

Όμως κάθε φορά επιστρέφεται μόνο ένα παιδί ως τιμή της μεταβλητής X.. Με το κατηγόρημα findall/3 δίνεται η δυνατότητα να συλλεχθούν όλα τα παιδιά του nick σε μία λίστα σε μια και μοναδική κλήση:

```
?- findall(X,father(nick,X),List).
List = [anna,dimitra]
```

Όπως παρατηρούμε, ο όρος Term στην περίπτωση αυτή, είναι η μεταβλητή X, η οποία επίσης υπάρχει μέσα στην κλήση Goal (father(nick,X)), και όλες οι εναλλακτικές τιμές που παίρνει η X, οι οποίες κάνουν το

father(nickX) αληθές, συλλέγονται μέσα στη λίστα List. Επισημαίνεται ότι η μεταβλητή X μετά το πέρας της εκτέλεσης του findall/3 παραμένει ελεύθερη μεταβλητή, δηλαδή δεν ενοποιείται με καμία από τις εναλλακτικές τιμές της.

Σε περίπτωση που η κλήση δεν ικανοποιείται, δηλαδή δεν υπάρχει καμία τιμή της μεταβλητής X που να καθιστά την κλήση αληθή, τότε το findall/3 επιστρέφει την κενή λίστα:

```
?- findall(X,father(john,X),List).
List = []
```

Γενικεύοντας, το πρώτο όρισμα της findall/3 μπορεί να περιέχει οποιονδήποτε όρο, ο οποίος έχει τουλάχιστον μία κοινή μεταβλητή με την κλήση:

```
?- findall(child(X),father(nick,X),L).
L = [child(anna),child(dimitra)]
```

Όπως βλέπουμε, το πρώτο όρισμα μπορεί και να θεωρηθεί ως το πρότυπο (pattern) των στοιχείων της λίστας του τρίτου ορίσματος. Επίσης, μέσα στην κλήση μπορεί να υπάρχουν περισσότερες από μία κοινές μεταβλητές με τον όρο, οι οποίες όλες "συλλέγονται" στη λίστα:

```
?- findall(child(X,Y),father(Y,X),L).
L = [child(anna,nick),child(dimitra,nick),child(nick,jim)]
```

Στο συγκεκριμένο παράδειγμα, η λίστα περιέχει ουσιαστικά όλα τα ζεύγη παιδί-πατέρας. Στην περίπτωση που στο πρώτο όρισμα Term υπάρχουν μεταβλητές που δεν είναι κοινές με την κλήση Goal τότε αυτές προφανώς παραμένουν ελεύθερες μεταβλητές και στην λίστα οι αντίστοιχοι όροι δεν θα είναι "πλήρως ορισμένοι" (grounded):

```
?- findall(child(X,Z),father(Y,X),L).
L =  [child(anna,_G754),child(dimitra,_G748),child(nick,_G742)]
```

Μία παρατήρηση στην παραπάνω περίπτωση είναι ότι σε κάθε απάντηση οι μεταβλητές Z είναι διαφορετικές μεταξύ τους. Αυτό φαίνεται από τον διαφορετικό εσωτερικό συμβολισμό για τις ελεύθερες μεταβλητές που δίνει η Prolog. Από την άλλη, αν μέσα στην κλήση υπάρχουν μεταβλητές, οι οποίες δεν είναι κοινές με τον όρο του πρώτου ορίσματος, τότε προφανώς οι τιμές τους δεν συλλέγονται στην λίστα του τρίτου ορίσματος:

```
?- findall(Y,father(Y,X),L).
L = [nick,nick,jim]
```

Η ερμηνεία του παραπάνω παραδείγματος είναι: "*βρες όλα τα Y τέτοια ώστε η κλήση father(Y,X) να αληθεύει, για κάποια τιμή του X*" ή πιο απλά "*βρες όλους τους πατεράδες*". Η ύπαρξη δηλαδή μιας μεταβλητής (ή περισσότερων) στην κλήση η οποία δεν είναι κοινή με το πρώτο όρισμα και δεν συλλέγεται υπονοεί τον υπαρξιακό ποσοδείκτη (*για κάποιο X)*.

Τέλος, η κλήση του δεύτερου ορίσματος μπορεί να είναι σύνθετη, δηλαδή να περιλαμβάνει πολλές απλές κλήσεις, συζευγμένες (με τελεστή κόμμα ",") ή διαζευγμένες (με τελεστή ";") ή συνδυασμό των παραπάνω. Ο σύνθετος στόχος πρέπει να περικλείεται σε ένα ζεύγος παρενθέσεων, αλλιώς η Prolog θα παρερμηνεύσει την ύπαρξη πολλών τελεστών "," μέσα στο findall/3 ως περισσότερα από τρία ορίσματα:

```
?- findall(X-Y,(father(Z,X),father(Z,Y),X\=Y),L).
L = [anna-dimitra,dimitra-anna]
```

Στο παραπάνω παράδειγμα ζητάμε όλα τα ζεύγη των αδελφών, δηλαδή των X και Y που έχουν κάποιο κοινό πατέρα Z, αλλά δεν έχουν την ίδια τιμή μεταξύ τους. Βλέπουμε ότι παρόλο που η μεταβλητή Z δεν "συλλέγεται", εντούτοις η ύπαρξή της είναι σημαντική καθώς πρέπει να έχει την ίδια τιμή στις δύο κλήσεις του father/2 μέσα στην σύνθετη κλήση, για κάθε διαφορετικό ζεύγος τιμών των X και Y. Ένα άλλο παράδειγμα σύνθετης κλήσης με τη χρήση της διάζευξης είναι το εξής:

```
?- findall(X,(father(nick,X);father(X,nick)),L).
L = [anna,dimitra,jim]
```

το οποίο βρίσκει τόσο τα παιδιά του nick, όσο και τον πατέρα του, δηλαδή όλους του συγγενείς πρώτου βαθμού (εκτός της μητέρας, στο συγκεκριμένο παράδειγμα).

Κατηγόρημα bagof/3

Το κατηγόρημα bagof/3 έχει ακριβώς την ίδια σύνταξη με το findall/3, πλην όμως έχουν θεωρητικές διαφορές στην σημασία - λειτουργία τους, οι οποίες γίνονται αντιληπτές στον χρήστη σε δύο μόνο περιπτώσεις χρήσης:

- Όταν δεν υπάρχει απάντηση στην κλήση Goal.
- Όταν υπάρχουν μεταβλητές στην κλήση Goal, οι οποίες δεν συλλέγονται στην λίστα, δηλαδή δεν είναι κοινές με τον όρο Term.

Σε όλες τις υπόλοιπες περιπτώσεις το bagof/3 συμπεριφέρεται όπως το findall/3. Για παράδειγμα, η ακόλουθη κλήση έχει ακριβώς το ίδιο αποτέλεσμα με αυτό της προηγούμενης υπο-ενότητας:

```
?- bagof(X,father(nick,X),List).
List = [anna,dimitra]
```

Όμως, στην περίπτωση που η κλήση δεν ικανοποιείται τότε το bagof/3 αποτυγχάνει, αντί να επιστρέψει κενή λίστα όπως το findall/3:

```
?- bagof(X,father(john,X),List).
false
```

Επίσης, το bagof/3 εμφανίζει διαφορετική συμπεριφορά από το findall/3 στην περίπτωση που μέσα στην κλήση μπορεί να υπάρχουν μεταβλητές, οι οποίες δεν "συλλέγονται" στην λίστα των απαντήσεων - λύσεων, δηλαδή δεν είναι κοινές με το πρώτο όρισμα:

```
?- bagof(X,father(Y,X),L).
Y = jim, L = [nick] ;
Y = nick, L = [anna,dimitra]
```

Στην περίπτωση αυτή βλέπουμε ότι το bagof/3 πρώτα δίνει μία συγκεκριμένη τιμή σε κάθε τέτοια μη-κοινή ελεύθερη μεταβλητή και στη συνέχεια για την τιμή αυτή επιστρέφει όλες τις τιμές των "συλλεγόμενων" (κοινών με το πρώτο όρισμα) μεταβλητών. Στο συγκεκριμένο παράδειγμα, η παραπάνω κλήση του bagof/3 ερμηνεύεται ως "*βρες όλα τα παιδιά X, κάποιου συγκεκριμένου πατέρα Y*" και όχι ως "*βρες όλα τα παιδιά X, κάποιου οποιουδήποτε πατέρα Y*", όπως είναι στην περίπτωση του findall/3. Με άλλα λόγια, η ύπαρξη μιας μεταβλητής (ή περισσότερων) στην κλήση η οποία δεν συλλέγεται δεν υπονοεί τον υπαρξιακό ποσοδείκτη ∃ (*για κάποιο Y*), αλλά απαιτεί την δέσμευση της ελεύθερης μεταβλητής με κάποια συγκεκριμένη τιμή πρώτα.

Για να συμπεριφερθεί το bagof/3 όπως το findall/3 πρέπει οι μεταβλητές που δεν "συλλέγονται" στην λύση να δηλώνονται ως υπαρξιακά ποσοτικοποιημένες μεταβλητές με την βοήθεια του τελεστή "^":

```
?- bagof(X,Y^father(Y,X),L).
L = [anna,dimitra,nick]
```

Σημειωτέον ότι όταν η κλήση είναι σύνθετη τότε ο τελεστής "^" πρέπει να μπαίνει πριν από την παρένθεση για να έχει εμβέλεια σε ολόκληρη την σύνθετη κλήση. Για παράδειγμα, για να βρούμε όλα τα ζεύγη αδελφιών, η κλήση bagof είναι:

```
?- bagof(X-Y,Z^(father(Z,X),father(Z,Y),X\=Y),L).
L = [anna-dimitra,dimitra-anna]
```

Χωρίς τον τελεστή "^" στο αποτέλεσμα θα συμπεριλαμβάνεται η τιμή της μεταβλητής Z:

```
?- bagof(X-Y,(father(Z,X),father(Z,Y),X\=Y),L).
Z = nick, L = [anna-dimitra,dimitra-anna].
```

Στην συγκεκριμένη περίπτωση δεν διαφέρει η τιμή της λίστας L από το προηγούμενο παράδειγμα, αλλά αν είχαμε περισσότερους πατεράδες και ζεύγη αδελφιών στο λογικό πρόγραμμα, οι απαντήσεις θα ήταν διαφορετικές.

Στην περίπτωση που υπάρχουν πολλές μη-συλλεγόμενες μεταβλητές, οι οποίες θέλουμε να είναι υπαρξιακά ποσοτικοποιημένες, τότε όλες οι μεταβλητές τοποθετούνται η μία μετά την άλλη με τον τελεστή "**^**" μετά από κάθε μία. Για παράδειγμα, αν θέλουμε να βρούμε όλους τους παππούδες στο λογικό πρόγραμμά μας, πρέπει να δώσουμε την ακόλουθη κλήση:

```
?- bagof(X,Z^Y^(father(X,Z),father(Z,Y)),L).
L = [jim,jim]
```

Στην συγκεκριμένη περίπτωση χρειαζόμαστε τρεις μεταβλητές X, Z και Y για τις τρεις "γενεές". Μας ενδιαφέρει η τιμή μόνο της X, δηλαδή της πρώτης γενεάς. Οι άλλες δύο δηλώνονται ως υπαρξιακά ποσοτικοποιημένες.

Κατηγόρημα setof/3

Το κατηγόρημα setof/3 έχει την ίδια σύνταξη με τα findall/3, bagof/3, και η σημασία του μοιάζει περισσότερο με το bagof/3 παρά με το findall/3. Οι διαφορές του με το bagof συνοψίζονται στα ακόλουθα σημεία:

- Το setof/3 απομακρύνει τα διπλά στοιχεία από τη λίστα, δηλαδή η λίστα αποτελεί πλέον σύνολο.
- Το setof/3 διατάσσει τα στοιχεία στην λίστα - σύνολο σε αύξουσα λεξικογραφική ή αριθμητική σειρά.

Για παράδειγμα, έστω ότι θέλουμε όλους τους πατεράδες. Με την χρήση του findall/3 είδαμε ότι η κλήση έχει ως εξής:

```
?- findall(Y,father(Y,X),L).
L = [nick,nick,jim]
```

Με το bagof/3 είναι απαραίτητη η υπαρξιακή ποσοτικοποίηση της μεταβλητής Y:

```
?- bagof(Y,X^father(Y,X),L).
L = [nick,nick,jim]
```

Αν όμως θέλουμε μόνο μία φορά τον κάθε πατέρα, άσχετα με πόσα παιδιά έχει, τότε ενδείκνυται η χρήση του setof/3:

```
?- setof(Y,X^father(Y,X),L).
L = [jim,nick]
```

Εκτός από την απομάκρυνση των διπλών στοιχείων, τα στοιχεία της λίστας διατάσσονται λεξικογραφικά σε αύξουσα σειρά.

Παράδειγμα μέτρησης λύσεων

Σε αυτήν την υπο-ενότητα θα δοθεί μία πολύ πιο "καθαρή", από άποψης λογικής, λύση στο πρόβλημα της μέτρησης των γειτονικών κρατών με μία χώρα, που παρουσιάστηκε στην προηγούμενη ενότητα με την βοήθεια των κατηγορημάτων assert/1 και retract/1. Ο κώδικας για το κατηγόρημα neighbours/2 είναι πολύ πιο σύντομος και κατανοητός στην περίπτωση της χρήσης του findall/3:

```
neighbours(Country,T) :-
  findall(N,next(Country,N),Neighbours),
  length(Neighbours,T).
```

Συγκεκριμένα, με την βοήθεια του findall/3 συλλέγονται όλες οι εναλλακτικές λύσεις της κλήσης next(Country,N), θεωρώντας ότι η μεταβλητή Country έχει ήδη πάρει τιμή, στην λίστα Neighbours, και στην συνέχεια απλά μετριέται το μήκος της λίστας με την βοήθεια του ενσωματωμένου κατηγορήματος length/2. Η κλήση του neighbours/2 με δεδομένη την χώρα, δίνει ακριβώς την ίδια απάντηση με το πρόγραμμα της προηγούμενης ενότητας:

```
?- neighbours(greece,N).
N = 4
```

Ας θεωρήσουμε την περίπτωση του να μην δοθεί συγκεκριμένη χώρα ως πρώτο όρισμα στην count/2:

```
?- neighbours(Country,N).
N = 7
```

Στην περίπτωση αυτή η μεταβλητή Country στην κλήση της findall είναι ελεύθερη, συνεπώς ερμηνεύεται ως υπαρξιακά ποσοτικοποιημένη: "*βρες όλες τις γειτονικές χώρες N, μιας κάποιας χώρας Country*". Την ίδια ακριβώς απάντηση θα παίρναμε και από το πρόγραμμα με το assert/1 και retract/1 της προηγούμενης ενότητας. Αν όμως αντικατασταθεί το findall/3 από το bagof/3 στον ορισμό της neighbours/2:

```
neighbours(Country,T) :-
  bagof(N,next(Country,N),Neighbours),
  length(Neighbours,T).
```

τότε η ίδια κλήση (με το πρώτο όρισμα της neighbours/2 να είναι ελεύθερη μεταβλητή) θα έχει πολύ διαφορετική απάντηση, καθώς η μεταβλητή Country μέσα στο bagof/3 είναι μια μεταβλητή που δεν συλλέγεται στο αποτέλεσμα, άρα πρέπει πρώτα να δεσμευθεί σε κάποια συγκεκριμένη τιμή ("*βρες όλες τις γειτονικές χώρες N, κάποιας συγκεκριμένης χώρας Country*"):

```
?- neighbours(Country,N).
Country = albania, N = 1 ;
Country = bulgaria, N = 2 ;
Country = greece, N = 4
```

Το ποιος ορισμός είναι πιο σωστός εξαρτάται από ποια είναι η επιθυμητή συμπεριφορά από το κατηγόρημα neighbours/2, δηλαδή δεν υπάρχει εξ΄ορισμού σωστό ή λάθος στην συγκεκριμένη περίπτωση.

10.8 Ολοκληρωμένο παράδειγμα διαχείρισης βάσης δεδομένων

Στην ενότητα αυτή θα παρουσιάσουμε ένα ολοκληρωμένο λογικό πρόγραμμα διαχείρισης βάσης δεδομένων, το οποίο ενοποιεί σχεδόν όλα τα μετα-λογικά ενσωματωμένα κατηγορήματα που παρουσιάστηκαν σε αυτό το κεφάλαιο, καθώς και αρκετά από τα έξτρα-λογικά κατηγορήματα του προηγούμενου κεφαλαίου. Συγκεκριμένα, πρόκειται για ένα πρόγραμμα διαχείρισης βάσης δεδομένων βαθμολογιών φοιτητών, το οποίο έχει τις δυνατότητες φόρτωσης και αποθήκευσης γεγονότων από / σε αρχείο, εισαγωγή, αναζήτηση και διαγραφή βαθμολογιών, καθώς και εμφάνιση κάποιων στατιστικών - συγκεντρωτικών στοιχείων από την βάση. Τέλος, όλες οι δυνατότητες του προγράμματος είναι διαθέσιμες μέσα από ένα αρχικό μενού επιλογών, το οποίο εμφανίζεται ξανά κάθε φορά που τελειώνει η εργασία κάθε επιλογής.

Στο αρχικό μενού εμφανίζονται όλες οι δυνατότητες του προγράμματος, καθώς και η δυνατότητα τερματισμού του προγράμματος:

- Φόρτωση βάσης δεδομένων βαθμολογίας από αρχείο στην κύρια μνήμη
- Αποθήκευση βάσης δεδομένων βαθμολογίας από την κύρια μνήμη σε αρχείο
- Εύρεση βαθμολογίας φοιτητή
- Εισαγωγή βαθμολογίας φοιτητή
- Διαγραφή βαθμολογίας φοιτητή
- Εμφάνιση αριθμού φοιτητών που "πέρασαν" το μάθημα
- Μέσος όρος βαθμολογίας φοιτητών στο μάθημα
- Εκτύπωση βαθμολογίου
- Έξοδος

Η επιλογή κάθε λειτουργίας θα γίνεται με την βοήθεια ενός αριθμού (0 για έξοδο). Αν ο χρήστης δεν εισάγει σωστό αριθμό, τότε θα τυπώνεται μήνυμα λάθους και θα επανεμφανίζεται το αρχικό μενού. Ακολουθεί ο κώδικας για την εμφάνιση του αρχικού μενού επιλογών:

```
run :-
  write('Main Menu:'), nl,
  write('----------'), nl,
  write('1 - Load a student table'), nl,
  write('2 - Save a student table'), nl,
  write('3 - Find a student grade'), nl,
  write('4 - Insert a student grade'), nl,
  write('5 - Delete a student grade'), nl,
  write('6 - Find how many student have passed'), nl,
  write('7 - Average Class Grade'), nl,
  write('8 - Print Student List with Marks'), nl,
  write('0 - Exit'), nl, nl,
  write('==> Choice: '),
  read(Choice),
```

```
  do_on_choice(Choice).
```

Μετά την πληκτρολόγηση από τον χρήστη της επιλογής του, το κατηγόρημα do_on_choice/1 αναλαμβάνει να εκτελέσει την αντίστοιχη επιλογή. Οι επιλογές είναι διατεταγμένες σε αύξουσα σειρά, αν και αυτό δεν είναι απαραίτητο. Αυτό που είναι εντελώς απαραίτητο είναι ο τελευταίος κανόνας του κατηγορήματος do_on_choice/1 να εκτελείται μόνο εφόσον αποτύχει η ενοποίηση του ορίσματος με κάποιον από τους προηγούμενους κανόνες, έτσι ώστε να τυπώνεται το μήνυμα λάθους:

```
do_on_choice(1) :-
  !,            % Η αποκοπή κόβει τις άλλες επιλογές
  ...           % Κώδικας που αντιστοιχεί στην επιλογή
  run.          % Εκτέλεση του αρχικού μενού (επιστροφή)

do_on_choice(2) :- !,          % Δεύτερη επιλογή
  ...
  run.
...                            % Υπόλοιπες επιλογές
do_on_choice(0) :- !,          % Έξοδος από το μενού
  nl, write('Goodbye!'), nl.   % και τερματισμός προγράμματος

do_on_choice(_) :-
        % Εκτύπωση μηνύματος αν εισαχθεί λάθος επιλογή
  nl, write('Give a number between 0 and 8!'), nl,
  run.
```

Βλέπουμε ότι η επιστροφή στο κύριο μενού μετά την εκτέλεση της κάθε επιλογής (ή μετά την λάθος επιλογή) γίνεται με την κλήση του κατηγορήματος run/0 ως τελευταία κλήση κάθε κανόνα, εκτός από την περίπτωση της επιλογής του τερματισμού του προγράμματος. Έτσι το πρόγραμμά μας είναι έμμεσα αναδρομικό. Αυτό έχει ως αποτέλεσμα σε κάθε εκτέλεση του αρχικού μενού να "καταναλώνεται" ένα τμήμα της μνήμης, καθώς δεν μπορεί η Prolog να υλοποιήσει την λεγόμενη βελτιστοποίηση *tail-recursion*, με συνέπεια το πρόγραμμα να μην μπορεί να εκτελεστεί άπειρες φορές. Πρακτικά όμως δεν δημιουργείται κάποιο πρόβλημα, καθώς σε προσομοίωση πολλών επαναλήψεων διαπιστώθηκε ότι σε έναν μέτριο σύγχρονο υπολογιστή το όριο των επαναλήψεων ξεπερνάει τις 1,000,000 επαναλήψεις πριν εξαντληθεί η μνήμη!

Η φόρτωση της βάσης δεδομένων βαθμολογίας από αρχείο στην κύρια μνήμη μπορεί να γίνει με πολλούς τρόπους. Εδώ θα περιοριστούμε στην περίπτωση που στο αρχείο υπάρχουν γεγονότα της μορφής student/2:

```
student(george, 10).
student(nick, 4).
...
```

Χωρίς περιορισμό της γενικότητας, θεωρούμε ότι σε κάθε αρχείο υπάρχει βαθμολογία ενός μόνο μαθήματος, και ότι τα μικρά ονόματα των φοιτητών είναι τα μοναδικά προσδιοριστικά της ταυτότητάς τους. Προφανώς μια τέτοια απλή βάση δεδομένων δεν ανταποκρίνεται στις πραγματικές απαιτήσεις μιας εφαρμογής, αλλά ο σκοπός του παραδείγματος είναι να αναδείξει τεχνικές και όχι να αναπτύξει μια εφαρμογή για τον πραγματικό κόσμο. Με κατάλληλες τροποποιήσεις όμως μπορούν να προστεθούν παραπάνω πληροφορίες σε κάθε γεγονός ή ακόμα και να σπάσει η βάση δεδομένων σε πολλά κατηγορήματα, όπως θα γινόταν σε μια σχεσιακή βάση δεδομένων. Ένα αρχείο που περιέχει γεγονότα θα μπορούσε να φορτωθεί στην κύρια μνήμη είτε με το κατηγόρημα consult:

```
?- consult('student.pl').
true
```

είτε με τη χρήση ενός βρόχου που διαβάζει ένα-ένα τα γεγονότα από το αρχείο ως όρους και τα αποθηκεύει στην μνήμη με την βοήθεια του assert/1. Εδώ θα ακολουθήσουμε την δεύτερη οδό, προκειμένου να επιδείξουμε την χρήση όλων αυτών των ενσωματωμένων κατηγορημάτων των δύο κεφαλαίων.

```
do_on_choice(1) :-
  !,   % Η αποκοπή κόβει τις άλλες επιλογές
  nl, write('File name: '),
       % Εισαγωγή ονόματος αρχείου από τον χρήστη
  read(File),
```

```
        % Άνοιγμα αρχείου για ανάγνωση
    see(File),
        % Επανάληψη μέσω οπισθοδρόμησης
    repeat,
        % Ανάγνωση ενός γεγονότος από αρχείο
    read(Fact),
        % Εισαγωγή γεγονότος στη βάση
    ins_fact(Fact),
        % Αποτυχία και επιστροφή στο repeat,
        % ή επιτυχία και συνέχεια
    Fact == end_of_file,
    !,   % Αποκοπή για το repeat
    nl, write('File loaded.'), nl,
        % Κλείσιμο αρχείου
    seen,
        % Εκτέλεση του μενού (επιστροφή)
    run.

ins_fact(end_of_file) :-
    !.            % Εξαίρεση για το τέλος αρχείου
ins_fact(Fact) :-
    assert(Fact). % Εισαγωγή γεγονότος στην μνήμη
```

Η επαναληπτική διαδικασία ανάγνωσης των όρων πραγματοποιείται με τη βοήθεια ενός βρόχου repeat-until, όπου ο έλεγχος επανάληψης ή εξόδου γίνεται στην βάση του βρόχου συγκρίνοντας τον όρο που αναγνώστηκε από το αρχείο με το άτομο end_of_file που επιστρέφεται όταν το αρχείο φτάσει στο τέλος του. Εφόσον δεν φτάσει, η αποτυχία "ωθεί" μέσω οπισθοδρόμησης τον βρόχο πίσω στο repeat/0. Όταν φτάσει στο τέλος, υπάρχει αποτυχία και έξοδος από τον βρόχο. Η αποκοπή εξασφαλίζει ότι δεν θα υπάρχει σημείο οπισθοδρόμησης αποτυχίας στο repeat αν τυχόν συμβεί αποτυχία αργότερα κάπου αλλού στο πρόγραμμα. Οι βρόχοι μέσω οπισθοδρόμησης εξασφαλίζουν μηδενική κατανάλωση μνήμης και έτσι προτιμώνται όταν ο αριθμός των επαναλήψεων μπορεί να είναι μεγάλος (π.χ. ανάγνωση μεγάλων αρχείων). Κάθε γεγονός που διαβάζεται, εισάγεται στην μνήμη με assert/1, αφού πρώτα ελεγχθεί αν έχει διαβαστεί το άτομο end_of_file το οποίο δεν θα πρέπει να εισαχθεί στην μνήμη. Για τον λόγο αυτό υπάρχει το κατηγόρημα ins_fact/1 με δύο προτάσεις.

Για την αποθήκευση της βάσης δεδομένων από την κύρια μνήμη σε αρχείο, ακολουθείται μια ανάλογη διαδικασία:

```
do_on_choice(2) :- !,
    write('File name: '), read(Name),
        % Άνοιγμα αρχείου για εγγραφή
    tell(Name),
        % Αποθήκευση γεγονότων
    save_facts,
        % Κλείσιμο αρχείου
    told,
    nl, write('File saved.'), nl,
    run.

save_facts :-
        % Αναζήτηση - Ανάκληση ενός γεγονότος
    student(Name,Grade),
        % Εγγραφή στο αρχείο
    write(student(Name,Grade)),
        % Εγγραφή "τελείας"
    write('.'), nl,
        % Αποτυχία και επιστροφή στο επόμενο γεγονός
    fail.
        % Όταν δεν υπάρχουν άλλα γεγονότα, απλά επιτυγχάνει
```

```
save_facts.
```

Εδώ, ο βρόχος επανάληψης και πάλι χρησιμοποιεί οπισθοδρόμηση, πλην όμως ο βρόχος είναι τύπου while, καθώς η είσοδος και η έξοδος στον βρόχο γίνεται από την κορυφή του. Συγκεκριμένα, το κατηγόρημα save_facts/0 έχει δύο κανόνες. Στον πρώτο κανόνα υπάρχει κλήση του γεγονότος student/2 με ελεύθερες μεταβλητές και στα δύο ορίσματα. Αυτό σημαίνει ότι δυνητικά όλη η βάση δεδομένων μπορεί να αποτελεί τις εναλλακτικές απαντήσεις στην κλήση. Το κατηγόρημα write/1 γράφει στο αρχείο έναν όρο ίδιο με το γεγονός που ενοποιήθηκε με την προηγούμενη κλήση. Στη συνέχεια ένα δεύτερο write γράφει στο αρχείο μια τελεία, έτσι ώστε να μπορεί να διαβαστεί σωστά από την πρώτη επιλογή το αρχείο. Θυμίζουμε ότι η εντολή read/1 απαιτεί μία τελεία που να υποδηλώνει το τέλος ενός όρου. Το fail στην βάση του βρόχου προκαλεί οπισθοδρόμηση στην κορυφή του βρόχου, στην επόμενη εναλλακτική λύση για το student/2. Όταν δεν υπάρχουν άλλα εναλλακτικά γεγονότα student/2 αποτυγχάνει ο πρώτος κανόνας του save_facts/0 και ο έλεγχος δίνεται στον δεύτερο, ο οποίος απλά πετυχαίνει χωρίς προϋποθέσεις, προκειμένου το υπόλοιπο πρόγραμμα να συνεχίσει να εκτελείται σειριακά.

Για την εύρεση βαθμολογίας φοιτητή, ζητείται η εισαγωγή του ονόματος του φοιτητή και στη συνέχεια το κατηγόρημα find_student/1 είτε βρίσκει το αντίστοιχο γεγονός student/2 και τυπώνει τον βαθμό είτε τυπώνει μήνυμα μη-εύρεσης:

```
do_on_choice(3) :- !,
  nl, write('Student Name: '),
  read(Name), nl,
      % Αναζήτηση βαθμού συγκεκριμένου φοιτητή
      % βάσει του ονόματος
  find_student(Name),

  nl, run.

find_student(Name) :-
  student(Name,Grade), !,
      % Αν ο φοιτητής υπάρχει, τότε τύπωσε το βαθμό του
  write('Grade: '),
  write(Grade), nl.

find_student(_) :-
      % Αν όχι, τύπωσε ότι δε βρέθηκε
  write('Not found!'), nl.
```

Για την εισαγωγή βαθμολογίας φοιτητή, ζητείται η εισαγωγή του ονόματος και της βαθμολογίας του φοιτητή και στη συνέχεια το κατηγόρημα insert_student/2 είτε απορρίπτει την εισαγωγή με μήνυμα λάθους, αν ο φοιτητής έχει ήδη βαθμολογία στην βάση, είτε εισάγει το αντίστοιχο γεγονός student/2 με την χρήση του assert.

```
do_on_choice(4) :- !,
  nl, write('Student Name: '),
     % Ανάγνωση από το χρήστη ονόματος
     % και βαθμού του φοιτητή
  read(Name), nl,
  write('Grade: '),
  read(Grade), nl,
      % Εισαγωγή φοιτητή στη μνήμη
  insert_student(Name,Grade),
  nl, run.

insert_student(Name,Grade) :-
      % Αν ο φοιτητής υπάρχει (με οποιονδήποτε βαθμό)
      % μην τον ξαναπροσθέτεις.
  student(Name,_), !,
  write('Student '), write(Name),
  write(' already exists!'), nl.
```

```
insert_student(Name,Grade) :-
      % Αν όχι, τότε εισαγωγή γεγονότος
  assert(student(Name,Grade)).
```

Για την διαγραφή βαθμολογίας φοιτητή, ζητείται η εισαγωγή του ονόματος και στη συνέχεια το κατηγόρημα delete_student/2 είτε διαγράφει το γεγονός student/2 με την βοήθεια του retract/1, ανεξαρτήτως βαθμού, είτε απορρίπτει την διαγραφή με μήνυμα λάθους, αν ο φοιτητής δεν υπάρχει στην βάση, άρα αν απέτυχε η retract/1.

```
do_on_choice(5) :- !,
  nl, write('Student Name: '),
      % Ανάγνωση του ονόματος του φοιτητή
  read(Name), nl,
      % Διαγραφή γεγονότος βάσει ονόματος
  delete_student(Name),
  nl, run.

delete_student(Name) :-
       % Αν ο φοιτητής υπάρχει, τότε διέγραψε
       % το γεγονός (βαθμός αδιάφορος)
  retract(student(Name,_)), !.

delete_student(_) :-
       % Αν όχι, τύπωσε ότι δε βρέθηκε
  write('Not found!'), nl.
```

Για την εμφάνιση του αριθμού των φοιτητών που “πέρασαν” το μάθημα, με τη βοήθεια του findall/3 μαζεύονται όλα τα ονόματα S των φοιτητών που έχουν βαθμό G μεγαλύτερο ή ίσο με πέντε στην λίστα L, στην συνέχεια με το length/2 μετριέται το μήκος της λίστας N και τυπώνεται. Η σύνθετη κλήση μέσα στο findall/3 πρέπει να περικλείεται υποχρεωτικά σε ζεύγος παρενθέσεων.

```
do_on_choice(6) :- !,
       % Μαζεύει όλους τους φοιτητές
       % που έχουν βαθμό από 5 και πάνω σε μια λίστα
  findall(S,(student(S,G), G >= 5),L),
  length(L,N),
       % Μετράει το μήκος της λίστας
  nl, write(N),
  write(' students have passed!'),
  nl, nl, run.
```

Για τον υπολογισμό και την εμφάνιση του μέσου όρου της βαθμολογίας των φοιτητών στο μάθημα, με τη βοήθεια της findall/3 συλλέγονται όλοι οι βαθμοί G των φοιτητών στην λίστα L. Τα ονόματα των φοιτητών είναι αδιάφορα, συνεπώς χρησιμοποιούμε την ανώνυμη μεταβλητή. Σημειωτέον ότι μόνο στην findall/3 μπορεί να χρησιμοποιηθεί η ανώνυμη μεταβλητή, ενώ στην bagof/3 θα πρέπει να υπάρχουν μόνο επώνυμες μεταβλητές υπαρξιακά ποσοτικοποιημένες, για να θεωρηθούν αδιάφορες οι τιμές τους.

```
do_on_choice(7) :- !,
       % Μαζεύει όλους τους βαθμούς των
       % φοιτητών σε μια λίστα
  findall(G,student(_,G),L),
       % Βρίσκει το μέσο όρο των αριθμών της λίστας
  average_list(L,Avg),
  nl, write('Course Average: '),
  write(Avg), nl, nl,
  run.
```

Στην συνέχεια το κατηγόρημα average_list/2 επιστρέφει τον μέσο όρο μιας λίστας αριθμών, ο οποίος και τυπώνεται στην οθόνη. Ο μέσος όρος υπολογίζεται αφού αθροίσει (sumlist/2) και μετρήσει τα στοιχεία (length/2) της λίστας και στην συνέχεια διαιρέσει τους δύο αριθμούς. Το κατηγόρημα length/2 είναι ενσωματωμένο στις περισσότερες εκδόσεις της Prolog, ενώ το sumlist/2 είναι ενσωματωμένο στην SWI-

Prolog. Αν δεν είναι ενσωματωμένο, τότε η κατασκευή του αφήνεται ως άσκηση (άσκηση 7.1). Στην περίπτωση που η λίστα είναι κενή, δηλαδή δεν υπάρχουν γεγονότα student/2 στην μνήμη, τότε ο μέσος όρος δεν ορίζεται, οπότε η average_list επιστρέφει ένα ανάλογο μήνυμα το οποίο τυπώνεται στην οθόνη, αντί του μέσου όρου. Αν δεν υπήρχε αυτός ο πρώτος κανόνας, τότε ο δεύτερος κανόνας θα προκαλούσε διακοπή της εκτέλεσης του προγράμματος με μήνυμα λάθους, καθώς η κενή λίστα έχει μήκος 0 και συνεπώς η διαίρεση με τον 0 στον δεύτερο κανόνα θα ήταν αδύνατη.

```
        % Αν είναι κενή η λίστα δεν ορίζεται ο Μ.Ο.
average_list([],'No student in the DB!') :- !.

        % Αν η λίστα δεν είναι κενή
average_list(L,A) :-
        % Άθροισε τα αριθμητικά στοιχεία της λίστας
  sumlist(L,S),
        % Μέτρησε τα στοιχεία της λίστας
  length(L,N),
        % Βρες τον μέσο όρο με διαίρεση
  A is S / N.
```

Για την εκτύπωση του βαθμολογίου χρησιμοποιείται το κατηγόρημα setof/3 για να συλλέξει ταξινομημένα σε λίστα τους βαθμούς και τα ονόματα των φοιτητών, και στη συνέχεια να τα τυπώσει στην οθόνη. Όπως φαίνεται παρακάτω, το πρώτο όρισμα του setof/3 είναι της μορφής G-S, δηλαδή πρόκειται για έναν σύνθετο όρο (με τελεστή την παύλα), ο οποίος έχει ως πρώτο όρισμα τον βαθμό και ως δεύτερο όρισμα το όνομα. Αυτό έχει ως αποτέλεσμα η ταξινόμηση να γίνεται με κύριο κλειδί ταξινόμησης τον βαθμό (σε αύξουσα σειρά 0 ως 10) και δευτερεύον κλειδί ταξινόμησης το όνομα, δηλαδή όσοι έχουν τον ίδιο βαθμό ταξινομούνται αλφαριθμητικά σε αύξουσα σειρά. Αυτό οφείλεται στο γεγονός ότι και για τους σύνθετους όρους ορίζεται αλφαριθμητική σειρά, η οποία συγκρίνει πρώτα αριθμητικά βάσει της τάξης του ορίσματος, στη συνέχεια αλφαριθμητικά βάσει του συναρτησιακού συμβόλου, και τέλος συγκρίνει ένα-προς-ένα τα ορίσματα από αριστερά προς τα δεξιά. Έτσι, στο συγκεκριμένο παράδειγμα η κύρια ταξινόμηση γίνεται με βάση το βαθμό, κοκ. Αν θέλαμε η ταξινόμηση να γίνει με κύριο κλειδί το όνομα θα έπρεπε το πρώτο όρισμα να είχε την μορφή S-G.

```
do_on_choice(8) :- !,
        % Μαζεύει όλους τους φοιτητές
        % και τους βαθμούς σε μια λίστα
  setof(G-S,student(S,G),L),
        % Στην ταξινόμηση προτάσσεται
        % ο βαθμός (μπαίνει πρώτος)
  print_students(L),
        % Τυπώνει τη λίστα
  nl, run.
```

Για την εκτύπωση, απλά αναλύεται κάθε κεφαλή της λίστας, όντας σύνθετος όρος της μορφής *βαθμός-όνομα*, και τυπώνεται πρώτα το όνομα και μετά ο βαθμός, αλλάζει γραμμή και αναδρομικά τυπώνονται τα υπόλοιπα στοιχεία της λίστας.

```
print_students([]) :- nl.
        % Δεν υπάρχουν άλλοι φοιτητές,
        % τύπωσε κενή γραμμή και σταμάτα
print_students([G-S|T]) :-
        % Τύπωσε τον πρώτο φοιτητή (όνομα)
        % και μετά το βαθμό του
  write(S), write(' : '),
  write(G), nl,
        % και μετά αναδρομικά τους
        % υπόλοιπους φοιτητές
  print_students(T).
```

Επέκταση παραδείγματος διαχείρισης βάσης δεδομένων

Στο πρόγραμμα διαχείρισης βάσης δεδομένων βαθμολογιών φοιτητών που δόθηκε, μερικές από τις λειτουργίες του αρχικού μενού, όπως η εκτύπωση του βαθμολογίου, ο υπολογισμός μέσου όρου, και ο υπολογισμός του αριθμού των προαχθέντων φοιτητών, είναι λειτουργίες που εξαρτώνται από τη φύση και τη σημασία των δεδομένων ή της εφαρμογής (domain dependent). Από την άλλη, κάποιες άλλες λειτουργίες, όπως το φόρτωμα - αποθήκευση της βάσης και η εισαγωγή - διαγραφή γεγονότων, είναι γενικές λειτουργίες για την διαχείριση βάσεων δεδομένων, ανεξάρτητες από την εφαρμογή (domain independent), οι οποίες μπορούν να παραμετροποιηθούν καταλλήλως, με τη βοήθεια των υπολοίπων μετα-λογικών κατηγορημάτων που παρουσιάστηκαν στο κεφάλαιο αυτό, και να αποτελέσουν κώδικα βιβλιοθήκης για χρήση με οποιαδήποτε άλλη ανάλογη εφαρμογή διαχείρισης βάσης δεδομένων.

Για παράδειγμα, η διαδικασία της αποθήκευσης της βάσης δεδομένων σε αρχείο μπορεί να παραμετροποιηθεί ώστε να δέχεται ως παράμετρο τη γενική μορφή των γεγονότων που πρέπει να αποθηκευθούν σε αρχείο. Τα δύο ορίσματα του κατηγορήματος save_facts/2 καθορίζουν το κατηγόρημα των γεγονότων που θα αναζητηθούν - αποθηκευτούν και την τάξη του κατηγορήματος. Με τη χρήση του κατηγορήματος functor/3 κατασκευάζεται ως σύνθετος όρος Term ο ατομικός τύπος της κλήσης που θα ανακαλέσει τα αντίστοιχα γεγονότα και στη συνέχεια μέσω μεταβλητής κλήσης γίνεται η ανάκληση. Ο "πλήρως ορισμένος" όρος Term πλέον εγγράφεται στο αρχείο.

```
save_facts(Predicate,Arity) :-
        % Κατασκευή όρου/ατομικού τύπου
  functor(Term,Predicate,Arity),
        % Μεταβλητή κλήση - Αναζήτηση - Ανάκληση γεγονότος
  Term,
        % Εγγραφή στο αρχείο
  write(Term),
        % Εγγραφή "τελείας"
  write('.'), nl,
        % Αποτυχία και επιστροφή στο επόμενο γεγονός
  fail.

        % Όταν δεν υπάρχουν άλλα γεγονότα, απλά επιτυγχάνει
save_facts(_,_).
```

Η κλήση του κατηγορήματος save_facts/2 από την επιλογή 2 του αρχικού μενού θα γίνεται ως εξής:

```
  ...
  save_facts(student,2),
  ...
```

Αναλόγως μπορούν να παραμετροποιηθούν και οι υπόλοιπες επιλογές του αρχικού μενού που σχετίζονται με την διαχείριση των γεγονότων, αρκεί βέβαια να υπάρχει τρόπος να παραμετροποιηθούν τα αντίστοιχα μηνύματα. Για παράδειγμα, όταν τυπώνεται ο βαθμός ενός φοιτητή, μπροστά από τον βαθμό τυπώνεται το μήνυμα 'Grade: '. Για να γενικευθεί το κατηγόρημα που περιέχει αυτήν την εκτύπωση, θα πρέπει αυτό το μήνυμα κάπως να παραμετροποιηθεί για να δίνεται ως όρισμα εισόδου στο αντίστοιχο κατηγόρημα. Αυτό μπορεί να γίνει με κάποια λίστα παραμέτρων των γεγονότων η οποία θα ονοματίζει όλα τα ορίσματα, κάτι σαν το σχήμα (schema) μιας βάσης δεδομένων δηλαδή. Η υλοποίηση μιας τέτοιας επέκτασης αφήνεται ως άσκηση.

10.9 Μετα-διερμηνέας της Prolog σε Prolog

Μία από τις μεγαλύτερες προκλήσεις στις γλώσσες προγραμματισμού ήταν η υλοποίηση ενός διερμηνέα (interpreter) για τη γλώσσα εκφρασμένος στην ίδια τη γλώσσα. Η Prolog είναι το μοναδικό παράδειγμα γλώσσας προγραμματισμού στην οποία μία τέτοια υλοποίηση είναι η απλούστερη δυνατή. Στη Prolog, ένας απλός αλλά πλήρης μετα-διερμηνέας (meta-interpreter) κωδικοποιείται με τρεις φράσεις:

```
prove(true).          %1
prove((B1,B2)):-      %2
```

```
    prove(B1),
    prove(B2).
prove(G):-          %3
    clause(G,B),
    prove(B).
```

Στο παραπάνω πρόγραμμα, η φράση 1 λύνει τη βασική περίπτωση του ενσωματωμένου κατηγορήματος true/0. Η φράση 2 αφορά σύζευξη στόχων (B1,B2), η οποία επιλύεται με την επιμέρους επίλυση του κάθε υποστόχου. Η φράση 3 αφορά στην επίλυση στόχων που ορίζονται με φράσεις της μορφής G:-B στο πρόγραμμα. Αν ο υποστόχος G ταιριάζει με μία τέτοια φράση, τότε για να επιλυθεί, θα πρέπει να επιλυθούν όλοι οι υποστόχοι της φράσης B. Είναι φανερό ότι ο παραπάνω μετα-διερμηνέας υλοποιεί πλήρως το μηχανισμό εκτέλεσης Prolog προγραμμάτων.

Στο παράδειγμα του κοινωνικού δικτύου, η ερώτηση για τη μετα-διερμηνεία του στόχου friends(petros,ilias) γίνεται:

```
?-prove(friends(petros,ilias)).
```

Λόγω της περίπτωσης 3, υπάρχει φράση στο πρόγραμμα, (friends(petros,ilias) :- follows(petros,ilias), follows(ilias,petros)) η οποία επιλύει την ερώτηση, άρα το επόμενο βήμα είναι να επιλυθεί η σύζευξη υποστόχων prove((follows(X,Y),follows(Y,X))). Λόγω της περίπτωσης 2, θα πρέπει να κληθεί πρώτα το prove(follows(petros,ilias)) και μετά το prove(follows(ilias,petros)). Για το πρώτο, λόγω της περίπτωσης 3, υπάρχει φράση στο πρόγραμμα, (follows(ilias,petros):-true), και άρα πρέπει να κληθεί το prove(true) το οποίο λόγω της περίπτωσης 1 είναι αληθές. Το ίδιο γίνεται και με τον υποστόχο prove(follows(ilias,petros)). Άρα η ερώτηση ?-friends(petros,ilias) έχει αποδειχθεί.

Στον μετα-διερμηνέα μπορεί κάποιος να προσθέσει φράσεις για την επίλυση διάζευξης υποστόχων καθώς και για την άρνηση (της Prolog). Για παράδειγμα:

```
prove((B1;B2)):-
    prove(B1);
    prove(B2).
```

και

```
prove(not(G)):-
    not(prove(B)).
```

αντίστοιχα. Γενικά, αυτός ο μετα-διερμηνέας αποτελεί μία καλή βάση για την υλοποίηση μετα-διερμηνέων συστημάτων κανόνων EAN-TOTE (IF-THEN), όπως αυτά που θα δούμε στο Κεφάλαιο 13.

Βιβλιογραφία

Σχεδόν σε όλα τα "κλασσικά" βιβλία αναφοράς για τη γλώσσα προγραμματισμού Prolog, όπως τα (Clocksin and Mellish, 2003), (Bratko, 2011), (Sterling and Shapiro, 1994) και (Covington et al., 1988), υπάρχει εκτενής ανάλυση των μετα-λογικών χαρακτηριστικών της γλώσσα Prolog και αναλυτική παρουσίαση των αντίστοιχων ενσωματωμένων κατηγορημάτων. Ιδιαίτερη μνεία γίνεται στο βιβλίο των (Sterling and Shapiro, 1994), στο οποίο εκτός από απλή παρουσίαση των δυνατοτήτων των μετα-λογικών κατηγορημάτων, γίνεται αναφορά και στις λογικές προεκτάσεις των κατηγορημάτων αυτών και για ποιον λόγο θεωρούνται μετα-λογικά. Επίσης, στο βιβλίο του (O'Keefe, 1990) γίνεται συγκριτική παρουσίαση των κατηγορημάτων διαχείρισης λύσεων καθώς και παρουσιάζονται πειράματα που ελέγχουν και συγκρίνουν την αποδοτικότητα του κάθε κατηγορήματος.

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Clocksin, W. F. and Mellish, C. S. (2003). *Programming in Prolog: Using the ISO Standard*. 5th edition. Springer.

Covington, M. A. and Nute, D. and Vellino A. (1988). *Prolog Programming in Depth*. Scott Foresman & Co. ISBN 9780673186591.

O'Keefe, R. A. (1990), *The craft of Prolog*, Cambridge, Mass: MIT Press

Sterling, L. and Shapiro, E. Y. (1994). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, Mass: MIT Press.

Ασκήσεις

10.1 Ως γνωστόν, το ενσωματωμένο κατηγόρημα **is/2** δεν πρέπει να περιέχει ελεύθερες μεταβλητές στην μαθηματική έκφραση του δεύτερου ορίσματός του, αλλιώς θα επιστρέψει μήνυμα λάθους. Π.χ. είναι αδύνατο να εκτελεστεί η παρακάτω κλήση:

```
?- 3 is 2 + X.
```

Όμως, αν συμπεριφερόταν "λογικά" η παραπάνω κλήση, θα έπρεπε να μπορούσε να βρει για ποια τιμή του X επαληθεύεται η παραπάνω εξίσωση. Το ζητούμενο της άσκησης είναι να υλοποιηθεί το κατηγόρημα plus(X,Y,Z) το οποίο να αληθεύει όταν το Z είναι το άθροισμα των X και Y, ανεξάρτητα από το ποιο όρισμα είναι αριθμός και ποιο μεταβλητή. Εννοείται ότι μόνο το ένα από τα τρία ορίσματα μπορεί να είναι μεταβλητή κάθε φορά:

```
?- plus(1,2,Z).
Z = 3

?- plus(X,2,3).
Z = 1

?- plus(1,Y,3).
Y = 2

?- plus(1,2,3).
true
```

10.2 Στην άσκηση 6.7 του Κεφαλαίου 6 δόθηκε το κατηγόρημα simplify(Expr1,Expr2), το οποίο απλοποιεί αλγεβρικές εκφράσεις. Για παράδειγμα:

```
?- simplify( 1 + 1 + a, E).
E = 2+a
```

Όμως όταν οι αριθμοί δεν είναι μαζεμένοι όλοι μαζί σε μια πλευρά μιας μαθηματικής έκφρασης, τότε δεν απλοποιούνται από την συγκεκριμένη υλοποίηση του simplify/2:

```
?- simplify( 1 + a + 1, E).
E = 1+a+1
```

Να επεκτείνετε τον κώδικα του simplify/2 ώστε να μπορεί να χειριστεί τέτοιες εκφράσεις με αριθμούς και σύμβολα (άτομα). Η επέκταση αυτή να αναδιατάσσει σε μία αριθμητική έκφραση τους όρους έτσι ώστε όλα τα σύμβολα να προηγούνται των αριθμών. Η νέα εκδοχή του κατηγορήματος θα πρέπει να δίνει τις ακόλουθες απαντήσεις:

```
?- simplify( 1 + 1 + a, E).
E=a+2

?- simplify( 1 + a + 4 + 2 + b + c, E).
E=a+b+c+7

?- simplify( 3 + x + x, E).
E = 2*x+3
```

10.3 Να υλοποιήσετε το κατηγόρημα reverse_args(T1,T2), το οποίο να αντιστρέφει τα ορίσματα ενός όρου T1, επιστρέφοντας τον όρο T2:

```
?- reverse_args(a(b,c,d,e),What).
What = a(e,d,c,b)
```

10.4 Να υλοποιήσετε το κατηγόρημα subterm(ST,T), το οποίο αληθεύει όταν ο όρος ST είναι υπο-όρος, δηλαδή υπάρχει αυτούσιος μέσα στον όρο T. Θεωρήστε ότι και οι δύο όροι είναι "πλήρως ορισμένοι" (grounded).

```
?- subterm(b,f(a,g(b))).
true
```

10.5 Ως επέκταση της άσκησης 1, προβληματιστείτε με την περίπτωση που ο ένας εκ των δύο όρων δεν είναι πλήρως ορισμένος. Π.χ.:

```
?- subterm(b,f(a,X)).
```

Τι απάντηση επιστρέφει το κατηγόρημα που αναπτύξατε στην άσκηση 1; Τι αλλαγές χρειάζεται να κάνετε ώστε η απάντηση στο παραπάνω ερώτημα να είναι false;

10.6 Να υλοποιήσετε το κατηγόρημα substitute(ST,T,ST1,T1), το οποίο αληθεύει αν ο όρος T1 προκύπτει από τον όρο T με αντικατάσταση του υπο-όρου ST με ST1. Αν ο όρος ST δεν υπάρχει μέσα στον όρο T να επιστρέφεται ο όρος T αναλλοίωτος

```
?- substitute(a+b, f(1,a+b), c, A).
A = f(1,c)
```

10.7 Να υλοποιήσετε το κατηγόρημα occurrences(Sub, Term, N), το οποίο να επιστρέφει τον αριθμό N των εμφανίσεων του υπο-όρου Sub μέσα στον όρο Term. Αρχικά, θεωρήστε ότι και οι δύο όροι είναι "πλήρως ορισμένοι" (grounded). Στην συνέχεια μπορείτε να άρετε αυτήν την θεώρηση.

10.8 Να υλοποιήσετε το κατηγόρημα count_atom(Atom, Term, N) το οποίο να επιστρέφει τον αριθμό **N** των εμφανίσεων του ατόμου Atom μέσα στον όρο Term, είτε ως όρισμα, είτε ως συναρτησιακό σύμβολο:

```
?- count_atom(a,f(a,[b,a(X),d]),N).
N = 2
```

10.9 Να υλοποιήσετε το κατηγόρημα position(Subterm, Term, Position), το οποίο να επιστρέφει μία λίστα Position η οποία να αντιπροσωπεύει τις συντεταγμένες της θέσης του υπο-όρου Subterm μέσα στον όρο Term. Για παράδειγμα:

```
?- position(X,2*sin(X),Pos).
Pos = [2,1]
```

καθώς το sin(X) βρίσκεται στην δεύτερη θέση του "εξωτερικού" σύνθετου όρου */2 και το X βρίσκεται μέσα στο sin(X) στην πρώτη θέση, δηλαδή για να βρει κάποιος το X μέσα στον αρχικό σύνθετο όρο πρέπει να "πλοηγηθεί" πρώτα στο δεύτερο όρισμα του "εξωτερικού" όρου και μετά στο πρώτο όρισμα του "εσωτερικού" όρου.

10.10 Να υλοποιήσετε το κατηγόρημα map(Pred, List, Result) στο οποίο Pred είναι το όνομα ενός κατηγορήματος με δύο ορίσματα, List είναι μία λίστα από τιμές οι οποίες χρησιμοποιούνται ως πρώτο όρισμα του Pred, και Result είναι η αντίστοιχη λίστα τιμών του δευτέρου ορίσματος. Το κατηγόρημα επιτυγχάνει όταν η κλήση του κατηγορήματος Pred επιτυγχάνει για όλα τα αντίστοιχα στοιχεία των δύο λιστών, δηλαδή με άλλα λόγια "εκτελεί" το κατηγόρημα Pred χρησιμοποιώντας ως ορίσματα τα αντίστοιχα στοιχεία των δύο λιστών:

```
?- map(student,[nick,george,sofia],Marks).
Marks = [9,5,10]
```

10.11 Να υλοποιήσετε τη δική σας εκδοχή του ενσωματωμένου κατηγορήματος abolish(P, N) το οποίο διαγράφει όλες τις προτάσεις του κατηγορήματος P με τάξη N.

10.12 Να αναπτύξετε ένα λογικό πρόγραμμα το οποίο να ελέγχει αν κάποιος ακέραιος αριθμός είναι πρώτος αριθμός. Φροντίστε το πρόγραμμά σας να γίνεται πιο αποδοτικό όσο πιο πολύ χρησιμοποιείται.

10.13 Έστω το παρακάτω σύνολο γεγονότων με στοιχεία των ενοίκων μιας πολυκατοικίας:

```
/* person(name,age,room_no).*/
person(dimitris,31,6).
person(kostas,32,5).
person(maria,22,1).
person(anna,31,5).
person(anna,12,6).
person(dimitris,11,5).
```

Να υλοποιήσετε κατηγορήματα, τα οποία να επιστρέφουν:

- τα ονόματα όλων όσων μένουν στην πολυκατοικία.
- τα ονόματα όσων μένουν στο δωμάτιο 5.
- την μικρότερη ηλικία ενοίκου της πολυκατοικίας.
- το όνομα του ενοίκου με την μικρότερη ηλικία.
- τον αριθμό δωματίου στο οποίο διαμένει ο ένοικος με την μικρότερη ηλικία.
- τα δωμάτια στα οποία υπάρχει / δεν υπάρχει συγκατοίκηση.

10.14 Στην ενότητα 11.3 του επόμενου κεφαλαίου παρουσιάζεται το πρόβλημα της αναζήτησης σε έναν γράφο και συγκεκριμένα παρουσιάζονται κάποια κατηγορήματα για την εύρεση της διαδρομής μεταξύ δύο σημείων path_loopcheck/3 στον γράφο καθώς και του κόστους αυτής της διαδρομής path_weight/4. Τα κατηγορήματα αυτά, σε περίπτωση που υπάρχουν περισσότερες της μιας εναλλακτικές διαδρομές μεταξύ δύο σημείων, τις επιστρέφουν όλες μέσω οπισθοδρόμησης. Να υλοποιήσετε τα κατηγορήματα shortest_path_loopcheck(Start, End, Path, Length) και min_path_weight(Start, End, Path, Cost), τα οποία να επιστρέφουν τη συντομότερη διαδρομή και την διαδρομή ελάχιστου κόστους, αντίστοιχα, μεταξύ δύο σημείων ενός γράφου.

10.15 Να υλοποιήσετε το κατηγόρημα intersect(X,Y,Z), το οποίο επιστρέφει την τομή Z δύο συνόλων X και Y, θεωρώντας ότι τα σύνολα αναπαρίστανται ως λίστες στις οποίες δεν υπάρχουν διπλά στοιχεία, με την χρήση κάποιου (ενός ή περισσοτέρων) κατηγορήματος διαχείρισης λύσεων. Τι συμβαίνει όταν τα σύνολα X και Y είναι ξένα μεταξύ τους; Αντιπαραβάλετε τον συγκεκριμένο κώδικα με αντίστοιχη αναδρομική υλοποίηση, η οποία παρουσιάστηκε στην ενότητα 9.2.

10.16 Να υλοποιήσετε το κατηγόρημα powerset(Set, Subsets), το οποίο να επιστρέφει το δυναμοσύνολο Subsets του συνόλου Set, δηλαδή το σύνολο όλων των δυνατών υποσυνόλων του συνόλου Set, με την χρήση του κατηγορήματος bagof/3.

ΚΕΦΑΛΑΙΟ 11: Τεχνικές Λογικού Προγραμματισμού για Επίλυση Προβλημάτων

Λέξεις Κλειδιά:
Λογική και Αλγοριθμική, Προβλήματα Παραγωγής και Δοκιμής, Προβλήματα Γράφων. Αναπαράσταση και αναζήτηση σε γράφους.

Περίληψη
Το κεφάλαιο αποτελεί επιστέγασμα όλων των προηγουμένων, καθώς παραθέτει παραδείγματα τεχνικών αναπαράστασης και επίλυσης σύνθετων προβλημάτων σε Prolog. Αντιπαρατίθεται η λογική προσέγγιση σε ένα πρόβλημα με την αντίστοιχη αλγοριθμική. Κλασσικές τεχνικές όπως εκείνη της "Παραγωγής και Δοκιμής" (Generate and Test) παρουσιάζονται μέσω κλασικών παραδειγμάτων, όπως η εύρεση λύσης σε ακέραιες εξισώσεις και άλλα παρόμοια μη-αριθμητικά προβλήματα ικανοποίησης περιορισμών. Παρουσιάζεται επίσης η πλέον δημοφιλής αναπαράσταση προβλημάτων της Τεχνητής Νοημοσύνης, η αναπαράσταση μέσω γράφων, καθώς και η υλοποίηση τυφλών και ευριστικών αλγορίθμων αναζήτησης, οι οποίοι οδηγούν στην επίλυση σύνθετων προβλημάτων.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός να:

- κατανοήσει τις βασικές διαφορές μεταξύ λογικής και αλγοριθμικής προσέγγισης,
- αντιληφθεί την βασική αλγοριθμική για την εύρεση διαδρομής σε ένα γράφο,
- ορίζει προβλήματα χώρου καταστάσεων στην Τεχνητή Νοημοσύνη,
- καταλάβει τις βασικές αρχές αναζήτησης και επίλυσης προβλημάτων στην Τεχνητή Νοημοσύνη,
- υλοποιήσει επιλύσεις με Παραγωγή και Δοκιμή και Αλγόριθμους Αναζήτησης Πρώτα σε Βάθος, Πρώτα σε Πλάτος και Πρώτα στο Καλύτερο.

Παράδειγμα κίνητρο

Η αυτόματη επίλυση προβλημάτων αποτελεί βασικό στόχο της Τεχνητής Νοημοσύνης (ΤΝ). Τα προβλήματα που καλείται να λύσει η ΤΝ ώστε το αποτέλεσμα (η λύση) να είναι καλύτερο ή τουλάχιστον όσο καλό είναι το αποτέλεσμα που παράγει ο άνθρωπος, είναι εξαιρετικής πολυπλοκότητας. Το μεγάλο πλεονέκτημα είναι ότι μία υπολογιστική μηχανή έχει πολύ μεγάλη υπολογιστική ισχύ, εκτελώντας 10^{15} εντολές ανά δευτερόλεπτο.

Ένα τέτοιο πρόβλημα είναι το λεγόμενο πρόβλημα του πλανόδιου πωλητή (Travelling Salesperson Problem ή TSP). Το πρόβλημα παριστάνεται με ένα γράφο που αποτελείται από Ν κόμβους οι οποίοι αντιπροσωπεύουν αντίστοιχα Ν τοποθεσίες. Κάποιος πρέπει να ξεκινήσει από έναν κόμβο, να επισκεφτεί μία φορά την κάθε τοποθεσία και να επιστέψει στην αρχική με το λιγότερο συνολικό κόστος, διανύοντας δηλαδή την ελάχιστη δυνατή απόσταση. Θεωρούμε ότι όλοι οι κόμβοι ενώνονται μεταξύ τους (πλήρης γράφος). Το πρόβλημα έχει πολλές εφαρμογές σε σειριακό σχεδιασμό ενεργειών με κόστος για κάθε ενέργεια. Οι πιθανές διαδρομές είναι (N-1)! και μία από αυτές είναι η βέλτιστη λύση. Το πρόβλημα ανήκει στην κατηγορία προβλημάτων με λύση μη-πολυωνυμικού χρόνου (NP-complete). Η εξαντλητική εξέταση (N-1)! μονοπατιών για Ν πόλεις είναι ένας πολύ μεγάλος αριθμός ακόμα και για μικρό αριθμό πόλεων. Για παράδειγμα, αν N=20, τότε υπάρχουν $19!=1.216\times10^{17}$ λύσεις, που σημαίνει ότι ακόμη και ένας υπολογιστής που εξετάζει 10^6 λύσεις το δευτερόλεπτο θα χρειαζόταν 385 χρόνια για να βρει τη βέλτιστη λύση.

Μονοπάτι Μάθησης

Οι ενότητες αυτού του κεφαλαίου χωρίζονται σε δύο ομάδες, την 11.1, την 11.2, την 11.3 που αποτελούν κλασσικές εφαρμογές και αναδεικνύουν βασικές τεχνικές του λογικού προγραμματισμού, και τις ενότητες 11.4 και 11.5 που δείχνουν βασικούς αλγορίθμους επίλυσης προβλημάτων. Οι τρεις πρώτες μπορούν να διαβαστούν ανεξάρτητα. Πριν την κατανόηση των 11.4 και 11.5 απαιτείται η κατανόηση της εύρεσης διαδρομής σε γράφους που περιγράφεται αναλυτικά στην ενότητα 11.3.

11.1 Λογική εναντίον Αλγοριθμικής

Ένα από τα σημαντικότερα θέματα στον προγραμματισμό είναι η αποτελεσματικότητα (efficiency) της εκτέλεσης ενός προγράμματος. Η αποτελεσματικότητα αυτή μπορεί να μετρηθεί με πολλούς παράγοντες, αλλά το πιο σημαντικό από αυτά είναι ο χρόνος εκτέλεσης και οι απαιτήσεις μνήμης που καταναλώνεται κατά τη διάρκεια της εκτέλεσης. Έχουν γίνει σημαντικές προσπάθειες για τη δημιουργία αποτελεσματικών μεταγλωττιστών και βελτιστοποιήσεων για όλες τις γλώσσες προγραμματισμού που θα μειώνουν το χρόνο εκτέλεσης και το χώρο διαχείρισης. Η Prolog δεν αποτελεί εξαίρεση στην ανάγκη ταχύτερης εκτέλεσης. Αυτή η ανάγκη ενισχύεται από το γεγονός ότι υπάρχει μια αντιστρόφως ανάλογη σχέση ανάμεσα στη λογική έκφραση ενός προβλήματος και σε κάποιον διαδικαστικό αλγόριθμο που το λύνει. Όσο πιο δηλωτικός είναι ο κώδικας τόσο ο χρόνος εκτέλεσης αυξάνεται, και όσο πιο διαδικαστικός είναι τόσο ο χρόνος εκτέλεσης μειώνεται. Ένα πρώτο παράδειγμα είδαμε στο Κεφάλαιο 7, με την υλοποίηση του κατηγορήματος εύρεσης μεγίστου μιας λίστας (max/2). Θα εξεταστεί εδώ αναλυτικότερα με μια μελέτη περίπτωσης, την ταξινόμηση αριθμών μέσα σε μία λίστα.

Ταξινόμηση Λίστας: η Λογική Προσέγγιση

Ταξινόμηση μία λίστας αριθμών είναι το πρόβλημα εύρεσης μίας μετάθεσης (permutation) των αριθμών μέσα στην λίστα, έτσι ώστε αυτοί να εμφανίζονται σε αύξουσα (ή φθίνουσα) σειρά. Στην Prolog αυτό μπορεί να εκφραστεί ως ο ορισμός:

```
nsort(List, PermList): -
  permutation(List, PermList),
  is_sorted(PermList).
```

όπου το κατηγόρημα permutation/2 παράγει όλες τις πιθανές μεταθέσεις μιας λίστας, και is_sorted/2 ένα κατηγόρημα που εξετάζει κατά πόσον μία λίστα από αριθμούς είναι ταξινομημένη. Το permutation/2 είναι συνήθως ενσωματωμένο κατηγόρημα στην Prolog ενώ το is_sorted/1 μπορεί εύκολα να οριστεί αναδρομικά (μία λίστα είναι ταξινομημένη αν η ουρά της είναι ταξινομημένη και το πρώτο στοιχείο της είναι μεγαλύτερο από το δεύτερο στοιχείο).

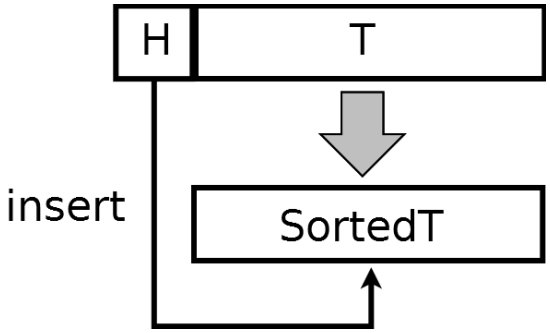
Ο παραπάνω ορισμός αποτελεί την ουσία του δηλωτικού προγραμματισμού, με την έννοια ότι δείχνει "τί" είναι η ταξινόμηση και όχι "πώς" γίνεται. Πρακτικά, ο μηχανισμός εκτέλεσης της Prolog δημιουργεί μεταθέσεις τις οποίες το κατηγόρημα is_sorted εξετάζει αν είναι ταξινομημένες. Αν κάποια δεν είναι, μία νέα μετάθεση δημιουργείται μέσω της οπισθοδρόμησης για να εξεταστεί και αυτή με τη σειρά της, με τη διαδικασία να τελειώνει όταν βρεθεί η κατάλληλη λίστα. Η διαδικασία αυτή ονομάζεται “παραγωγή και δοκιμής” (generate and test) ή “προσπάθεια και αποτυχία” (trial and error) στην οποία θα αναφερθούμε εκτενώς στη συνέχεια.

Ταξινόμηση Λίστας: η Αναδρομική Προσέγγιση

Εφαρμόζοντας την αρχή της αναδρομής, “για να ταξινομήσω μία λίστα με αριθμούς, αρκεί να υποθέσω ότι η ουρά της είναι ταξινομημένη, οπότε αυτό που απομένει είναι να εισάγω την κεφαλή της λίστας στη σωστή της θέση” (Σχήμα 11.1). Σε ορισμό Prolog, αυτό εκφράζεται:

```
isort([], []).
isort([H|T], SortedList):-
  isort(T, SortedT),
  insert_sorted(H, SortedT, SortedList).
```

όπου το κατηγόρημα insert_sorted/3, εισάγει ένα αριθμό σε μία ταξινομημένη λίστα στη σωστή του θέση, δηλαδή ανάμεσα σε ένα μεγαλύτερο και ένα μικρότερο αριθμό. Μπορείτε να δείτε ένα video για την υλοποίηση του isort/2 εδώ.

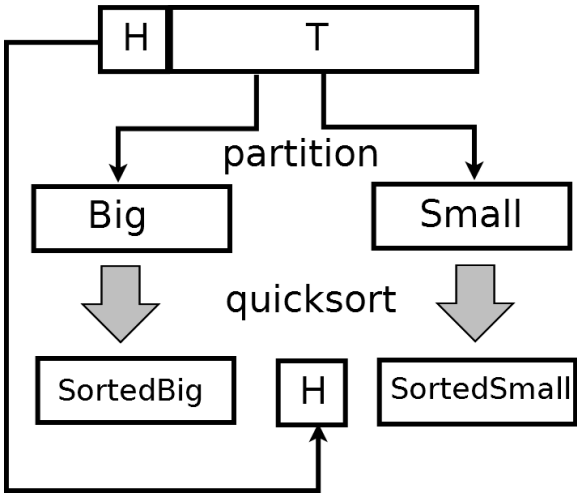


Σχήμα 11.1: *Αναδρομική ταξινόμηση με εισαγωγή*

Ο ορισμός αυτός είναι ένας τυπικός αναδρομικός ορισμός, όπως πιθανότερα θα σκεφτόταν ένας εξοικειωμένος προγραμματιστής της Prolog. Όπως και ο προηγούμενος είναι δηλωτικός αλλά με κάποια επιπλέον στοιχεία διαδικασίας εύρεσης της λύσης, όπως κάθε αναδρομικός ορισμός.

Ταξινόμηση Λίστας: η Αλγοριθμική Προσέγγιση

Αλγοριθμικά η ταξινόμηση μπορεί να εκφραστεί με πολλούς τρόπους, ένας από αυτούς είναι η γρήγορη ταξινόμηση (quicksort). Σύμφωνα με αυτόν τον αλγόριθμο, η αρχική λίστα χωρίζεται σε δύο λίστες, μια που περιέχει τους αριθμούς μεγαλύτερους από την κεφαλή της λίστας και μια που περιέχει τους μικρότερους. Στη συνέχεια, αυτές οι δύο λίστες ταξινομούνται με τον ίδιο τρόπο (αναδρομή) και μετά συνενώνονται σε μία μεγαλύτερη έχοντας στη μέση την αρχική κεφαλή. Προφανώς η συνολική λίστα είναι και ταξινομημένη. Γραφικά, ο αλγόριθμος απεικονίζεται στο Σχήμα 11.2.



Σχήμα 11.2: *Γρήγορη ταξινόμηση*

Ο αντίστοιχος Prolog κώδικας είναι:

```
qsort([],[]).
qsort([H|T], SortedList):-
  partition(H, T, Big, Small),
  qsort(Big, SortedBig),
  qsort(Small, SortedSmall),
  append(SortedBig, [H | SortedSmall], SortedList).
```

όπου το κατηγόρημα partition/4 χωρίζει μία λίστα L με βάση έναν αριθμό H, σε δύο άλλες λίστες, την Big που περιέχει όλους τους αριθμούς μεγαλύτερους από το H και την Small που περιέχει όλους τους αριθμούς μικρότερους από το H.

Είναι εμφανές ότι ο ορισμός αυτός είναι ο πιο διαδικαστικός (αλγοριθμικός) από τους προηγούμενους, και συμβιβάζει την εύκολη, γρήγορη και ευανάγνωστη κωδικοποίηση με δηλωτικά χαρακτηριστικά με την αποδοτικότητα σε χρόνο εκτέλεσης.

Επιλύοντας τα διλήμματα;

Θα ήταν ενδιαφέρον κάποιος να τρέξει κάποια "πειράματα" για να βεβαιώσει του λόγου το αληθές. Η αποδοτικότητα της εκτέλεσης ενός προγράμματος Prolog μετριέται με:

- cputime: ο χρόνος εκτέλεσης για την απάντηση
- inferences: ο συνολικός αριθμός λογικών συνεπαγωγών, δηλαδή των κλήσεων σε κατηγορήματα
- localused: ο όγκος της τοπικής στοιβάδας (local stack) που χρησιμοποιήθηκε, για τις κλήσεις των κατηγορημάτων
- gloabalused: ο όγκος της καθολικής στοιβάδας (global stack) που χρησιμοποιήθηκε, για τους όρους, τις λίστες, τις μεταβλητές και τις τιμές τους.

Το ενσωματωμένο κατηγόρημα statistics/2 με όρισμα ένα από τα παραπάνω επιστρέφει την αντίστοιχη τιμή. Έτσι, ο παρακάτω ορισμός μπορεί να χρησιμοποιηθεί για να μετρήσει την απόδοση όλων των προγραμμάτων ταξινόμησης που παρουσιάστηκαν:

```
test(What,Predicate,List,Result):-
  Call=..[Predicate,L,Result],
  statistics(What,V1),
  Call,
  statistics(What,V2),
  V is V2-V1,
  nl,write(What),write('='),write(V),nl.
```

Αξίζει να τονιστεί ότι ακόμη και μετά από έναν ικανοποιητικό αριθμό "πειραμάτων" πάνω σε διαφορετικές λίστες, δεν είναι ξεκάθαρος ο νικητής σε όλες τις κατηγορίες των μετρήσεων. Μπορεί να βρεθεί ότι κάποια κωδικοποίηση υπερτερεί των άλλων ως προς το χρόνο, αλλά υστερεί ως προς τον αποθηκευτικό χώρο που απαιτεί.

Στις περισσότερες υλοποιήσεις της Prolog υπάρχουν ενσωματωμένα κατηγορήματα για ταξινόμηση λίστας, όπως τα sort/2 και keysort/2, τα οποία λειτουργούν και για άλλους τύπους όρων (χαρακτήρες, συμβολοσειρές κλπ).

11.2 Παραγωγή και Δοκιμή

Η στρατηγική της **Παραγωγής και Δοκιμής (Genarate and Test)** ή **Προσπάθειας και Αποτυχίας (Trial and Error)** είναι η πιο αφελής στρατηγική επίλυσης προβλημάτων. Η διαδικασία είναι η δημιουργία μιας πιθανής λύσης και ο έλεγχος για το αν αυτή η λύση τηρεί κάποιες προδιαγραφές. Αν όχι, μια άλλη λύση δημιουργείται και επανελέγχεται, με τη διαδικασία να επαναλαμβάνεται μέχρις ότου βρεθεί η σωστή λύση το πρόβλημα. Αυτό μοιάζει με το να προσπαθεί να ανοίξει κανείς ένα χρηματοκιβώτιο, παράγοντας έναν-έναν τους πιθανούς συνδυασμούς και δοκιμάζοντας αν κάθε ένας από αυτούς ανοίγει την πόρτα. Όσο και αν αυτή η στρατηγική φαίνεται αδύνατη για τον άνθρωπο, αυτή μπορεί να χρησιμοποιηθεί από προγράμματα λόγω της μεγάλης ταχύτητας εκτέλεσης. Φυσικά, η στρατηγική αυτή έχει καλά αποτελέσματα σε προβλήματα μικρού σχετικά μεγέθους.

Στην Prolog η στρατηγική της παραγωγής και δοκιμής υλοποιείται μέσω της οπισθοδρόμησης. Έτσι, κατηγορήματα παράγουν λύσεις που δοκιμάζονται από άλλα κατηγορήματα. Αν αυτά αποτύχουν τότε δημιουργούνται μέσω οπισθοδρόμησης άλλες λύσεις που δοκιμάζονται και αυτές με τη σειρά τους.

Εύρεση λύσης σε ακέραιες εξισώσεις

Ένα κλασικό παράδειγμα επίλυσης προβλήματος με παραγωγή και δοκιμή είναι αυτό της εύρεσης ακέραιων τιμών που ικανοποιούν αριθμητικές παραστάσεις. Για παράδειγμα, έστω το σύστημα των παραστάσεων: χ + ψ < 15 και χ - ψ = 5, όπου το χ ανήκει στο διάστημα [1,10] και το ψ στο διάστημα [1,5].

Το παρακάτω πρόγραμμα επιλύει το σύστημα παράγοντας τιμές για τα χ και ψ και δοκιμάζοντάς τες για το αν πληρούν τους περιορισμούς που τέθηκαν:

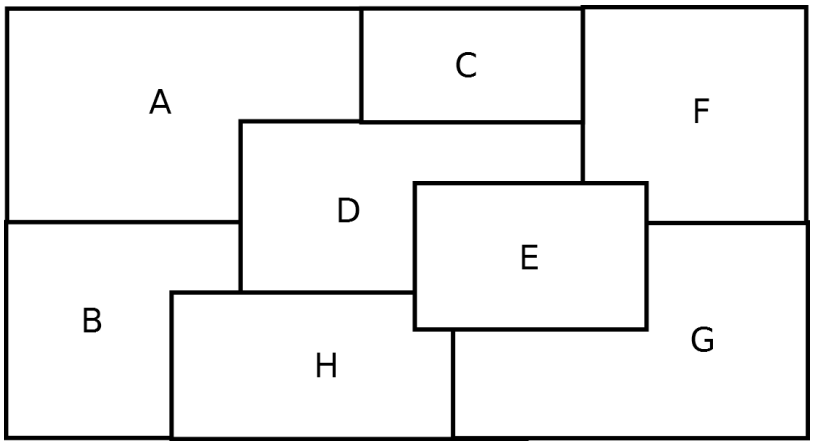
```
solve(X, Y) :-
  member(X, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
  member(Y, [1, 2, 3 ,4 ,5]),
  T is X+Y,
  T < 15,
  5 is X-Y.
```

Οι μεταβλητές X και Y δεσμεύονται σταδιακά με τιμές από τις λίστες [1..10] και [1..5] αντίστοιχα. Αμέσως μετά ελέγχονται οι περιορισμοί 5 is X-Y και X+Y<15. Αν και οι δύο αληθεύουν οι τιμές των X και Y αποτελούν λύση. Αν δεν αληθεύει ένας από αυτούς τότε η οπισθοδρόμηση παράγει τις επόμενες τιμές στη λίστα μέσω του member/2.

Άλλα προβλήματα ικανοποίησης περιορισμών

Υπάρχουν και άλλα προβλήματα, μη αριθμητικά, που εντάσσονται στη γενικότερη κατηγορία των **προβλημάτων ικανοποίησης περιορισμών (constraint satisfation problems**). Εκτενέστερη αναφορά στην επίλυση τέτοιων προβλημάτων μέσω Prolog θα γίνει σε επόμενο κεφάλαιο (Κεφάλαιο 12). Εδώ, απλά αναφέρονται στα πλαίσια της στρατηγικής με παραγωγή και δοκιμή. Τέτοια προβλήματα είναι το κρυπταριθμητικό παζλ (για παράδειγμα η εύρεση αριθμητικής τιμής για κάθε γράμμα της εξίσωσης SEND+MORE=MONEY), το πρόβλημα των N Βασιλισσών (τοποθέτηση N βασιλισσών σε μια σκακιέρα N x N, ώστε να μην απειλεί η μία την άλλη), το πρόβλημα χρωματισμού ενός χάρτη και πολλά άλλα.

Στο τελευταίο, το πρόβλημα του χρωματισμού ενός χάρτη, πρέπει να χρωματίσουμε ένα λευκό χάρτη με χρώματα έτσι ώστε η κάθε περιοχή να μην έχει το ίδιο χρώμα με οποιαδήποτε γειτονική της (Σχήμα 11.3).



Σχήμα 11.3: *Παράδειγμα ενός χάρτη προς χρωματισμό*

Ξεκινούμε με τη βασική αναπαράσταση του χάρτη σαν λίστα με ζευγάρια που έχουν περιοχή και το αντίστοιχο χρώμα της (αρχικά μη δεσμευμένη μεταβλητή):

```
template_map([(a,_),(b,_),(c,_),(d,_),(e,_),(f,_),(g,_),(h,_)]).
```

τα χρώματα που είναι διαθέσιμα:

```
colours([red, green, blue, yellow, orange]).
```

και τις γειτονικές περιοχές:

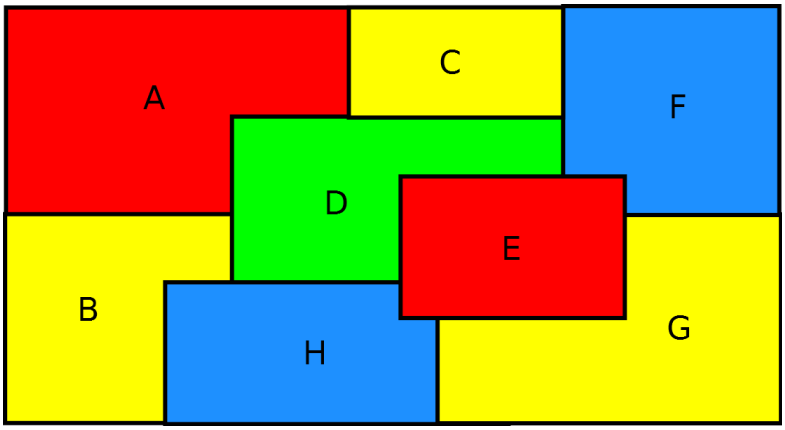
```
next(a,b).
next(a,c).
```

```
next(a,d).
next(b,h).
next(b,d).
...
neighbourgh(X,Y):-next(X,Y),!.
neighbourgh(X,Y):-next(Y,X),!.
```

Με βάση τα παραπάνω, ο κύριος ορισμός είναι:

```
map(Map):-
  colours(C),
  template_map(Map),
  solve_map(Map,C).
```

που παραπέμπει τη λύση του προβλήματος στο κατηγόρημα solve_map/3 που ουσιαστικά θα δώσει τιμές στις μεταβλητές χρώματος.



Σχήμα 11.4: *Παράδειγμα λύσης ενός χάρτη που χρωματίστηκε*

Ο ορισμός είναι φυσικά αναδρομικός: "*Για να χρωματίσουμε μία λίστα από χώρες, υποθέτουμε ότι έχουμε χρωματίσει τις υπόλοιπες χώρες (ουρά) και διαλέγουμε ένα χρώμα για την πρώτη χώρα (κεφαλή) που δεν έρχεται σε συγκρούεται με τις γειτονικές*". Αυτό σε μορφή κώδικα εκφράζεται:

```
solve_map([],_).
solve_map([(Country,Colour)|Rest],AllColours):-
  solve_map(Rest,AllColours),
  member(Colour,AllColours),
  not(conflict((Country,Colour),Rest)).
```

το οποίο με τη σειρά του παραπέμπει τη λύση στο κατηγόρημα conflict/2 που ελέγχει αν ο χρωματισμός μιας χώρας είναι ίδιος με το χρωματισμό μιας γειτονικής. Ο ορισμός είναι και πάλι αναδρομικός τηρώντας την αρχή της αναδρομής:

```
conflict((Country1,Colour),[(Country2,Colour)|_]):-
  neighbourgh(Country1,Country2),!.
conflict((Country,Colour),[_|Rest]):-
  conflict((Country,Colour),Rest).
```

Η κατάλληλη ερώτηση δίνει και τις σωστή απάντηση (Σχήμα 11.4) που μπορεί να είναι περισσότερες από μια:

```
?-map(M).
M=[(a,red),(b,yellow),(c,yellow),(d,green),(e,red),(f,blue),
(g,yellow),(h,blue)]
```

11.3 Αναζήτηση σε γράφο

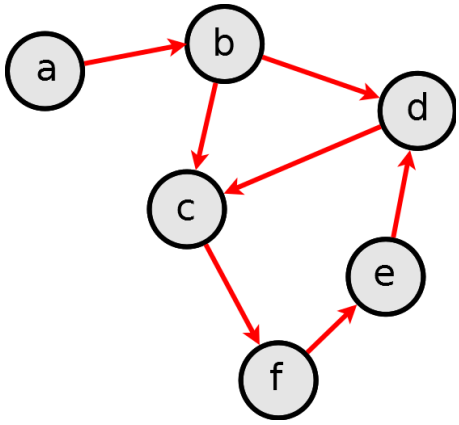
Πολλά από τα προβλήματα στον τομέα της Τεχνητής Νοημοσύνης, και όχι μόνον, ανάγονται σε προβλήματα αναζήτησης διαδρομής σε ένα γράφο. Περισσότερες λεπτομέρειες θα αναφερθούν σε παρακάτω ενότητα.

Ένας γράφος ορίζεται ως ένα σύνολο από κορυφές (κόμβους) και ακμές (συνδέσεις) μεταξύ των κόμβων. Οι γράφοι μπορεί να είναι κατευθυνόμενοι (υπάρχει ένας τρόπος μετάβασης από κόμβο σε κόμβο) ή μη-κατευθυνόμενοι (υπάρχει αμφίδρομη σύνδεση μεταξύ των κόμβων).

Ένα από τα πλέον κοινά προβλήματα στους γράφους είναι να βρούμε ένα μονοπάτι (αλληλουχία κόμβων ή ακμών) από έναν αρχικό κόμβο σε έναν άλλον τελικό. Πολλές φορές επιβάλλεται η τήρηση ορισμένων περιορισμών σχετικά με την τελική διαδρομή, για παράδειγμα απαιτείται η συντομότερη ή αυτή με το ελάχιστο κόστος διαδρομή. Όλα αυτά θα δοθούν βήμα-βήμα στις ενότητες που ακολουθούν.

Αναπαράσταση Γράφου

Το Σχήμα 11.5 αναπαριστά ένα κατευθυνόμενο γράφο.

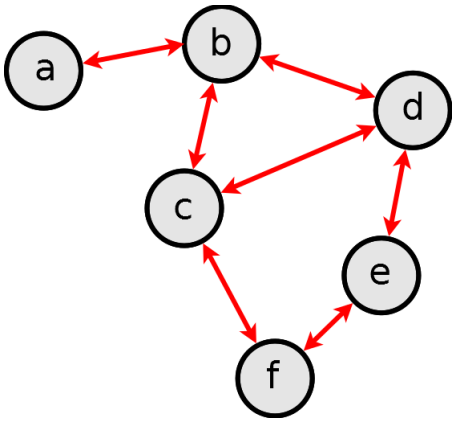


Σχήμα 11.5: *Ένας κατευθυνόμενος γράφος*

Ένας γράφος αναπαριστά σχέσεις μεταξύ κόμβων, όπως στο παράδειγμα του κοινωνικού δικτύου και του οικογενειακού δένδρου στα αρχικά κεφάλαια. Όπως σε εκείνα, έτσι και στο γενικό αφαιρετικό γράφο που παρουσιάζεται εδώ, οι συνδέσεις μπορεί να είναι ένα σύνολο από γεγονότα της μορφής:

```
link(a, b).
link(b, c).
link(b, d).
...
```

Το Σχήμα 11.6 αναπαριστά ένα μη-κατευθυνόμενο γράφο (οι ακμές δεν έχουν κατεύθυνση):



Σχήμα 11.6: *Ένας μη-κατευθυνόμενος γράφος*

Ένας απλός τρόπος για να αναπαραστήσουμε ένα μη-κατευθυνόμενο γράφο, είναι να διπλασιάσουμε τον αριθμό των γεγονότων για να δηλώσουμε την αντίστροφη σύνδεση, για παράδειγμα link(b,a). Αντ' αυτού, υπάρχει ένας καλύτερος τρόπος, με δύο κανόνες της μορφής:

```
next(X, Y) :- link(X, Y).
```

```
next(X, Y) :- link(Y, X).
```

όπου το κατηγόρημα next/2 θα είναι αυτό που αναπαριστά την αμφίδρομη σχέση μεταξύ των κόμβων.

Κάποιος θα μπορούσε να αναρωτηθεί γιατί ένας και μόνο επιπλέον κανόνας:

```
link(X, Y) :- link(Y, X).
```

δεν αρκεί για την αμφίδρομη σχέση; Η απορία έχει βάση εφόσον μια ερώτηση όπως η:

```
?- link(b,a).
```

αφού έψαχνε στα γεγονότα δε θα έβρισκε το link(b,a) αλλά μέσω του κανόνα θα επαλήθευε το link(a,b). Αυτό είναι σωστό. Όμως σε περίπτωση που ο δεσμός δεν υπάρχει, όπως για παράδειγμα στην ερώτηση:

```
?-link(a,c).
```

η εκτέλεση του προγράμματος δε θα τερμάτιζε ποτέ, λόγω του ατέρμονου βρόχου που δημιουργείται στην εναλλασσόμενη αναζήτηση του link(a,c) και link(c,a).

Μια άλλη αναπαράσταση των δεσμών μεταξύ κόμβων είναι με γεγονότα της μορφής:

```
links(a,[b]).
links(b,[a,c,d]).
links(c,[b,d,f]).
...
```

Ένας χωριστός κανόνας μπορεί να οριστεί για τον έλεγχο της σχέσης μεταξύ δύο κόμβων:

```
next(X, Y) :-
  links(X, ListofNodes),
  member(Y, ListofNodes).
```

Αυτή η αναπαράσταση απαιτεί λιγότερες φράσεις γεγονότα για όλο το γράφο αλλά χρειάζεται περισσότερο χρόνο για να βρει ή να επιβεβαιώσει δεσμούς ανάμεσα σε κόμβους λόγω της επιπλέον αναζήτησης του member/2 μέσα στη λίστα.

Ύπαρξη διαδρομής

Η αναζήτηση μιας διαδρομής που συνδέει δύο μακρινούς κόμβους είναι μάλλον κοινή για όλες τις εφαρμογές που περιλαμβάνουν γράφους ή δέντρα. Η αναζήτηση μπορεί να επιτευχθεί με πολλούς διαφορετικούς τρόπους. Υπάρχει πληθώρα τέτοιων αλγορίθμων αναζήτησης οι οποίοι δεδομένου ενός γράφου μπορούν να βρουν διαδρομές οι οποίες πληρούν ορισμένα κριτήρια. Αρχικά θα αναφερθούμε στον απλούστερο αλγόριθμο από όλους, την Αναζήτηση Πρώτα Κατά Βάθος (Depth-First Search), ο οποίος αντιστοιχεί στην στρατηγική εκτέλεσης της Prolog και είναι ως εκ τούτου πιο εύκολο να εφαρμοστεί σε αυτήν.

Έστω η ερώτηση:

```
?- exists_path(a, f).
yes
```

η οποία επιβεβαιώνει την ύπαρξη ενός διαδρομής ανάμεσα στο a και στο f. Το πρόγραμμα που υλοποιεί την εύρεση της διαδρομής είναι παρόμοιο με αυτό της εύρεσης διασύνδεσης μεταξύ φίλων στο κοινωνικό δίκτυο του κεφαλαίου της αναδρομής:

```
exists_path(Node, FinalNode):-
  next(Node, FinalNode).
exists_path(Node, FinalNode):-
  next(Node, SomeNextNode),
  exists_path(SomeNextNode, FinalNode).
```

Εύρεση διαδρομής

Το παραπάνω πρόγραμμα δεν είναι ιδιαίτερα ενδιαφέρον γιατί δεν επιστρέφει τη διαδρομή που συνδέει τους δύο κόμβους. Έστω η ερώτηση:

```
?-  path(a, f, Route).
Route = [a, b, c, f] ;
Route = [a, b, d, e, f] ;
```

```
Route = [a, b, d, c, f] ;
...
```

η οποία επιστρέφει πολλά διαφορετικά μονοπάτια σαν μία λίστα από κόμβους αρχίζοντας από τον αρχικό a μέχρι τον τελικό κόμβο f. Το πρόγραμμα παίρνει την εξής μορφή:

```
path(Node, FinalNode, [Node, FinalNode]):-
  next(Node, FinalNode).
path(Node, FinalNode, [Node | RestRoute]):-
  next(Node, SomeNextNode),
  path(SomeNextNode, FinalNode, RestRoute).
```

που ερμηνεύεται ως εξής: "*η διαδρομή μεταξύ δύο γειτονικών κόμβων Node και FinalNode περιέχει αυτούς τους κόμβους, αλλιώς (αν δεν είναι γειτονικοί), υποθέτοντας ότι υπάρχει μονοπάτι μεταξύ ενός γειτονικού κόμβου SomeNextNode μέχρι τον τελικό FinalNode που δίνεται σε μία λίστα, τότε το τελικό μονοπάτι είναι αυτή η λίστα με την προσθήκη του αρχικού κόμβου Node*".

Εύρεση διαδρομής χωρίς βρόχους

Παρόλο που το παραπάνω πρόγραμμα φαίνεται να δουλεύει σωστά, υπάρχει περίπτωση να εκτελείται ατέρμονα, λόγω των κυκλικών διαδρομών που εμφανίζονται στο γράφο. Για να μη γίνεται αυτό πρέπει να υπάρχει πρόβλεψη, ώστε η διαδικασία της αναζήτησης να μην περνά από τους ίδιους κόμβους.

Αυτό απαιτεί μία λίστα που θα κρατά τους κόμβους που έχει επισκεφθεί η αναζήτηση και έναν επιπλέον έλεγχο για το αν ένας προς επίσκεψη κόμβος είναι ήδη μέσα στη λίστα ή όχι. Το νέο πρόγραμμα γίνεται:

```
path_loopcheck(InitialNode, FinalNode, Route):-
  path_loopcheck (InitialNode, FinalNode, [InitialNode], Route).


path_loopcheck(Node, FinalNode, _, [Node, FinalNode]) :-
  next(Node, FinalNode).

path_loopcheck(Node, FinalNode, VisitedNodes, [Node | RestRoute]) :-
  next(Node, SomeNextNode),
  not(member(SomeNextNode, VisitedNodes)),
  path_loopcheck(SomeNextNode, FinalNode,
        [SomeNextNode|VisitedNodes], RestRoute).
```

όπου VisitedNodes είναι η λίστα που αναπαριστά τους κόμβους που έχει ήδη επισκεφθεί η διαδικασία. Σε κάθε αναδομική κλήση ένας νέος κόμβος προστίθεται σε αυτή τη λίστα.

Παράμετρος Συσσώρευσης

Η λίστα που αποθηκεύει τους κόμβους ονομάζεται και **παράμετρος συσσώρευσης (accumulative parameter)** γιατί συσσωρεύει πληροφορία σε κάθε αναδρομική κλήση. Μια παράμετρος συσσώρευσης αρχικοποιείται με μία τιμή, όπως στο παραπάνω παράδειγμα τον αρχικό κόμβο InitialNode.

Η παράμετρος συσσώρευσης είναι μια συνηθισμένη πρακτική στην Prolog που βοηθά στο να μπει η αναδρομική κλήση στο τέλος του ορισμού. Για παράδειγμα, ο ορισμός του αριθμού στοιχείων σε μία λίστα μπορεί να γραφεί και ως:

```
list_length(List, R) :-
  list_length(List, 0, R).
```

όπου το μηδέν είναι η αρχική τιμή της παραμέτρου συσσώρευσης . Το κατηγόρημα list_length/3 ορίζεται ως εξής:

```
list_length([], L, L).
list_length([H | T], SoFar, L) :-
  NewLength is SoFar + 1,!,
  list_length(T, NewLength, L).
```

Η συνθήκη τερματισμού αναδεικνύει ότι το τελικό αποτέλεσμα, δηλαδή το μήκος της λίστας, που είναι όσο η συσσωρευμένη αξία γίνεται μέσω της εκτέλεσης του προγράμματος. Η διαφορά μεταξύ αυτής της

κωδικοποίησης του list_length και αυτής που παρουσιάστηκε σε προηγούμενο κεφάλαιο, είναι ότι η αναδρομή εδώ είναι πιο αποτελεσματική ως προς τη μνήμη που απαιτείται για την εκτέλεση. Η εξοικονόμηση μνήμης ονομάζεται **βελτιστοποίηση της τελευταίας κλήσης (last call optimisation)**. Εφόσον η αναδρομική κλήση είναι η τελευταία κλήση στον ορισμό και όλες οι μεταβλητές μέχρι εκείνο το σημείο είναι δεσμευμένες, η μνήμη που έχει χρησιμοποιηθεί μπορεί να επαναχρησιμοποιηθεί. Αυτό έχει σαν αποτέλεσμα η μνήμη να είναι σταθερή ανεξάρτητα από τον αριθμό των αναδρομικών κλήσεων.

Από την άλλη πλευρά όμως, η παραπάνω εκδοχή απαιτεί μια επιπλέον παράμετρο. Παρατηρήστε ότι η επιπλέον παράμετρος παραμένει αδέσμευτη μεταβλητή και δεσμεύεται με συγκεκριμένη τιμή μόνο στη συνθήκη τερματισμού. Θα μπορούσαμε να την παρομοιάσουμε ως μια "τεμπέλικη μεταβλητή" (lazy variable), επειδή βρίσκεται σε αναμονή για τη δέσμευση της κατά τη διάρκεια όλων των αναδρομικών κλήσεων.

Με βάση τα παραπάνω, κάποιος θα μπορούσε να ξαναγράψει τον ορισμό του path_loopcheck/4 με μια παράμετρο συσσώρευσης που επιστρέφει το τελικό μονοπάτι σαν τη λίστα με όλους τους κόμβους που έχουν επισκεφθεί:

```
path_loopcheck _alt(InitialNode, FinalNode, Route) :-
  path_loopcheck _alt(InitialNode, FinalNode, [InitialNode], Route).

path_loopcheck _alt(Node, FinalNode, Route, [FinalNode|Route]):-
  next(Node, FinalNode).

path_loopcheck _alt(Node, FinalNode, VisitedNodes, Route) :-
  next(Node, SomeNextNode),
  not member(SomeNextNode, VisitedNodes),
  path_loopcheck _alt(SomeNextNode, FinalNode,
             [SomeNextNode|VisitedNodes], Route).
```

Το μόνο μειονέκτημα αυτής της εναλλακτικής λύσης είναι ότι η διαδρομή εμφανίζεται με αντίστροφη σειρά, από τον τελικό προς τον αρχικό κόμβο:

```
?- path_loopcheck _alt(a, f, Route).
Route = [f,c,b,a] ;
...
```

κάτι που θα μπορούσε εύκολα να διευθετηθεί μέσω της αλλαγής του ορισμού:

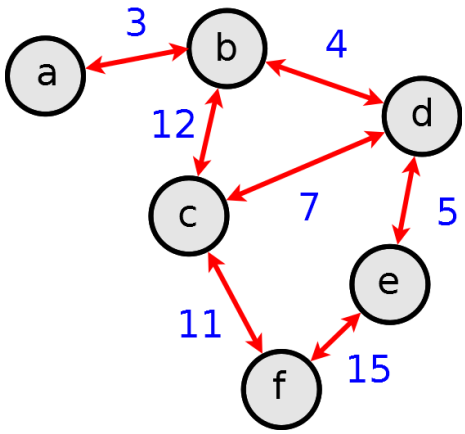
```
path_loopcheck _alt(Node, FinalNode, Route, FinalRoute):-
  next(Node, FinalNode),
  reverse([FinalNode|Route], FinalRoute).
```

Στους παραπάνω ορισμούς πρέπει να σημειώσουμε την ύπαρξη παρόμοιων κατηγορημάτων αλλά με διαφορετικό αριθμό ορισμάτων, για παράδειγμα το path_loopcheck/3 και το path_loopcheck/4, και το list_length/2 και list_length/3. Παρόλο που τα ονόματα των κατηγορημάτων είναι τα ίδια, για την Prolog πρόκειται για δύο τελείως διαφορετικά κατηγορήματα.

Εύρεση διαδρομής και κόστους

Κάποιοι γράφοι εμπεριέχουν κόστος μεταξύ κόμβων (Σχήμα 11.7). Η αναπαράσταση του γράφουν αλλάζει, έτσι ώστε να δηλώνεται το βάρος (κόστος) μεταξύ των κόμβων ως εξής:

```
link(a,b,3).
link(b,d,4).
link(b,c,12).
link(d,c,7).
...
```



Σχήμα 11.7: *Ένας γράφος με βάρη*

Το συνολικό κόστος μιας διαδρομής Cost υπολογίζεται ταυτόχρονα με την εύρεσή της:

```
path_weight(InitialNode, FinalNode, Route, Cost):-
  path_weight(InitialNode, FinalNode, [InitialNode], Route, Cost).

path_weight(Node, FinalNode, Route, [FinalNode|Route], Cost) :-
  next(Node, FinalNode, Cost).

path_weight(Node, FinalNode, VisitedNodes, Route, TotalCost):-
  next(Node, NextNode, Cost),
  not member(SomeNextNode, VisitedNodes),
  path_weight(NextNode, FinalNode, [NextNode|VisitedNodes], Route,
RestCost),
  TotalCost is RestCost + Cost.
```

Και πάλι, μια παράμετρος συσσώρευσης μπορεί να χρησιμοποιηθεί για το κόστος.

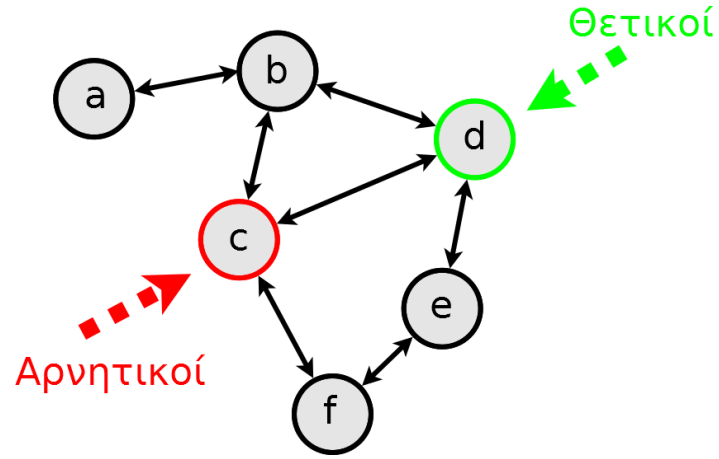
Εύρεση διαδρομής με κριτήρια

Υπάρχουν περιπτώσεις όπου η παραγόμενη διαδρομή πρέπει να ικανοποιεί κάποια κριτήρια. Για παράδειγμα, η διαδρομή πρέπει να περάσει από συγκεκριμένους κόμβους (positive constraint) ή/και δεν πρέπει να περάσει από κάποιους άλλους κόμβους (negative constraint), όπως φαίνεται στο Σχήμα 11.8. Έκφραση των κριτηρίων γίνεται με κατηγορήματα. Για παράδειγμα, αν η διαδρομή πρέπει να περιέχει το d αλλά δεν πρέπει να περιέχει το c, οι ορισμοί είναι:

```
negative_constraint(Path):-
  member(c, Path).
```

και

```
positive_constraint(Path):-
  member(d, Path).
```



Σχήμα 11.8: *Ένας γράφος με θετικούς και αρνητικούς περιορισμούς*

Ο ορισμός του κατηγορήματος εύρεσης διαδρομής γίνεται:

```
path_constraint(InitialNode, FinalNode, Route) :-
  path_constraint(InitialNode, FinalNode, [InitialNode], Route).

path_constraint(Node, FinalNode, Route, [FinalNode|Route]):-
  next(Node, FinalNode),
  positive_constraint([FinalNode|Route]).

path_constraint(Node, FinalNode, VisitedNodes, Route):-
  next(Node, SomeNextNode),
  not member(SomeNextNode, VisitedNodes),
  not negative_constraint([SomeNextNode|VisitedNodes]),
  path_constraint(SomeNextNode, FinalNode,
        [SomeNextNode|VisitedNodes], Route).
```

Έτσι, η απάντηση περιέχει διαδρομές που ικανοποιούν αυτούς τους περιορισμούς:

```
?- path_constraint(a, f, Route).
Route = [a,b,d,e,f]
```

11.4 Επίλυση Προβλημάτων ΤΝ με Τυφλή Αναζήτηση

Η επίλυση ενός προβλήματος αποτελεί θεμελιώδη διαδικασία στην Τεχνητή Νοημοσύνη. Σε αυτό το βιβλίο, δεν είναι σκοπός μας να αναλύσουμε με λεπτομέρεια πως αυτό συμβαίνει. Περισσότερες λεπτομέρειες αναφέρονται εκτενώς σε βιβλία Τεχνητής Νοημοσύνης, όπως το βιβλίο "Τεχνητή Νοημοσύνη" των Ι. Βλαχάβα, Π. Κεφαλά, Ν. Βασιλειάδη, Φ. Κόκκορα, Η. Σακελλαρίου. Εδώ θα παρουσιάσουμε τρεις βασικούς αλγορίθμους επίλυσης προβλημάτων βασισμένα σε καταστάσεις (state-based problems) για να δείξουμε την ομοιότητα που παρουσιάζει η διαδικασία επίλυσης με αυτή της εύρεσης μίας διαδρομής σε γράφο.

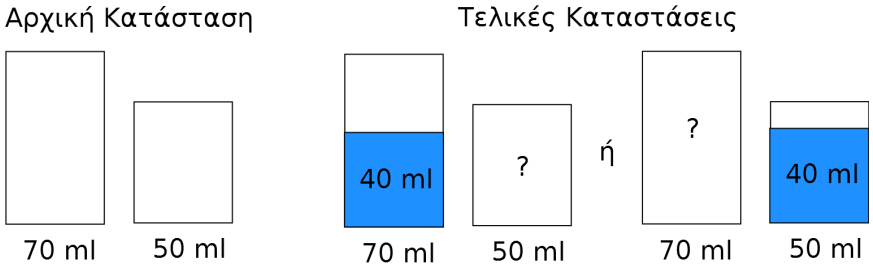
Η έννοια του Προβλήματος

Κατάσταση (state) ενός κόσμου είναι ένα στιγμιότυπο (instance) μίας συγκεκριμένης χρονικής στιγμής κατά την εξέλιξη του κόσμου. Τα προβλήματα βασισμένα σε καταστάσεις ορίζονται ως:

- μια αρχική κατάσταση (initial state)
- μια ή περισσότερες τελικές καταστάσεις (goal states)
- ενέργειες που μπορούν να αλλάξουν τις καταστάσεις του προβλήματος (operators).

Λύση σε ένα πρόβλημα αποτελεί η σειρά των ενεργειών που θα γίνουν για να γεφυρώσουν την αρχική με μία από τις τελικές καταστάσεις. Για λόγους απλούστευσης εδώ θα θεωρήσουμε λύση τη σειρά των καταστάσεων από την αρχική μέχρι την τελική.

Για παράδειγμα, στο πρόβλημα των ποτηριών, δύο ποτήρια A και B χωρίς υποδιαιρέσεις χωράνε ακριβώς το A 70 ml και το B 50 ml αντίστοιχα. Τα ποτήρια μπορούν να γεμίζουν μέχρι το χείλος από μία βρύση και να αδειάζουν είτε το ένα μέσα στο άλλο ή στο νεροχύτη, με τελικό στόχο να υπάρχουν στο τέλος ακριβώς 40 ml σε κάποιο από τα δύο. Λύση στο πρόβλημα αποτελεί η σειρά των ενεργειών που θα γίνουν ώστε το ένα από τα δύο ποτήρια να έχει τελικά 40 ml (Σχήμα 11.9).



Σχήμα 11.9: *Η αρχική και οι τελικές καταστάσεις του προβλήματος των ποτηριών*

Η χωρητικότητα των δύο ποτηριών A και B είναι:

```
glassA(70).
glassB(50).
```

Θα περιγράψουμε μια κατάσταση με τον όρο state(Description,Path), όπου:

- Description είναι ένας όρος που περιγράφει το στιγμιότυπο, και
- Path είναι η διαδρομή μέσα σε λίστα που έχει ακολουθηθεί μέχρι αυτή την κατάσταση

Η παράμετρος Path είναι χρήσιμη γιατί αποθηκεύει τη μέχρι τώρα διαδρομή και στην περίπτωση της τελικής κατάστασης τη λύση του προβλήματος. Δεν είναι άμεσα διαισθητικό αλλά η ανάγκη της θα φανεί παρακάτω. Συνεπώς, η αρχική κατάσταση είναι η:

```
initial_state(state((0,0),[])).
```

και οι τελικές καταστάσεις:

```
final_state(state((_,40),_Path)).
final_state(state((40,_),_Path)).
```

Οι ενέργειες που μπορούν να γίνουν σε οποιαδήποτε κατάσταση (εφόσον ικανοποιούνται ορισμένες συνθήκες) είναι συνολικά οκτώ:

- άδειασε όλο το περιεχόμενο του B στο A
- άδειασε όλο το περιεχόμενο του A στο B
- άδειασε μέρος του B στο A για να γεμίσει
- άδειασε μέρος του A στο B για να γεμίσει
- γέμισε το A από τη βρύση
- γέμισε το B από τη βρύση
- άδειασε το A
- άδειασε το B

Για παράδειγμα η ενέργεια "άδειασε όλο το περιεχόμενο του B στο A" περιγράφεται ως εξής:

```
operator(state((V1,V2),Path),state((Vsum,0),[(V1,V2)|Path])) :-
  V2 > 0,
  Vsum is V1 + V2,
  glassA(G1),
  Vsum =< G1.
```

Στα δύο ορίσματα αναφέρονται:

- η τρέχουσα κατάσταση state((V1,V2),Path), και
- η επόμενη κατάσταση state((Vsum,0),[(V1,V2)|Path])

Άλλες ενδεικτικές ενέργειες είναι η "άδειασε μέρος του B στο A για να γεμίσει":

```
operator(state((V1,V2),Path),state((G1,Vdiff),[(V1,V2)|Path])) :-
  V2 > 0,
  V1 >= 0,
  glassA(G1),
  Vdiff is V2 - ( G1 - V1 ),
  Vdiff > 0.
```

η ενέργεια "γέμισε το A από τη βρύση":

```
operator(state((V1,V2),Path),state((G1,V2),[(V1,V2)|Path])) :-
  glassA(G1), V1\=G1.
```

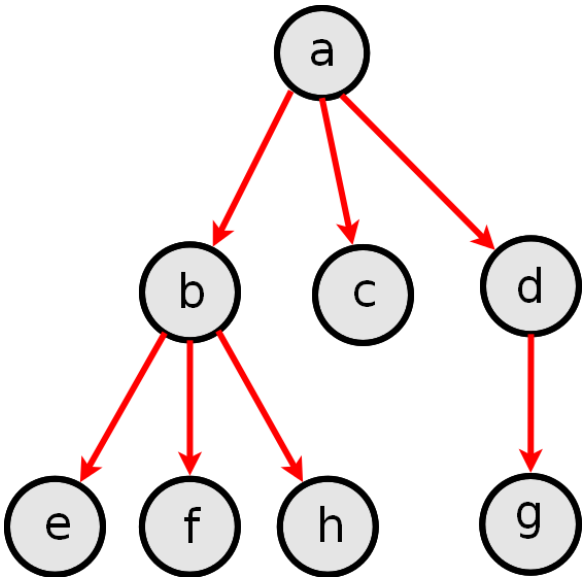
και τέλος η ενέργεια "άδειασε το A":

```
operator(state((V1,V2),Path),state((0,V2),[(V1,V2)|Path])).
```

Όμοια υλοποιούνται και οι υπόλοιπες αντίστοιχες τέσσερις ενέργειες.

Η εύρεση λύσης ενός προβλήματος ως εύρεση διαδρομής σε γράφο

Μετά τον ορισμό ενός προβλήματος ακολουθεί η προσπάθεια επίλυσής του. Ξεκινώντας από την αρχική κατάσταση μπορούμε να δημιουργήσουμε τις επόμενες καταστάσεις που ακολουθούν. Για παράδειγμα από την (0,0) μπορούν να δημιουργηθούν δύο επόμενες καταστάσεις η (70,0) και η (0,50) γεμίζοντας το A ή B ποτήρι αντίστοιχα. Μετά, από κάθε μια από αυτές τις νέες καταστάσεις με άλλες ενέργειες δημιουργούνται άλλες νέες καταστάσεις και αυτή η διαδικασία συνεχίζεται έως ότου συναντήσει κάποιος μια από τις τελικές καταστάσεις. Η διαδικασία δημιουργίας νέων καταστάσεων ονομάζεται επέκταση (expansion) μιας τρέχουσας κατάστασης. Με τη διαρκή επέκταση καταστάσεων, αυτό που δημιουργείται είναι ένα δένδρο (μη κυκλικός γράφος) που ονομάζεται και **δένδρο αναζήτησης (search tree)**. Κι αυτό γιατί σε αυτό το δένδρο αναζητείται η λύση ενός προβλήματος, ως η διαδρομή που ενώνει την αρχική και μία τελική κατάσταση. Στο Σχήμα 11.10 φαίνεται ένα από δένδρο αναζήτησης με αρχική κατάσταση τη και a τελική την g. Η λύση σε αυτό το δένδρο είναι η διαδρομή a-d-g.



Σχήμα 11.10: *Ένα δένδρο αναζήτησης με όλες τις καταστάσεις ενός προβλήματος με αρχική κατάσταση την a και τελική κατάσταση τη g.*

Το Prolog κατηγόρημα για την επέκταση μιας κατάστασης και δημιουργία των επομένων καταστάσεων πρέπει να έχει την εξής αφαιρετική μορφή:

expand(*Τρέχουσα Κατάσταση,Λίστα Επόμενων Καταστάσεων*):-

findall(*Νέα Κατάσταση*,

τέτοια ώστε

operator(*Τρέχουσα Κατάσταση, Επόμενη Κατάσταση*),

και η επόμενη κατάσταση να μην ανήκει στη μέχρι τώρα διαδρομή,

Λίστα Επόμενων Καταστάσεων).

Το κατηγόρημα expand/2 εφαρμόζει όλους τους πιθανούς τελεστές/ενέργειες που μπορεί να γίνουν στην τρέχουσα κατάσταση και βάζει όλες τις νέες καταστάσεις σε μία λίστα. Υπάρχει ένας επιπλέον έλεγχος ο οποίος αποφεύγει να βάλει στη λίστα καταστάσεις που υπάρχουν ήδη στη διαδρομή που έχει δημιουργηθεί μέχρι τώρα.

Ο πλήρης ορισμός του κατηγορήματος expand/2 που εφαρμόζεται ανεξάρτητα από το προς επίλυση πρόβλημα είναι:

```
expand(state(Description,Path),NextStates):-
  findall(state(NewDescription,[Description|Path]),
   (
    operator(
     state(Description,Path),
      state(NewDescription,[Description|Path])
            ),
     not(member(NewDescription,[Description|Path]))
    ),
    NextStates).
```

Το κατηγόρημα expand/2 παίζει ακριβώς το ρόλο του κατηγορήματος next/2 στους γράφους.

Άρα λοιπόν, η επίλυση ενός προβλήματος ανάγεται σε εύρεση διαδρομής σε ένα γράφο. Το θέμα είναι με ποιον τρόπο θα γίνει η αναζήτηση, δηλαδή με ποιον αλγόριθμο θα δημιουργήσουμε τις νέες καταστάσεις. Αυτή η επιλογή είναι κρίσιμη γιατί μπορεί να συντελέσει στην εύρεση ή όχι της λύσης, στην εύρεση μιας οποιαδήποτε λύσης αντί της "καλύτερης", στο χρόνο επίλυσης κλπ. Όλα αυτά όμως είναι αντικείμενο της Τεχνητής Νοημοσύνης και δε θα μας απασχολήσουν σε αυτό το βιβλίο.

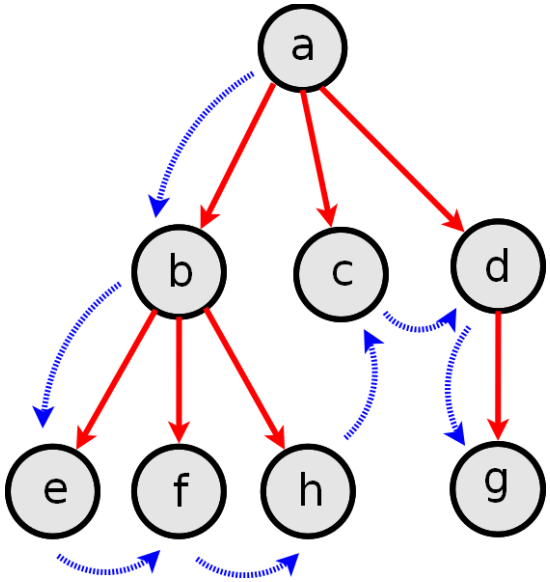
Εύρεση λύσης με αναζήτηση Πρώτα σε Βάθος

Η αναζήτησης της λύσης/διαδρομής απαιτεί μια συγκεκριμένη μεθοδολογία/αλγόριθμο που να καθορίζει ποιο είναι το επόμενο βήμα. Στην επίλυση προβλημάτων ένας τέτοιος αλγόριθμος πρέπει να καθορίσει προς τα που θα κινηθεί η αναζήτηση. Για παράδειγμα, αν επεκταθεί η κατάσταση a θα δώσει τις b,c και d. Σε ποια από τις νέες τρεις θα συνεχίσει η αναζήτηση; Αν υποθέσουμε ότι ανάμεσα σε καταστάσεις του ίδιου επιπέδου/βάθους διαλέγουμε εκείνη που βρίσκεται πιο αριστερά στο δένδρο, δηλαδή την b, αυτή θα δημιουργήσει άλλες τρεις f, e και h οι οποίες θα προστεθούν στις προηγούμενες εναλλακτικές. Έτσι τώρα όλες οι υποψήφιες καταστάσεις είναι οι c, d, f, e και h. Ανάμεσα σε καταστάσεις διαφορετικού επιπέδου/βάθους ποια κατάσταση θα επιλεγεί για επέκταση;

Η επιλογή καθορίζει και τον αλγόριθμο. Για παράδειγμα:

- αν επιλεγεί προς επέκταση η βαθύτερη στο δένδρο και ανάμεσα στις ισοβαθείς η αριστερότερη, τότε διεξάγεται Αναζήτηση Πρώτα σε Βάθος (Depth-First Search)
- αν επιλεγεί προς επέκταση η ρηχότερη στο δένδρο και ανάμεσα στις πιο ρηχές η αριστερότερη, τότε διεξάγεται Αναζήτηση Πρώτα σε Πλάτος (Breadth-First Search)

Όλες οι υποψήφιες προς επέκταση καταστάσεις αποτελούν μία λίστα που ονομάζεται Ανοιχτή Λίστα (Open List) ή Σύνορο της Αναζήτησης (Frontier). Ο αλγόριθμος Αναζήτηση Πρώτα σε Βάθος επιλέγει την πρώτη κατάσταση σε αυτή τη λίστα. Η Ανοιχτή Λίστα διαμορφώνεται έτσι ώστε οι νέες καταστάσεις να προστίθενται στο τέλος της λίστας.



Σχήμα 11.11: *Η σειρά που επισκέπτεται ο Αλγόριθμος Πρώτα σε Βάθος τις καταστάσεις στο δένδρο αναζήτησης*

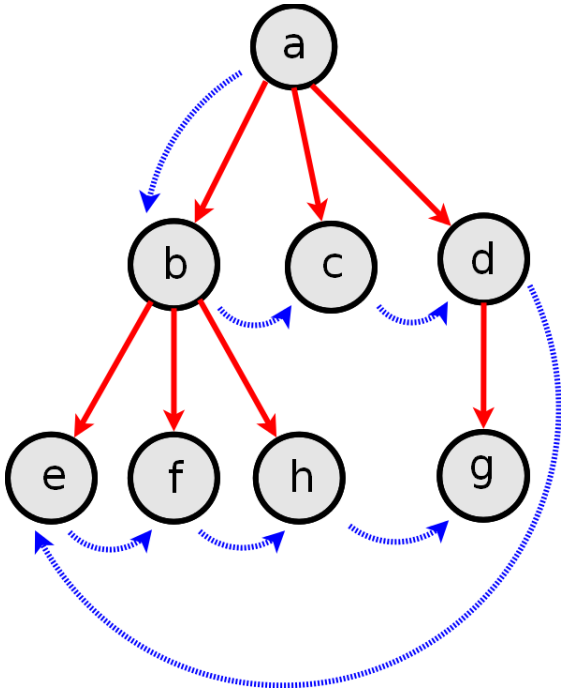
Το βίντεο που δείχνει τη σειρά με την οποία ο αλγόριθμος επισκέπτεται ένα δένδρο αναζήτησης (Σχήμα 11.11) είναι Animation of Depth-First Search ή εναλλακτικά αυτό.

Ο αλγόριθμος Αναζήτηση Πρώτα σε Βάθος υλοποιείται σε Prolog ως εξής:

```
dfs:-
  initial_state(S),
  dfs([S],[],Solution),
  write(Solution),nl.

dfs([State|_],State):-
  final_state(State).

dfs([State|RestOpen],Solution):-
  expand(State,NextStates),
  append(NextStates,RestOpen,NewOpenList),
  !,
  dfs(NewOpenList,Solution).
```

Εύρεση λύσης με αναζήτηση Πρώτα σε Πλάτος

Ο αλγόριθμος Αναζήτηση Πρώτα σε Πλάτος επιλέγει πάλι την πρώτη κατάσταση σε αυτή τη λίστα. Αλλά σε αντίθεση με τα παραπάνω, η Ανοιχτή Λίστα διαμορφώνεται έτσι ώστε οι νέες καταστάσεις να προστίθενται στην αρχή της λίστας.



Σχήμα 11.12: *Η σειρά που επισκέπτεται ο Αλγόριθμος Πρώτα σε Πλάτος τις καταστάσεις στο δένδρο αναζήτησης*

Το βίντεο που δείχνει τη σειρά με την οποία ο αλγόριθμος επισκέπτεται ένα δένδρο αναζήτησης (Σχήμα 11.12) είναι Animation of Breath-First Search.

Ο αλγόριθμος Αναζήτηση Πρώτα σε Πλάτος υλοποιείται σε Prolog ως εξής:

```
bfs:-
  initial_state(S),
  bfs([S],[],Solution),
  write(Solution),nl

bfs([State|_],State):-
   final_state(State).

bfs([State|RestOpen],Solution):-
  expand(State,NextStates),
  append(RestOpen, NextStates,NewOpenList),
  !,
  bfs(NewOpenList,Solution).
```

Είναι εύκολο να παρατηρήσει κανείς ότι η διαφορά των δύο αλγορίθμων έγκειται στη σειρά των ορισμάτων του append/3.

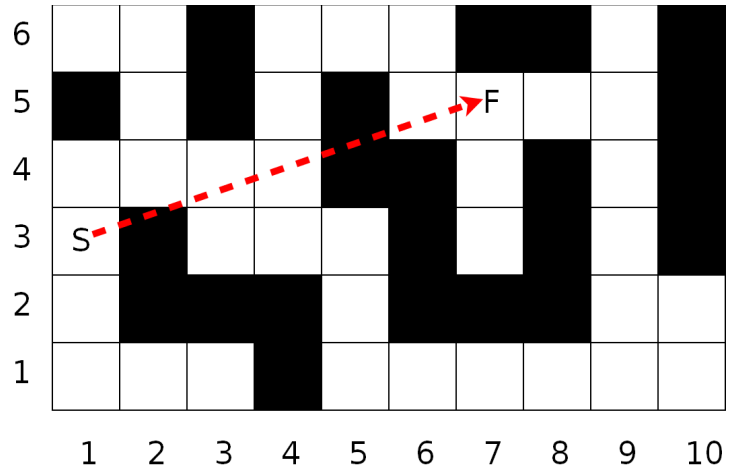
11.5 Προχωρημένα Θέματα: Επίλυση Προβλημάτων ΤΝ με Ευριστική Αναζήτηση

Οι δύο αλγόριθμοι που παρουσιάστηκαν παραπάνω ανήκουν στην κατηγορία των Αλγορίθμων **Τυφλής Αναζήτησης (Blind Search)**, γιατί δεν έχουν κάποια ένδειξη ποια διαδρομή πρέπει να ακολουθήσουνε για να βρεθούν πιο κοντά στην τελική κατάσταση. Υπάρχουν όμως προβλήματα στα οποία οι καταστάσεις μπορούν να ταξινομηθούν σε "μάλλον καλύτερες" και "μάλλον χειρότερες" με βάση την "απόστασή" τους από την τελική κατάσταση. Για παράδειγμα στο πρόβλημα εύρεσης διαδρομής σε ένα λαβύρινθο στο Σχήμα 11.13 η κατάσταση του να βρεθεί η αναζήτηση στο (2,4) μοιάζει καλύτερη από τη (1,1). Άρα, αν στην ανοιχτή λίστα υπήρχαν αυτές οι δύο καταστάσεις, θα ήταν πιο υποσχόμενο να διαλέξουμε τη φαινομενικά καλύτερη, δηλαδή τη (2,4), εφόσον η απόσταση της από την τελική (αγνοώντας τα εμπόδια) είναι μικρότερη από αυτήν της (1,1).

Οι αλγόριθμοι που βασίζονται σε μία τέτοια ταξινόμηση καταστάσεων ονομάζονται Αλγόριθμοι **Ευριστικής Αναζήτησης (Heuristic Search)**. Η μέθοδος (συνήθως μια αριθμητική παράσταση που δίνει μία τιμή) βάση της οποίας γίνεται η αξιολόγηση κάθε κατάστασης ονομάζεται **Ευριστική Συνάρτηση (Heuristic Function)**. Ένας τέτοιος είναι ο αλγόριθμος **Αναζήτησης Πρώτα στο Καλύτερο (Best-First Search)**.

Η έννοια του Προβλήματος με Ευριστικό Μηχανισμό

Έστω το πρόβλημα αναζήτησης διαδρομής σε ένα λαβύρινθο με εμπόδια, όπως φαίνεται στο Σχήμα 11.13 .



Σχήμα 11.13: *Το πρόβλημα εύρεσης διαδρομής σε λαβύρινθο με αρχική κατάσταση την S και τελική κατάσταση την F.*

Τα εμπόδια μπορούν να αναπαρασταθούν ως γεγονότα της μορφής:

```
blocked_square((1,5)).
blocked_square((2,2)).
blocked_square((2,3)).
blocked_square((3,2)).
...
```

Οι αρχική και η τελική κατάσταση:

```
initial_state(state((1,3),[])).
final_state(state((7,5),_PATH)).
```

Οι ενέργειες/τελεστές είναι τέσσερις (μπορώ να κινηθώ ένα τετράγωνο προς τα δεξιά, αριστερά, πάνω και κάτω):

```
operator(state(Parent,Path),state(Child,[Parent|Path])):-
  can_go(Parent,Child),
  not(blocked_square(Child)),
  valid(Child).

can_go((X,Y),(NX,Y)):- NX is X + 1.
can_go((X,Y),(NX,Y)):- NX is X - 1.
can_go((X,Y),(X,NY)):- NY is Y + 1.
can_go((X,Y),(X,NY)):- NY is Y - 1.

valid((X,Y)):-
  X>0,Y>0,dimension(LX,LY),X =<LX,Y=<LY.
```

όπου dimension/2 δίνει τις διαστάσεις του λαβύρινθου:

```
dimension(10,6).
```

Η ευριστική συνάρτηση που χρησιμοποιείται σε αυτό το πρόβλημα είναι η λεγόμενη απόσταση Manhattan που υπολογίζει την απόσταση από την τελική ως άθροισμα απόλυτων τιμών των πλευρών:

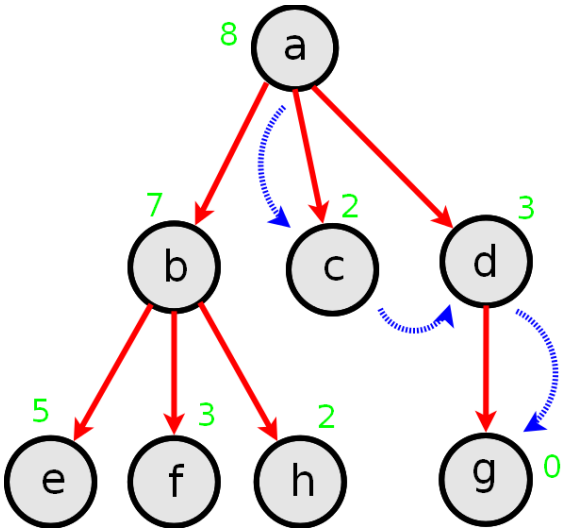
```
heuristic_Func(state((X,Y),_),HV):-
  final_state(state((XF,YF),_)),
```

```
HV is abs(X-XF) + abs(Y-YF).
```

Εναλλακτικά χρησιμοποιείται ως ευριστική συνάρτηση και η Ευκλείδειος απόσταση που υπολογίζει την απόσταση από την τελική ως τη ρίζα του αθροίσματος των πλευρών.

Εύρεση λύσης με αναζήτηση Πρώτα στο Καλύτερο

Ο αλγόριθμος **Πρώτα στο Καλύτερο (Best-First Search)** χρησιμοποιεί την ευριστική συνάρτηση για να αξιολογήσει την κάθε κατάσταση και επισκέπτεται αυτήν με την μικρότερη τιμή. Έτσι στο Σχήμα 11.14, η σειρά με την οποία θα επεκτείνει τις καταστάσεις του δένδρου αναζήτησης είναι η a, c, d και g.



Σχήμα 11.14: *Η σειρά που επισκέπτεται ο Αλγόριθμος Πρώτα στο Καλύτερο τις καταστάσεις στο δένδρο αναζήτησης*

Για να είναι δυνατή η ταξινόμηση των καταστάσεων, η ανοιχτή λίστα έχει δυάδες της μορφής (Ευριστική Τιμή, Κατάσταση). Ο αλγόριθμος υλοποιείται στην Prolog ως εξής:

```
bestfs:-
  initial_state(S),
  evaluateStates([S],EVSTATES),
  bestfs(EVSTATES,Solution),
  write(Solution),nl.

bestfs([(_,State)|_],State):-
  final_state(State).

bestfs([(_,State)|OPEN],Solution):-
  expand(State,NextStates),
  evaluateStates(NextStates,EV_States),
  sort(EV_States,NextSorted),
  merge(NextSorted,OPEN,NewOpen),
  !,
  bestfs(NewOpen,Solution).
```

όπου το κατηγόρημα evaluateStates/2 καλεί το heuristic_Func/2 για αναθέσει ευριστικές τιμές σε κάθε νέα κατάσταση πριν αυτές προστεθούν στην ανοιχτή λίστα:

```
evaluateStates(States,Evaluated):-
  findall((HV,State),
          (member(State,States), heuristic_Func(State,HV)),
          Evaluated).
```

Τα ενσωματωμένα κατηγορήματα sort/2 και merge/2 ταξινομούν μια λίστα και ενώνουν δύο ταξινομημένες λίστες αντίστοιχα.

Βιβλιογραφία

Οι αλγόριθμοι αναζήτησης για επίλυση προβλημάτων είναι από τα βασικά θέματα που αναφέρονται σε οποιοδήποτε βιβλίο σχετικό με Τεχνητή Νοημοσύνη (Βλαχάβας I. et al., 2011), (Russel & Norvig, 2009), (Luger, 2008). Η πρώτη προσέγγιση των αλγορίθμων με την γλώσσα Prolog έγινε με εκτενή τρόπο στο κλασσικό βιβλίο του Bratko και τις μετέπειτα εκδόσεις του. Οι αλγόριθμοι στο κεφάλαιο αυτό έχουν κάποιες διαφορές από του (Bratko, 2011), κυρίως βέβαια διότι η ανάπτυξή τους παρουσιάζεται σταδιακά, όπως παραδείγματος χάριν συμβαίνει με το παράδειγμα της αναζήτησης διαδρομής σε γράφο.

Βλαχάβας, I. και Κεφαλάς, Π. και Βασιλειάδης, N. και Κόκκορας, Φ. και Σακελλαρίου Η. (2011). *Τεχνητή Νοημοσύνη*. Γ' Έκδοση. ISBN: 978-960-8396-64-7. Εκδόσεις Πανεπιστημίου Μακεδονίας.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. 3rd edition. Prentice Hall.

Luger G.F. (2008). *Artificial Intelligence: Structures and Strategies for Complex Problem-Solving*. 6th edition. Addison Wesley Longman.

Bratko, I. (2011). *Prolog programming for artificial intelligence*. 4th edition. Pearson Education Canada

Άλυτες Ασκήσεις

11.1 Για να συμπληρωθεί ο ορισμός του κατηγορήματος nsort/2, να γραφεί ο ορισμός του κατηγορήματος is_sorted/2 που εξετάζει αν μία λίστα από αριθμούς είναι διατεταγμένη κατά φθίνουσα σειρά. Για παράδειγμα:

```
?- is_sorted([10,8,5,3,2]).
yes
?-is_sorted([10,5,2,8,3]).
no
```

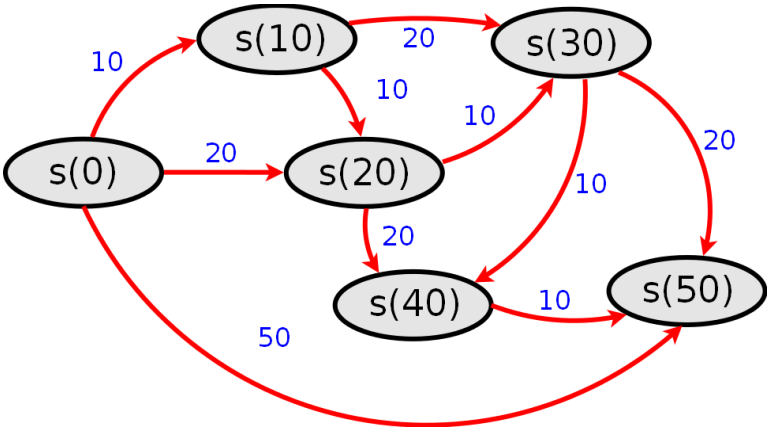
11.2 Για να συμπληρωθεί ο ορισμός του κατηγορήματος isort/2, να γραφεί ο ορισμός του κατηγορήματος insert_sorted/3, το οποίο εισάγει ένα αριθμό σε μία ταξινομημένη λίστα κατά φθίνουσα σειρά στη σωστή του θέση, δηλαδή ανάμεσα σε ένα μεγαλύτερο και ένα μικρότερο αριθμό. Για παράδειγμα:

```
?-insert_sorted(3,[10,8,5,2],L).
L=[10,8,5,3,2]
?-insert_sorted(10,[8,5,3,2],L).
L=[10,8,5,3,2]
?-insert_sorted(2,[10,8,5,3],L).
L=[10,8,5,3,2]
```

11.3 Για να συμπληρωθεί ο ορισμός του κατηγορήματος quicksort/2, να γραφεί ο ορισμός του κατηγορήματος partition/4, το οποίο χωρίζει μία λίστα L με βάση έναν αριθμό X, σε δύο άλλες λίστες, την Big που περιέχει όλους τους αριθμούς μεγαλύτερους από το X και την Small που περιέχει όλους τους αριθμούς μικρότερους από το X. Για παράδειγμα:

```
?-partition(4,[10,7,2,1,8,3,5],Big,Small).
Big=[10,7,8,5]
Small=[2,1,3]
```

11.4 Ένα μηχάνημα αυτόματης πώλησης εισιτηρίων σε αστικά λεωφορεία μοντελοποιείται από την ακόλουθη μηχανή πεπερασμένων καταστάσεων. Η τιμή του εισιτηρίου είναι 50 λεπτά, και το μηχάνημα μπορεί να δεχθεί νομίσματα των 10, 20 και 50 λεπτών. Ο αυτόματος πωλητής επιστρέφει εισιτήριο μόνο αν το ακριβές αντίτιμο εισάγεται στη μηχανή. Η μηχανή πεπερασμένων καταστάσεων που περιγράφει την λειτουργία του αυτόματου πωλητή παρουσιάζεται στο Σχήμα 11.15:



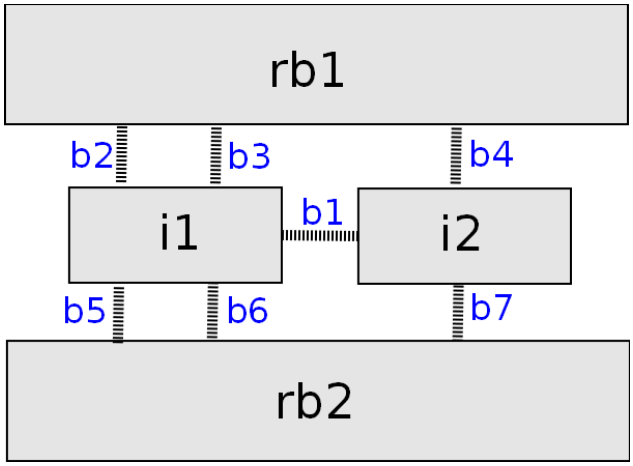
Σχήμα 11.15: *Η μηχανή πεπερασμένων καταστάσεων που περιγράφει την λειτουργία ενός αυτόματου πωλητή*

(α) Να υλοποιήσετε ένα κατηγόρημα transitions/3 το οποίο περιγράφει τις μεταβάσεις του παραπάνω αυτομάτου. Το κατηγόρημα θα έχει σαν ορίσματα δύο καταστάσεις και το νόμισμα που απαιτείται για την μετάβαση από την πρώτη κατάσταση στη δεύτερη.

(β) Να υλοποιήσετε το κατηγόρημα coins_to_insert/3, το οποίο δεδομένης μιας αρχικής κατάστασης και μιας επιθυμητής στην οποία θα "πάει" το αυτόματο,να επιστρέφει σε λίστα τα νομίσματα που απαιτούνται.

(γ) Μπορείτε να υπολογίσετε πόσοι διαφορετικοί τρόποι υπάρχουν (συνδυασμοί νομισμάτων) για να αγοράσει κάποιος εισιτήριο? Θα πρέπει να σημειωθεί ότι δύο ακολουθίες όπου τα νομίσματα εμφανίζονται διαφορετικά, θεωρούνται διαφορετικές μεταξύ τους.

11.5 Η πόλη του Konigsberg καταλαμβάνει τις δύο όχθες (rb1, rb2) και τα δύο νησιά (i1, i2) του ποταμού Pregel και τα προηγούμενα συνδέονται με γέφυρες (b1, b2, ...b7), όπως φαίνεται σχηματικά στο Σχήμα 11.16.



Σχήμα 11.16: *Η πόλη του Konigsberg*

(α) Να ορίσετε ένα κατηγόρημα connection/3 το οποίο πετυχαίνει όταν τα πρώτα δύο ορίσματα του είναι τοποθεσίες, δηλαδή όχθες ή νησιά και το τρίο μια γέφυρα που τα ενώνει. Για παράδειγμα:

```
?- connection(rb1,i2,Bridge).
Bridge = b4
yes
?- connection(rb1,i1,Bridge).
Bridge = b2 ;
Bridge = b3 ;
No
?- connection(rb1,rb2,Bridge).
```

```
No
```

(β) Το κύριο τουριστικό αξιοθέατο της πόλης είναι οι γέφυρες, έτσι οι περιηγητές προσπαθούν να σχεδιάσουν διαδρομές που περιλαμβάνουν διέλευση από αυτές. Να ορίσετε ένα κατηγόρημα walk/3 το οποίο πετυχαίνει όταν τα δύο πρώτα του ορίσματα είναι τοποθεσίες και το τρίτο μια λίστα από γέφυρες τις οποίες θα πρέπει να διέλθει ο περιηγητής για να φτάσει από την πρώτη τοποθεσία στην δεύτερη. Θα πρέπει σημειωθεί ότι ο περιηγητής θα πρέπει να διέλθει κάθε γέφυρα μόνο μια φορά, ασχέτως πόσες φορές θα επισκεφτεί μια τοποθεσία (όχθη ή νησί). Για παράδειγμα:

```
?- walk(rb1,i1,L).
L = [b2] ;
L = [b3] ;
L = [b2, b1, b4, b3]
..(υπάρχουν και πολλές άλλες λύσεις)...
Yes
?- walk(rb1,rb2,L).
L = [b2, b5] ;
L = [b2, b6] ;
L = [b2, b1, b7] ;
L = [b2, b1, b4, b3, b6] ;
..(υπάρχουν και πολλές άλλες λύσεις)...
```

(γ) Ο Euler ήταν ο πρώτος που απέδειξε ότι δεν υπάρχει διαδρομή που να διέρχεται από οποιαδήποτε τοποθεσία και καταλήγει σε οποιαδήποτε τοποθεσία που να διέρχεται από όλες τις γέφυρες μόνο μια φορά. Να ορίσετε ένα κατηγόρημα euler/0 που να αποδεικνύει ότι δεν υπάρχει τέτοια διαδρομή.

```
?- euler.
yes
```

11.6 Ο λεγόμενος και γρίφος του Einstein λέει ότι: "Πέντε άνδρες έχουν διαφορετικές εθνικότητες και ζουν σε πέντε διαφορετικά σπίτια. Εξασκούν πέντε διαφορετικά επαγγέλματα και καθένας τους έχει ένα αγαπημένο κατοικίδιο και ένα αγαπημένο ποτό. Κάθε σπίτι είναι βαμμένο με διαφορετικό χρώμα.

- The Englishman lives in the red house.
- The Spaniard owns a dog.
- The Japanese is a painter.
- The Italian drinks tea.
- The Norwegian lives in the first house on the left.
- The owner of the fox drinks water.
- The owner of the green house drinks coffee.
- The green house is on the right of the white one.
- The sculptor breeds snails.
- The diplomat lives in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian's house is next to the blue one.
- The violinist drinks fruit juice.
- The fox is in a house next to that of the doctor.
- The horse is in a house next to that of the violinist."

Γράψτε ένα Prolog πρόγραμμα που να απαντά στην ερώτηση "*Ποιος έχει τη ζέβρα?*". Η λύση εκφράζεται σαν λίστα που αρχικά έχει πολλές αδέσμευτες μεταβλητές:

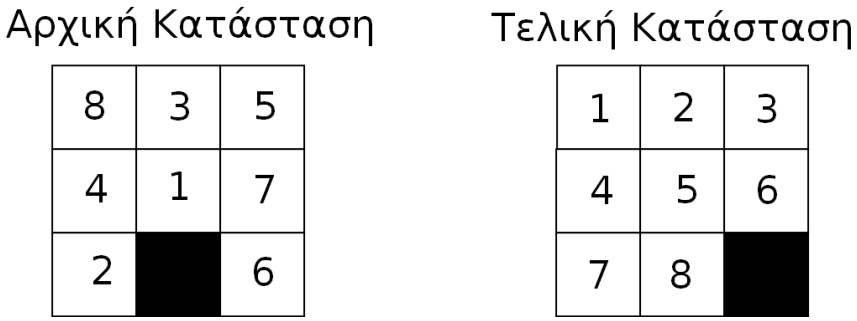
```
houseplan([
  house(_,norwegian,_,_,_),
  house(_,_,_,_,_),
  house(_,_,_,milk,_),
  house(_,_,_,_,_),
  house(_,_,_,_,_)
        ]).
```

Το τελικό πρόγραμμα θα δείχνει κάπως έτσι:

```
who_owns_zebra(Owner):-
  houseplan(Houses),
... (many prolog lines) ...
  next_to(house(_,_,_,_,seven_stars),  house(_,Owner,zebra,_,_),
Houses),
... (many Prolog lines)....
```

Προφανώς πρέπει να ορίσετε το κατηγόρημα next_to/3 που αληθεύει όταν τα δύο πρώτα ορίσματα είναι διπλανά σε μία λίστα που δίνεται ως τρίτο όρισμα.

11.7 Το N-παζλ είναι ένα γνωστό παζλ με συρόμενα τετράγωνα που αποτελείται από αριθμημένα τετράγωνα και έναν κενό χώρο (Σχήμα 11.17). Τα τετράγωνα βρίσκονται αρχικά σε μια (ψεύδο) τυχαία σειρά. Ο στόχος του παζλ είναι να αναδιατάξετε τα τετράγωνα με κινήσεις που ολισθαίνουν τετράγωνα στο κενό χώρο.



Σχήμα 11.17: *Αρχική και τελική κατάσταση του 8-παζλ*

Συνήθως οι ευρετικές συναρτήσεις που χρησιμοποιούνται για αυτό το πρόβλημα είναι η μέτρηση των πλακιδίων ή το άθροισμα των αποστάσεων manhattan. Μια δυνατή αναπαράσταση του παζλ θα μπορούσε να είναι μια λίστα με τη μορφή [2,3,4,5,1,e, 6,7,8], όπου το πρώτο στοιχείο του πίνακα αντιστοιχεί στη θέση παζλ (1,1), η δεύτερη σε η θέση (1,2) κ.α.

α) Να αναπαραστήσετε το πρόβλημα του παζλ σε Prolog.

β) Να υλοποιήσετε το κατηγόρημα heuristic_Func(S, HV) για το παζλ.

γ) Να κάνετε μερικά πειράματα για να συγκρίνετε την απόδοση των τριών αλγορίθμων που παρουσιάστηκαν σε αυτό το κεφάλαιο.

11.8 Να υλοποιήσετε ένα πρόγραμμα Prolog που να λύνει την κρυπταριθμητική εξίσωση:

DONALD + GERALD = ROBERT,

αναθέτοντας έναν διαφορετικό αριθμό σε κάθε γράμμα.

ΚΕΦΑΛΑΙΟ 12: Λογικός Προγραμματισμός με Περιορισμούς

Λέξεις Κλειδιά:
Προβλήματα ικανοποίησης περιορισμών. Η έννοια του περιορισμού σε μεταβλητές. Πεδία μεταβλητών. Επίλυση προβλημάτων περιορισμών. Αλγόριθμοι διήθησης δυαδικών περιορισμών και περιορισμών ανώτερης τάξης. Υποστήριξη περιορισμών στο Λογικό Προγραμματισμό. Η γλώσσα ECLiPSe ως όχημα Προγραμματισμού με περιορισμούς. Παραδείγματα κατηγοριών προβλημάτων (χρονοπρογραμματισμός, ανάθεση πόρων).

Περίληψη
Ο λογικός προγραμματισμός με περιορισμούς, αποτελεί μια από τις πιο πετυχημένες επεκτάσεις του λογικού προγραμματισμού, με εφαρμογή σε πλήθος βιομηχανικών προβλημάτων, όπως για παράδειγμα προβλήματα χρονοπρογραμματισμού. Το κεφάλαιο παρουσιάζει σύντομα τον ορισμό των προβλημάτων ικανοποίησης περιορισμών, καθώς και έννοιες όπως εκείνη της εφαρμογής περιορισμών σε πεδία μεταβλητών. Αναφέρονται σύντομα το πώς επιλύονται τέτοιου είδους προβλήματα με εφαρμογή αλγορίθμων διήθησης περιορισμών και αναζήτηση. Χρησιμοποιώντας μια καθιερωμένη πλατφόρμα προγραμματισμού περιορισμών (ECLiPSe Prolog) και των βιβλιοθηκών που προσφέρει, παρουσιάζονται πλήθος παραδειγμάτων συνδυαστικών εφαρμογών, όπως είναι ο χρονοπρογραμματισμός και η ανάθεση πόρων, δείχνοντας έτσι στον αναγνώστη πως ο λογικός προγραμματισμός αποτελεί κατάλληλη πλατφόρμα για την ανάπτυξη τέτοιων προγραμμάτων.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός:

- Να γνωρίζει τις διαφορές μεταξύ συστημάτων λογικού προγραμματισμού και λογικού προγραμματισμού με περιορισμούς, και ιδιαίτερα τις έννοιες των μεταβλητών περιορισμών, των πεδίων τιμών τους, την λειτουργία των περιορισμών και της διαδικασίας επίλυσής τους.
- Να μοντελοποιεί απλά προβλήματα του πραγματικού κόσμου σαν προβλήματα ικανοποίησης περιορισμών και να παρέχει τις αντίστοιχες υλοποιήσεις τους.
- Να παράγει βέλτιστες λύσεις με τη χρήση του αλγορίθμου επέκτασης και οριοθέτησης.
- Να επιλύει προβλήματα ανάθεσης χρησιμοποιώντας ειδικούς περιορισμούς, όπως ο περιορισμός element/3.
- Να χρησιμοποιεί τους περιορισμούς disjunctive/2 και cumulative/4 για να μοντελοποιεί και να υλοποιεί λύσεις σε προβλήματα χρονοπρογραμματισμού.

Παράδειγμα κίνητρο

Η διαχείριση των αριθμητικών εκφράσεων στην Prolog δεν ακολουθεί τη συνηθισμένη δηλωτική ερμηνεία. Για παράδειγμα, στην ερώτηση:

```
?- X > 3, X < 6.
```

η Prolog θα απαντήσει με ένα μήνυμα σφάλματος, καθώς δεν μπορεί να αποφανθεί για την αλήθεια των δύο υποστόχων. Για να μπορέσει να απαντήσει η Prolog κατάλληλα στο παραπάνω ερώτημα, είναι αναγκαίο να έχουμε ορίσει ένα εύρος επιτρεπτών τιμών για την μεταβλητή X. Για παράδειγμα, θα μπορούσαμε να δώσουμε στο X ακέραιες τιμές από ένα σύνολο χρησιμοποιώντας το κατηγόρημα member/2:

```
?- member(X,[1,2,3,4,5,6,7]), X > 3, X < 6.
X = 4
Yes
X = 5
Yes
```

```
No
```

Όπως φαίνεται στην παραπάνω αλληλεπίδραση με το σύστημα, για να είναι δυνατό να απαντήσει η Prolog στο παραπάνω ερώτημα, θα πρέπει να ορίσουμε ένα εύρος τιμών (πεδίο-domain), να δώσουμε μια τιμή στην μεταβλητή από αυτό το πεδίο (υποστόχος member/2) και έπειτα να διατυπώσουμε τις σχέσεις (περιορισμούς) για τη συγκεκριμένη μεταβλητή. Αυτό αποτελεί υλοποίηση της κλασικής τεχνικής παραγωγής και δοκιμής που εξετάστηκε στο Κεφάλαιο 11, καθώς πρώτα κατασκευάζεται η λύση με το κατηγόρημα member/2 και έπειτα γίνεται ο έλεγχος. Σημειώνεται ότι λύση στην περίπτωση αυτή είναι μια τιμή στην μεταβλητή X, που να ικανοποιεί τις σχέσεις. Ας εξετάσουμε ένα παράδειγμα με τρεις μεταβλητές και περισσότερους περιορισμούς:

```
solve(X,Y,Z):-
  member(X,[1,2,3,4,5,6,7,8,9,10]),
  member(Y,[1,2,3,4,5,6,7,8,9,10]),
  member(Z,[4,5,6,7,8,9,10,11]),
  X > 3, X < 9,
  Y > 2, Y < 9,
  X =:= Y + 1,
  Z > 5,
  Z < X.
```

Μια προσεκτικότερη ανάλυση του παραπάνω, δείχνει ότι η συγκεκριμένη τεχνική, οδηγεί σε άσκοπες οπισθοδρομήσεις. Για παράδειγμα, η τιμή 1 για την μεταβλητή X, δοκιμάζεται με όλους τους συνδυασμούς τιμών για την Y (από 1 έως και 10), ενώ είναι προφανές από την σχέση X > 3 ότι η συγκεκριμένη τιμή δεν θα αποτελέσει ποτέ μέρος της λύσης. Ομοίως, το ίδιο συμβαίνει για την τιμή 2 του X, κοκ. Σε μικρά προβλήματα όπως το παραπάνω οι περιττές οπισθοδρομήσεις δεν αποτελούν πρόβλημα, όμως αν το πρόβλημα έχει πολλές μεταβλητές, με μεγάλο εύρος τιμών (για παράδειγμα από -1000 έως 1000) και πλήθος περιορισμών, τότε η παραπάνω διαδικασία είναι εξαιρετικά χρονοβόρα, καθώς παρουσιάζεται το φαινόμενο της **συνδυαστικής έκρηξης** (combinatorial explosion).

Το πρόβλημα που παρουσιάστηκε στο παράδειγμα έχει μερικά ενδιαφέροντα χαρακτηριστικά: α) υπάρχει ένα σύνολο μεταβλητών οι οποίες παίρνουν τιμές από ένα πεδίο τιμών και β) υπάρχουν λογικές σχέσεις πάνω στις μεταβλητές που περιορίζουν τις δυνατές τιμές που μπορούν να ανατεθούν σε αυτές. Το ζητούμενο είναι να βρεθούν αναθέσεις τιμών στις μεταβλητές ώστε να ικανοποιούνται οι περιορισμοί. Αυτή είναι η περιγραφή ενός προβλήματος ικανοποίησης περιορισμών και ο (λογικός) προγραμματισμός με περιορισμούς φιλοδοξεί να βρει τεχνικές για την αποδοτική επίλυση προβλημάτων της κατηγορίας.

Στο παρόν κεφάλαιο θα χρησιμοποιηθεί σαν όχημα, το σύστημα λογικού προγραμματισμού ECLiPSe. Αν και οι περισσότερες υλοποιήσεις της γλώσσας Prolog προσφέρουν πλέον την δυνατότητα προγραμματισμού με περιορισμούς, η ECLiPSe Prolog αποτελεί ένα από τα ισχυρότερα και αποδοτικότερα εργαλεία στην συγκεκριμένη περιοχή. Μια διαφορά της με την SWI-Prolog που χρησιμοποιήθηκε στα προηγούμενα κεφάλαια είναι ότι στην ECLiPSe Prolog η απόδειξη ενός ερωτήματος σηματοδοτείται με yes (true), ενώ η αποτυχία απόδειξης με no (false).

Μονοπάτι Μάθησης

Τα βασικά στοιχεία για το κεφάλαιο αυτό βρίσκονται στις ενότητες 12.1 έως και 12.8 οι οποίες πρέπει να ολοκληρωθούν από τον αναγνώστη. Οι ενότητες περιλαμβάνουν βασικές έννοιες για τα προβλήματα ικανοποίησης περιορισμών, παρουσίαση των βιβλιοθηκών των ακεραίων περιορισμών, των εναλλακτικών τρόπων αναζήτησης της λύσης, εύρεσης βέλτιστης λύσης και τους ειδικούς περιορισμούς που έχουν αναπτυχθεί για προβλήματα χρονοπρογραμματισμού. Οι ενότητες 12.7 και 12.8 ειδικότερα παρουσιάζουν δύο σημαντικά παραδείγματα εφαρμογής. Η ενότητα 12.9 περιέχει μια παρουσίαση των περιορισμών ως δεδομένα (reified constraints) που δεν είναι όμως απαραίτητη σε πρώτη ανάγνωση. Τέλος, η ενότητα 12.10 περιέχει σημαντικές παρατηρήσεις πάνω στον προγραμματισμό με περιορισμούς και θα πρέπει να περιληφθεί σε μια πρώτη ανάγνωση.

12.1 Προβλήματα Ικανοποίησης Περιορισμών

Ξεκινώντας με τον κλασικό ορισμό, ένα πρόβλημα **ικανοποίησης περιορισμών** (constraint satisfaction problem) αποτελείται από:

- Ένα σύνολο n **μεταβλητών** $V_1, V_2,...,V_n$,
- Ένα σύνολο n **πεδίων τιμών** $D_1, ...D_n$, που αντιστοιχούν σε κάθε μεταβλητή, έτσι ώστε $V_i \in D_i$
- Ένα σύνολο **περιορισμών** $C_1, C_2,...C_m$ όπου $C_i(V_k,...,V_m)$ είναι μια σχέση μεταξύ των μεταβλητών του προβλήματος.

Λύση στο πρόβλημα, αποτελεί μια ανάθεση τιμών στις μεταβλητές του προβλήματος, τέτοια ώστε να ικανοποιούνται οι περιορισμοί, δηλαδή:

$$V_1 = d_1 \wedge V_2 = d_2 \wedge ... \wedge V_n = d_n \wedge d_1 \in D_1 \wedge d_2 \in D_2 \wedge ... \wedge d_i \in D_n \wedge C_1 \wedge C_2 \wedge ... \wedge C_m$$

Οι περιορισμοί είναι λογικές σχέσεις μεταξύ μεταβλητών, δηλαδή αποτιμώνται σε αληθείς ή ψευδείς και περιορίζουν τις πιθανές τιμές που μπορούν να πάρουν οι μεταβλητές, δηλαδή εκφράζουν μερική πληροφορία για το πρόβλημα. Για παράδειγμα η σχέση X =:= Y + 1, δηλώνει ότι οι τιμές για τις δύο μεταβλητές X και Y, πρέπει να είναι διαδοχικές (διαδοχικοί ακέραιοι), με την τιμή της X μεγαλύτερη εκείνης της Y. Ανάλογα με τη μοντελοποίηση του προβλήματος, οι περιορισμοί μπορούν να εκφράσουν πολύ πιο ενδιαφέρουσα πληροφορία. Για παράδειγμα σε μια εφαρμογή χρονοπρογραμματισμού, αν SA και SB είναι οι χρόνοι έναρξης των εργασιών A και B, και DA η διάρκεια της A, τότε ο περιορισμός

$$SA + DA < SB$$

δηλώνει ότι η εργασία B πρέπει να γίνει μετά την περάτωση της εργασίας A.

Ανάλογα με το πόσες μεταβλητές περιλαμβάνει ένας περιορισμός χαρακτηρίζεται ως:

- **μοναδιαίος** (unary), όταν αφορά μια μεταβλητή,
- **δυαδικός** (binary), όταν αφορά δύο μεταβλητές ή
- **ανώτερης τάξης** (higher order), όταν αφορά περισσότερες των δύο μεταβλητές.

Χαρακτηριστικά των περιορισμών είναι ότι είναι:

- δηλωτικοί, δηλαδή ορίζουν μια σχέση μεταξύ των οντοτήτων του προβλήματος χωρίς να ορίζουν μια συγκεκριμένη υπολογιστική διαδικασία,
- προσθετικοί, γιατί ενδιαφέρει συνήθως η σύζευξη των περιορισμών και όχι η σειρά με την οποία τέθηκαν, και
- σπανίως ανεξάρτητοι, καθώς στη συνηθέστερη περίπτωση οι περιορισμοί έχουν κοινές μεταβλητές.

Αν και ο ορισμός που παρουσιάστηκε παραπάνω φαίνεται να είναι εξαιρετικά απλός, εντούτοις υπάρχει ένα πλήθος πολύπλοκων και με ιδιαίτερα μεγάλο ενδιαφέρον προβλημάτων που εμπίπτουν σε αυτόν, όπως θα δούμε σε επόμενες παραγράφους.

Για την επίλυση των παραπάνω προβλημάτων μπορούν να εφαρμοστούν πολλές τεχνικές. Αυτό που παίζει σημαντικό ρόλο στο ποιες τεχνικές θα χρησιμοποιηθούν είναι το είδος των τιμών που περιέχονται στο πεδίο τιμών των μεταβλητών. Έτσι, ανάλογα με το αν αυτές οι τιμές είναι ακέραιες, πραγματικές, σύνολα πεπερασμένων τιμών, κλπ, χρησιμοποιούνται τεχνικές, όπως αλγόριθμοι διήθησης, μέθοδος Simplex κοκ. Στα επόμενα θα επικεντρωθούμε κυρίως σε τεχνικές που εφαρμόζονται σε πεπερασμένα πεδία (finite domains) και ειδικότερα στο πεδίο των ακέραιων τιμών.

Επίλυση Προβλημάτων Περιορισμών

Η απλούστερη και λιγότερο αποδοτική προσέγγιση στην επίλυση περιορισμών, είναι η παραγωγή και δοκιμή (Κεφάλαιο 11) η υλοποίηση της οποίας παρουσιάστηκε παραπάνω και η οποία χρησιμοποιείται σπάνια για την επίλυση πολύπλοκων προβλημάτων του πραγματικού κόσμου. Καθώς το πρόβλημα των περιορισμών

μπορεί να μοντελοποιηθεί σαν ένα πρόβλημα αναζήτησης, μπορούν κάλλιστα να εφαρμοστούν κλασικές τεχνικές επίλυσης προβλημάτων από την περιοχή της τεχνητής νοημοσύνης, δηλαδή τυφλή ή ευρετική αναζήτηση (Κεφάλαιο 11).

Στην περίπτωση αυτή, μια κατάσταση (state) αποτελείται από τις μεταβλητές του προβλήματος και υπάρχει ένας μόνο τελεστής, ο οποίος αντιστοιχεί στην ανάθεση μιας τιμής σε μια μη-δεσμευμένη μεταβλητή. Προφανώς στην αρχική κατάσταση όλες οι μεταβλητές είναι μη-δεσμευμένες, ενώ στην τελική κατάσταση, η οποία αποτελεί λύση του προβλήματος, έχουν ανατεθεί τιμές σε όλες τις μεταβλητές του προβλήματος και ικανοποιούνται οι περιορισμοί. Καθώς σε κάθε βήμα μια μεταβλητή δεσμεύεται σε τιμή, το δένδρο αναζήτησης είναι πεπερασμένο, άρα ένας κατάλληλος αλγόριθμός αναζήτησης είναι η αναζήτηση κατά βάθος. Σε κάθε βήμα της αναζήτησης, δηλαδή σε κάθε ανάθεση τιμής, ελέγχεται η συνέπεια των περιορισμών στους οποίους συμμετέχουν μεταβλητές οι οποίες έχουν ήδη πάρει τιμή. Υπενθυμίζουμε εδώ ότι η αναζήτηση κατά βάθος είναι ο αλγόριθμος που χρησιμοποιεί η Prolog για την κατασκευή του δένδρου απόδειξης.

Υλοποιώντας την προσέγγιση σε Prolog, στο παράδειγμα που παρουσιάστηκε παραπάνω, οι περιορισμοί που αφορούν μόνο τη μεταβλητή X, μπορούν να ελεχθούν ακριβώς μετά τον πρώτο υποστόχο member/2, ο οποίος αποτελεί και το βήμα ανάθεσης τιμής (τελεστής ανάθεσης) και φυσικά το ίδιο μπορεί να γίνει και για τις μεταβλητές Y και Z. Επίσης ο περιορισμός X =:= Y + 1, μπορεί να τοποθετηθεί πριν από την ανάθεση τιμής στην μεταβλητή Z:

```
solve_dfs(X,Y,Z):-
  member(X, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
  X > 3, X < 9,
  member(Y, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
  Y > 2, Y < 9,
  X =:= Y + 1,
  member(Z, [4, 5, 6, 7, 8, 9, 10, 11]),
  Z > 5,
  Z < X.
```

Η παραπάνω υλοποίηση είναι περισσότερο αποδοτική, γιατί αντί να ελεγχθούν όλοι οι συνδυασμοί των τιμών των μεταβλητών Y και Z για τιμή X = 1, δηλαδή (1,1,4), (1,1,5), … (1,10,11), ο περιορισμός αποτυγχάνει άμεσα, ακριβώς μετά την ανάθεση τιμής στην μεταβλητή X, γλυτώνοντας έτσι 80 (10 * 8) δοκιμές, οι οποίες είναι βέβαιο ότι δεν θα οδηγήσουν σε λύση. Το ίδιο συμβαίνει και για τις τιμές 2 και 3 του X κοκ.

Μια βελτίωση του προγράμματος προέρχεται από την αναδιάταξη των βημάτων ανάθεσης τιμών στις μεταβλητές. Έτσι αν η ανάθεση τιμών για την μεταβλητή Z γίνει πριν από τις δύο άλλες αναθέσεις τιμών, με κατάλληλη αναδιάταξη των υποστόχων των περιορισμών, μειώνεται ο αριθμός των οπισθοδρομήσεων:

```
solve_dfs_h(X,Y,Z):-
  member(Z, [4, 5, 6, 7, 8, 9, 10, 11]),
  Z > 5,
  member(X, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
  X > 3, X < 9,
  Z < X,
  member(Y, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
  Y > 2, Y < 9,
  X =:= Y + 1.
```

Είναι εύκολο να διαπιστώσουμε τη μείωση των οπισθοδρομήσεων, αν αναλογιστούμε ότι η πρώτη έγκυρη (συνεπής) τιμή του Z είναι το 6 (περιορισμός Z >5) και άρα οι τιμές X = 4 και X = 5, θα αποτύχουν (νωρίς) στον έλεγχο X > Z, πριν να γίνει οποιαδήποτε ανάθεση στις τιμές της μεταβλητής Y. Ποια είναι όμως η κατάλληλη σειρά με την οποία οι μεταβλητές πρέπει να πάρουν τιμές, ώστε το δένδρο να μειωθεί όσο το δυνατό περισσότερο;

Καθώς δεν υπάρχει ακριβής μέθοδος για να καθορίσουμε μια τέτοια σειρά οδηγούμαστε στην εισαγωγή ευρετικών μεθόδων στην επίλυση προβλημάτων ικανοποίησης περιορισμών. Εφόσον στην αναζήτηση υπάρχει μόνο ένας τελεστής, η επιλογή σε κάθε βήμα έγκειται στο ποια θα είναι η επόμενη ελεύθερη μεταβλητή στην οποία θα ανατεθεί τιμή (variable ordering). Μια συνηθισμένη ευρετική συνάρτηση βασίζεται στην αρχή της συντομότερης αποτυχίας (first fail principle): επιλέγεται η μη-δεσμευμένη μεταβλητή η οποία

είναι πιθανότερο να οδηγήσει συντομότερα σε αποτυχημένους κόμβους στην αναζήτηση, δηλαδή εκείνη με το μικρότερο πεδίο τιμών και στο συγκεκριμένο παράδειγμα είναι η τεχνική που εξετάστηκε παραπάνω (κατηγόρημα solve_dfs_h/3). Σε περίπτωση που περισσότερες μεταβλητές έχουν ίδιο αριθμό τιμών στα πεδία τους, τότε επιλέγεται η μεταβλητή που συμμετέχει σε περισσότερους περιορισμούς (most constrained principle).

Μια δεύτερη επιλογή αφορά σε ποια τιμή από το πεδίο τιμών μιας μεταβλητή θα εξεταστεί πρώτη (value ordering). Οι δύο αυτές επιλογές επηρεάζουν σημαντικά το μέγεθος του δένδρου αναζήτησης και έχουν δραματική επίδραση στην ταχύτητα εύρεσης λύσης σε ένα πρόβλημα περιορισμών.

Αν και με την εισαγωγή των ευρετικών μεθόδων η απόδοση μπορεί να αυξηθεί σημαντικά, η κλασική προσέγγιση της ΤΝ στο πρόβλημα δεν εκμεταλλεύεται την πληροφορία που υπάρχει στους περιορισμούς και αφορά στα πεδία των τιμών. Αυτό επιτυγχάνουν οι αλγόριθμοι διήθησης.

Αλγόριθμοι Διήθησης/Συνέπειας

Η απόδοση των παραπάνω υλοποιήσεων μπορεί να αυξηθεί αν εκμεταλλευτούμε τη μερική πληροφορία που υπάρχει στους περιορισμούς. Για παράδειγμα, για τη μεταβλητή Z, είναι δυνατό να μειώσουμε εύκολα το αρχικό πεδίο τιμών αν λάβουμε υπόψη τους μοναδιαίους περιορισμούς για το Z. Έτσι οι ακόλουθοι υποστόχοι:

```
member(Z, [4, 5, 6, 7, 8, 9, 10, 11]),
Z > 5,
```

μετατρέπονται στον:

```
member(Z, [6, 7, 8, 9, 10, 11]),
```

Η ιδέα της εκμετάλλευσης των περιορισμών για τη μείωση των πεδίων τιμών είναι κεντρική στους **αλγορίθμους διήθησης** (filtering) ή αλλιώς **συνέπειας τιμών** (consistency). Το παραπάνω αποτελεί εφαρμογή της λεγόμενης μοναδιαίας συνέπειας ή συνέπειας κόμβου (node consistency), δηλαδή της αφαίρεσης από τα πεδία των μεταβλητών των τιμών εκείνων που αποκλείονται λόγω περιορισμών στους οποίους συμμετέχει μόνο μια μεταβλητή και αποτελεί την απλούστερη μορφή συνέπειας. Εφαρμόζοντας την ίδια προσέγγιση για τις υπόλοιπες μεταβλητές η υλοποίηση γίνεται:

```
solve_filter(X,Y,Z):-
  member(Z, [6, 7, 8, 9, 10, 11]),
  member(X, [4, 5, 6, 7, 8]),
  Z < X,
  member(Y, [3, 4, 5, 6, 7, 8]),
  X =:= Y + 1.
```

Εξετάζοντας την σχέση Z < X, και το πεδίο της μεταβλητής Z, μπορούμε να συμπεράνουμε ότι η τιμή 8 της Z δεν μπορεί ποτέ να συμμετέχει στην τελική λύση, γιατί δεν υπάρχει τιμή για τη X, τέτοια ώστε να ικανοποιεί τον περιορισμό. Διατυπώνοντας το λίγο διαφορετικά, αν την Z πάρει την τιμή 8 δεν υπάρχει διαθέσιμη τιμή για την X που να ικανοποιεί το περιορισμό Z < X, καθώς η μέγιστη τιμή της X είναι το 8. Το ίδιο ισχύει και για τις τιμές 9, 10 και 11 της μεταβλητής Z. Άρα, το πρόγραμμα μπορεί να μετασχηματιστεί στο:

```
solve_filter(X,Y,Z):-
  member(Z, [6, 7]),
  member(X, [4, 5, 6, 7, 8]),
  Z < X,
  member(Y, [3, 4, 5, 6, 7, 8]),
  X =:= Y + 1.
```

Αλλά ο συγκεκριμένος περιορισμός θέτει όρια και για τις τιμές της X, και εφόσον η τιμή της X πρέπει να είναι αυστηρά μεγαλύτερη της τιμής της Z, οι τιμές 4,5 και 6 της X δεν μπορούν να συμμετέχουν στην λύση:

```
solve_filter(X,Y,Z):-
  member(Z, [6, 7]),
  member(X, [7, 8]),
  Z < X,
  member(Y, [3, 4, 5, 6, 7, 8]),
```

```
X =:= Y + 1.
```

Η παραπάνω λογική της εξέτασης των δυαδικών περιορισμών, αποτελεί την βάση των αλγορίθμων συνέπειας τόξου (arc consistency). Οι αλγόριθμοι διαγράφουν τιμές από τα πεδία των μεταβλητών εξετάζοντας δυαδικούς περιορισμούς μεταξύ δύο μεταβλητών X και Y, αφαιρώντας εκείνες τις τιμές από το πεδίο της X για τις οποίες δεν υπάρχει υποστήριξη (support) στο πεδίο της Y. Με άλλα λόγια εκείνες τις τιμές της μεταβλητής X για τις οποίες δεν υπάρχει τιμή της Y, τέτοια ώστε να ικανοποιείται ο περιορισμός.

Η εφαρμογή των παραπάνω στην περίπτωση του περιορισμού X =:= Y + 1, καταλήγει στην αφαίρεση των τιμών 3,4,5 και 8 από το πεδίο της Y:

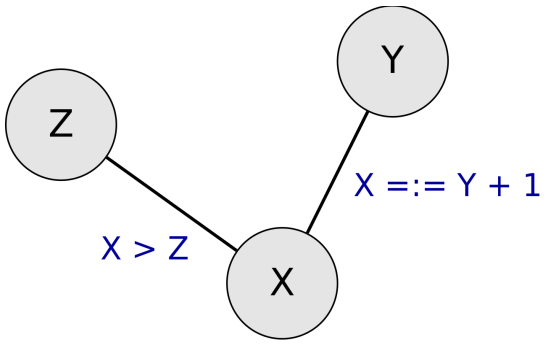
```
solve_filter(X,Y,Z):-
  member(Z, [6, 7]),
  member(X, [7, 8]),
  Z < X,
  member(Y, [6, 7]),
  X =:= Y + 1.
```

Η μείωση των πεδίων τιμών έχει δραματική επίδραση στο μέγεθος του δένδρου αναζήτησης. Στο παράδειγμά που εξετάστηκε, από τους αρχικούς 800 συνδυασμούς τιμών (10 * 10 * 8), απέμειναν στο τέλος της διαδικασίας μόνο 8 συνδυασμοί.

Υπάρχουν μερικές ενδιαφέρουσες παρατηρήσεις στα παραπάνω:

- μετά την εξέταση των μοναδιαίων περιορισμών, οι τελευταίοι δεν ήταν πλέον απαραίτητοι για την εύρεση της λύσης,
- οι δυαδικοί περιορισμοί διατηρήθηκαν ακόμη και μετά την μείωση των αντίστοιχων πεδίων τιμών,
- οι αλλαγές στο πεδίο τιμών της μεταβλητής X λόγω του περιορισμού Z < X, οδήγησε σε μείωση τους πεδίου της Y λόγω του περιορισμού X =:= Y + 1. Άρα μετά από κάποια αλλαγή στα πεδία των μεταβλητών, είναι απαραίτητο να εξεταστούν περιορισμοί στους οποίους συμμετέχει η μεταβλητή της οποίας το πεδίο μειώθηκε.

Τα παραπάνω αποτελούν τον κορμό της λειτουργίας των αλγορίθμων συνέπειας. Σε αυτούς τους αλγορίθμους το πρόβλημα αναπαρίσταται ως γράφος (γράφος περιορισμών - constraint graph), με τους κόμβους (nodes) να αναπαριστούν τις μεταβλητές και τα τόξα (arcs) τους περιορισμούς μεταξύ αυτών (Σχήμα 12.1). Καθώς η επίδραση στα πεδία τιμών των μοναδιαίων περιορισμών, γίνεται μία μόνο φορά, στους γράφους περιορισμών εμφανίζονται συνήθως μόνο περιορισμοί που αφορούν περισσότερες μεταβλητές.



Σχήμα 12.1: *Γράφος περιορισμών παραδείγματος*

Υπάρχουν διαφορετικοί αλγόριθμοι διήθησης τιμών με διαφορετική απόδοση και ο βαθμός συνέπειας σηματοδοτεί το πόσες ασυνεπείς τιμές αφαιρούν από τα πεδία. Συνήθως ο βαθμός συνέπειας που επιτυγχάνει ένας αλγόριθμος είναι αντιστρόφως ανάλογος με τον απαιτούμενο χρόνο εκτέλεσής του.

Ο απλούστερος αλγόριθμος συνέπειας είναι ο αλγόριθμος συνέπειας κόμβου (Node Consistency), που εξετάζει μόνο τους μοναδιαίους περιορισμούς. Η πλέον διαδεδομένη κατηγορία αλγορίθμων είναι οι αλγόριθμοι συνέπειας τόξου (Arc Consistency), οι οποίοι απαλείφουν τιμές ελέγχοντας τους δυαδικούς περιορισμούς του προβλήματος. Στην βιβλιογραφία, έχουν προταθεί πλήθος αλγορίθμων όπως για

παράδειγμα AC-3, AC-4, AC-5, AC-2000, κλπ. οι οποίοι αποτελούν βελτιώσεις της βασικής μεθόδου δηλαδή της διαγραφής τιμών από τα πεδία μέχρι να μην μπορούν να εφαρμοστούν άλλοι περιορισμοί. Υπάρχουν και άλλες κατηγορίες αλγορίθμων συνέπειας, όπως για παράδειγμα οι αλγόριθμοι συνέπειας μονοπατιού (path consistency algorithms), που επιβάλλουν μεγαλύτερο βαθμό συνέπειας σε ένα γράφο περιορισμών. Οι αλγόριθμοι ελέγχου συνέπειας αποτελούν μια μορφή διάδοσης περιορισμών (constraint propagation), καθώς τα πεδία τιμών των μεταβλητών μειώνονται σταδιακά βάσει των περιορισμών.

Στο παρόν κεφάλαιο δεν θα περιγράψουμε σε μεγαλύτερο βάθος την περιοχή των αλγορίθμων περιορισμών καθώς ξεφεύγει από τον σκοπό του. Ο αναγνώστης μπορεί να ανατρέξει στην βιβλιογραφία στο τέλος του κεφαλαίου, για μια περισσότερο λεπτομερή περιγραφή των παραπάνω.

Οι κλασικοί αλγόριθμοι συνέπειας τόξου δεν αφαιρούν πάντα όλες τις ασυνεπείς τιμές από τα πεδία των περιορισμών. Κλασικό παράδειγμα που αναφέρεται στην βιβλιογραφία είναι όταν το πρόβλημα αποτελείται από τρεις μεταβλητές, με πεδία τιμών το σύνολο [1,2] οι οποίες πρέπει να είναι διαφορετικές μεταξύ τους:

$$X,Y,Z \in [1,2] : X \neq Y \wedge Y \neq Z \wedge X \neq Z$$

Αν και στο παραπάνω πρόβλημα δεν υπάρχει λύση, καθώς οι αλγόριθμοι συνέπειας τόξου εξετάζουν τον κάθε περιορισμό μεμονωμένα, δεν αφαιρούν καμία τιμή από τα πεδία. Για παράδειγμα, στον περιορισμό X =\= Y, αν η τιμή 1 του X έχει υποστήριξη την τιμή 2 του Y, κοκ. Άρα είναι δυνατό να παραμένουν στα πεδία τιμές οι οποίες δεν μπορούν να συμμετέχουν στην τελική λύση. Αν και υπάρχουν αλγόριθμοι συνέπειας μεγαλύτερης τάξης (και υψηλότερης υπολογιστικής πολυπλοκότητας) που αφαιρούν όλες τις τιμές από τα πεδία των μεταβλητών, η εφαρμογή τους είναι συνήθως αρκετά χρονοβόρα και δεν προτιμάται.

Η παραπάνω παρατήρηση οδηγεί στην τελική προσέγγιση στην επίλυση των προβλημάτων της κατηγορίας: συνδυασμός αναζήτησης και αλγορίθμων συνέπειας/διήθησης τιμών. Η βασική ιδέα είναι η μείωση του χώρου αναζήτησης με τη χρήση ενός αλγορίθμου συνέπειας πριν από κάθε βήμα ανάθεσης τιμών. Αυτήν την προσέγγιση ακολουθούν τα περισσότερα συστήματα λογικού προγραμματισμού με περιορισμούς. Τα συστήματα αυτά, όπως για παράδειγμα η ECLiPSe Prolog, προσφέρουν την υποδομή για την εύκολη και αποδοτική ανάπτυξη προγραμμάτων της κατηγορίας.

Λογικός Προγραμματισμός και Περιορισμοί

Στην προηγούμενη ενότητα παρουσιάστηκε η λογική επίλυσης προβλημάτων περιορισμών με μείωση των πεδίων τιμών των μεταβλητών, χρησιμοποιώντας το κατηγόρημα member/2 και εφαρμόζοντας "χειρωνακτικά" τους αλγορίθμους διήθησης. Μια τέτοια προσέγγιση μπορεί να είναι εφικτή σε πολύ απλά προβλήματα όπως το παραπάνω, αλλά σε προβλήματα με πολλές μεταβλητές και μεγάλο πλήθος περιορισμών είναι απαγορευτική. Είναι απαραίτητο η παραπάνω διαδικασία να πραγματοποιείται αυτοματοποιημένα, δηλαδή η γλώσσα προγραμματισμού να προσφέρει όλες τις υποδομές για την επίλυση και ο προγραμματιστής να ορίζει τα πεδία των μεταβλητών, τους περιορισμούς πάνω σε αυτές και να αναζητεί τη λύση.

Για την υποστήριξη των παραπάνω, ο κλασικός λογικός προγραμματισμός επεκτείνεται στις ακόλουθες κατευθύνσεις:

- Εισάγεται ένας νέος τύπος μεταβλητών, των μεταβλητών περιορισμών, οι οποίες συνδέονται με ένα πεδίο τιμών.
- Για τις παραπάνω μεταβλητές, η κλασική ενοποίηση αντικαθίσταται με επίλυση περιορισμών. Αν για παράδειγμα προσπαθήσουμε να δώσουμε μια τιμή σε μια μεταβλητή περιορισμών, έξω από το πεδίο της, τότε η ανάθεση θα αποτύχει.
- Τέλος εισάγεται η έννοια της "αποθήκης περιορισμών" (constraint store), της διατήρησης δηλαδή των περιορισμών πάνω στις μεταβλητές, τους οποίους ο αλγόριθμος συνέπειας που χρησιμοποιείται από το σύστημα εκμεταλλεύεται για τη μείωση του πεδίου των τιμών, σε κάθε βήμα της εκτέλεσης.

Έτσι για παράδειγμα στην βιβλιοθήκη περιορισμών ic της ECLiPSe Prolog, την οποία θα εξετάσουμε στην επόμενη ενότητα, ο τελεστής #:: χρησιμοποιείται για να μετατρέψουμε μια μεταβλητή σε μεταβλητή περιορισμών με συγκεκριμένο πεδίο:

```
?- X #:: [1..10].
X = X{1..10}
yes
```

Η συγκεκριμένη σημειογραφία X{1..10} σημαίνει ότι η X είναι μεταβλητή περιορισμών με πεδίο τους ακεραίους από 1 έως και 10. Οι επόμενες ερωτήσεις δείχνουν πώς αλλάζει η διαδικασία ενοποίησης:

```
?- X #:: [1..10], X = 3.
X = 3
Yes
?- X #:: [1..10], X = 12.
No
```

Η πρώτη ερώτηση επιτυγχάνει, καθώς ο αριθμός 3 είναι μέσα στο πεδίο της μεταβλητής, ενώ η δεύτερη αποτυγχάνει εφόσον το 12 είναι εκτός των τιμών του πεδίου. Η ακόλουθη ερώτηση δείχνει ότι τα πεδία των μεταβλητών περιορισμών είναι δυναμικά, δηλαδή μεταβάλλονται κατά την εκτέλεση τους προγράμματος:

```
?- X #:: [1..10], X #> 5.
X = X{6..10}
Yes
```

Σαν απάντηση στην ερώτηση η Prolog επέστρεψε yes (true) και το νέο πεδίο της μεταβλητής. Ο τελεστής #> εκφράζει τον περιορισμό "μεγαλύτερο" στην βιβλιοθήκη ic. Θα πρέπει να σημειωθεί εδώ ότι η μεταβλητή X δεν έχει δεσμευτεί σε κάποια τιμή, αλλά μπορεί να πάρει μια τιμή (άγνωστη ακόμη) μέσα από το πεδίο της. Ενδιαφέρον παρουσιάζει και η ακόλουθη ερώτηση:

```
?- X #:: [1..10], Y #:: [1..10], X #< Y.
X = X{1..9}
Y = Y{2..10}
There is 1 delayed goal.
Yes
```

Από τα αρχικά πεδία των X και Y αφαιρέθηκαν οι τιμές που δεν μπορούν να συμμετέχουν στην τελική λύση, άρα έχουμε την εφαρμογή ενός αλγορίθμου συνέπειας. Η επισήμανση της Prolog ότι υπάρχει ένας στόχος σε αναβολή (delayed goal), σημαίνει ότι υπάρχει περιορισμός ο οποίος θα μπορούσε να μειώσει και άλλο τα πεδία τιμών (δες παράγραφο Στόχοι σε Αναβολή). Επιστρέφοντας στο ίδιο παράδειγμα, αν η ερώτηση ήταν:

```
?- X #:: [1..10], Y #:: [1..10], X #< Y, X = 3.
X = 3
Y = Y{4..10}
Yes
```

δεν υπάρχουν delayed goals, καθώς ο περιορισμός X #< Y δεν μπορεί να μειώσει άλλο τα πεδία, μετά την ανάθεση της τιμής 3 στο X. Ενδιαφέρον στο παραπάνω είναι ότι ο τελευταίος υποστόχος που έθεσε το X ίσο με 3, μείωσε ταυτόχρονα το πεδίο της Y αντίστοιχα. Αυτό γίνεται κάθε φορά που παρατηρείται οποιαδήποτε αλλαγή στο πεδίο τιμών μιας μεταβλητής περιορισμών. Για παράδειγμα:

```
?- X #:: [1..10], Y #:: [1..10], X #< Y, X #> 5.
X = X{6..9}
Y = Y{7..10}
There is 1 delayed goal.
Yes
```

Η έννοια της αποτυχίας των στόχων επεκτείνεται κατάλληλα στην περίπτωση των μεταβλητών περιορισμών, όπου ένας στόχος μπορεί να αποτύχει αν μετά την εφαρμογή των αλγορίθμων συνέπειας το πεδίο μιας μεταβλητής απομείνει κενό (χωρίς τιμές). Έτσι η ερώτηση:

```
?- X #:: [1..10], X #> 8.
X = X{[9, 10]}
Yes
```

επιτυγχάνει με νέο πεδίο της X το {9,10}, ενώ αν προσθέσω τον περιορισμό X #< 9:

```
?- X #:: [1..10], X #> 8, X #< 9.
No
```

αποτυγχάνει. Αυτό γίνεται γιατί το πεδίο της X απομένει χωρίς τιμές, άρα δεν υπάρχει λύση στο πρόβλημα. Για παράδειγμα:

```
?- X #:: [1..10], Y #:: [1..10], X #< Y, X #> Y.
No
```

Τέλος, είναι εύλογο η τελική ανάθεση τιμών να γίνεται δοκιμάζοντας τιμές από το τρέχον πεδίο των μεταβλητών. Αυτό γίνεται με το κατηγόρημα indomain/1, το οποίο δεσμεύει τη μεταβλητή περιορισμών του ορίσματός του σε διαδοχικές τιμές του πεδίου της. Κατά την οπισθοδρόμηση, η μεταβλητή περιορισμών δεσμεύεται σε διαδοχικές τιμές του πεδίου:

```
?- X #:: [1..10], X #> 8, indomain(X).
X = 9
Yes ;

X = 10
Yes
```

Σε περίπτωση που υπάρχουν πολλές μεταβλητές, συνηθίζεται να χρησιμοποιείται το κατηγόρημα labeling/1, το οποίο δέχεται μια λίστα μεταβλητών περιορισμών και για κάθε μια χρησιμοποιεί αναδρομικά το indomain/1 για να κάνει ανάθεση τιμών:

```
?- X #:: [1..10], Y #:: [1..10], X #< Y, X #> 7, labeling([X,Y]).
X = 8
Y = 9 ;
Yes

X = 8
Y = 10 ;
Yes

X = 9
Y = 10
Yes
```

Η παραπάνω ερώτηση, παρέχει και την πλήρη, κλασική δομή κωδικοποίησης ενός προβλήματος περιορισμών, η οποία περιλαμβάνει τη δήλωση των πεδίων τιμών, τη δήλωση των περιορισμών και τέλος την ανάθεση τιμών στις μεταβλητές. Στα επόμενα θα μελετήσουμε σε μεγαλύτερο βάθος την βιβλιοθήκη ic της ECLiPSe Prolog και τις σχετικές βιβλιοθήκες για την υποστήριξη προγραμματισμού με περιορισμούς. Για λόγους αναγνωσιμότητας, τα μηνύματα που αφορούν τους στόχους σε αναβολή παραλείπονται στο υπόλοιπο κεφάλαιο.

12.2 Περιορισμοί στην ECLiPSe Prolog

Η βιβλιοθήκη ic (interval constraints) υποστηρίζει περιορισμούς επί διαστημάτων ακεραίων και πραγματικών αριθμών. Χρησιμοποιεί σαν αλγόριθμο επίλυσης τον αλγόριθμο συνέπειας ορίων (bounds consistency) και επιτρέπει να αναμιχθούν ακέραιες και πραγματικές μεταβλητές στους περιορισμούς. Ένα σημαντικό χαρακτηριστικό είναι ότι υποστηρίζει τόσο γραμμικές όσο και μη γραμμικές αριθμητικές εκφράσεις στους υπολογισμούς.

Για την χρήση των κατηγορημάτων της βιβλιοθήκης, είναι απαραίτητο να "φορτωθεί" η τελευταία με μια από τις ακόλουθες δύο ντιρεκτίβες (directives), η οποία θα πρέπει να εμφανίζεται στην αρχή του αρχείου του προγράμματός:

```
:- lib(ic).
:- use_module(library(ic)).
```

Εναλλακτικά στην γραμμή ερωτήσεων της ECLiPSe να δώσουμε μία από τις παρακάτω ερωτήσεις:

```
?- lib(ic).
?- use_module(library(ic)).
```

Στα επόμενα θα επικεντρωθούμε στην επίλυση περιορισμών σε πεδία ακεραίων, αναφέροντας διαδοχικά το πώς δηλώνονται τα πεδία τιμών των μεταβλητών, τους βασικούς περιορισμούς σε ακέραιες εκφράσεις και τα κατηγορήματα εύρεσης λύσης σε ακέραια πεδία.

Δηλώσεις Πεδίων Τιμών

Η δήλωση του πεδίου τιμών μιας ή περισσοτέρων μεταβλητών, γίνεται με τη χρήση του τελεστή #::/2 :

```
Vars #:: Domain
```

όπου:

- Vars μια μεταβλητή ή μια λίστα μεταβλητών
- Domain ένα διάστημα της μορφής Low..High ή μια λίστα τέτοιων διαστημάτων καθώς και μεμονωμένων τιμών. Οι ειδικές τιμές -1.0Inf και 1.0Inf δηλώνουν το -∞ και το +∞ αντίστοιχα.

Ο τελεστής #:: δηλώνει ότι το πεδίο της μεταβλητής αποτελείται από ακέραιες τιμές. Για παράδειγμα

```
X #:: -10..10 .
X #:: [-2..3,8..15,19].
[X,Y,Z] #:: -20..1.0Inf.
```

Η τελευταία χρήση του τελεστή #:: ορίζει σαν πεδίο τιμών το διάστημα από -20 έως και +∞ (άπειρο).

Αν απουσιάζει η δήλωση του πεδίου μιας μεταβλητής, αλλά αυτή χρησιμοποιείται σε ένα περιορισμό, τότε αυτόματα μετατρέπεται σε μεταβλητή περιορισμών, με το κατάλληλο πεδίο. Για παράδειγμα:

```
?- X #> 0, X #< 10.
X = X{1..9}
Yes
```

Περιορισμοί Ακέραιων Εκφράσεων

Οι περιορισμοί για ακέραιες εκφράσεις της βιβλιοθήκης ic, περιλαμβάνουν τις κλασικές σχέσεις ισότητας και ανισότητας, στα σύμβολα των οποίων υπάρχει πάντα ως πρόθεμα η δίεση (#), όπως εμφανίζονται στον Πίνακα 12.1:

Πίνακας 12.1: *Περιορισμοί Ακέραιων Εκφράσεων*

Περιορισμός	**Εξήγηση**
ExprX #= ExprY	Ισότητα
ExprX #>= ExprY	Μεγαλύτερο ή ίσο
ExprX #=< ExprY	Μικρότερο ή ίσο
ExprX #> ExprY	Μεγαλύτερο
ExprX #< ExprY	Μικρότερο
ExprX #\= ExprY	Ανισότητα

Στους παραπάνω περιορισμούς τα ExprX και ExprY μπορούν να είναι αριθμητικές εκφράσεις, οι οποίες στην πλέον απλή και συνήθη μορφή τους περιέχουν αριθμητικές σταθερές, μεταβλητές περιορισμών και τις συνήθεις πράξεις (+,-,*,/, κλπ):

```
?- [X,Y,Z] #:: [1..10], X #< Y + Z, Z #> Y, Z #= X + 3, X + Y + Z #=
25.
X = X{[6,7]}
Y = Y{[8,9]}
Z = Z{[9,10]}
yes
```

Εκτός από την παραπάνω συνήθη μορφή όρων στους περιορισμούς, η ECLiPSe Prolog υποστηρίζει και περισσότερο πολύπλοκες μορφές όρων μαθηματικών εκφράσεων. Ενδεικτικά αναφέρονται:

- min(E1,E2), max(E1,E2), όπου ο όρος αποτιμάται στο ελάχιστο ή στο μέγιστο μεταξύ των εκφράσεων E1 και E2,
- sqr(Expr), sqrt(Expr), exp(Expr), ln(Expr), το τετράγωνο, η τετραγωνική ρίζα, το εκθετικό και ο λογάριθμος της έκφρασης Expr αντίστοιχα,
- abs(Expr), η απόλυτη τιμή της έκφρασης Expr,
- sum(ExprList), το άθροισμα μιας λίστας εκφράσεων,
- min(ExprList), max(ExprList), το ελάχιστο και το μέγιστο μιας λίστας εκφράσεων αντίστοιχα.

Για παράδειγμα, η σχέση που διέπει το μήκος των πλευρών ενός ορθογωνίου τριγώνου (πυθαγόρειο θεώρημα) μπορεί να εκφραστεί ως:

```
?- X #> 0, Y #> 0, Z #> 0, sqr(X) #= sqr(Y) + sqr(Z).
```

Ενδιαφέρον παρουσιάζουν οι όροι μεγίστου ελαχίστου. Στην ερώτηση:

```
?- Z #:: [4..8], [X,Y] #:: [1..10], W #= X + min(Z,Y).
Z = Z{4..8}
X = X{1..10}
Y = Y{1..10}
W = W{2..18}
yes
```

η μεταβλητή W, έχει πεδίο από 2 έως και 18, καθώς στην τρέχουσα κατάσταση των πεδίων, η ελάχιστη τιμή της X είναι 1, και η ελάχιστη τιμή των Y, Z είναι το 1, ενώ η μέγιστη πιθανή τιμή του είναι το 18, καθώς το X μπορεί να πάρει την τιμή 10 και η ελάχιστη μέγιστη τιμή των Y, Z είναι το 8. Εδώ θα πρέπει να τονιστεί ιδιαίτερα ότι τα πεδία μεταβάλλονται δυναμικά κατά την εκτέλεση του προγράμματος. Κατά συνέπεια, αν σε μεταγενέστερο υπολογιστικό βήμα το πεδίο της Y μεταβληθεί, αντίστοιχες αλλαγές θα γίνουν και στο πεδίο της W. Για παράδειγμα στην ακόλουθη ερώτηση, ο περιορισμός Y #< 4 που εισήχθηκε αργότερα στον υπολογισμό, επιδρά αντίστοιχα στο πεδίο της W:

```
?- Z #:: [4..8], [X,Y] #:: [1..10], W #= X + min(Z,Y), Y #< 4.
Z = Z{4..8}
X = X{1..10}
Y = Y{1..3}
W = W{2..13}
Yes
```

Ο όρος max/1, δέχεται μια λίστα εκφράσεων και έχει παρόμοια συμπεριφορά:

```
?- [X,Y,Z] #:: [1..10], W #= max([3 + X,4 + Y,10 + Z]).
X = X{1..10} Y = Y{1..10}
Z = Z{1..10} W = W{11..20}
Yes
```

Οι παραπάνω περιορισμοί επιτρέπουν να μοντελοποιήσουμε πολύπλοκα προβλήματα εύκολα και με δηλωτικό τρόπο.

Εύρεση Λύσης σε Ακέραια Πεδία

Στο εισαγωγικό παράδειγμα παρουσιάστηκαν συνοπτικά δύο κατηγορήματα που αφορούν στην διαδικασία ανάθεσης τιμών στις μεταβλητές:

- indomain(X): Η μεταβλητή X είναι μια μεταβλητή περιορισμών. Η κλήση indomain(X) αναθέτει στην X μια από τις τιμές του πεδίου της, ενώ ταυτόχρονα ενημερώνει όλες τις μεταβλητές που συμμετέχουν σε περιορισμούς με την X.
- labeling(List): Η λίστα List περιλαμβάνει ένα σύνολο μεταβλητών περιορισμών. Η κλήση στην labeling/1 επιχειρεί να βρει μια ανάθεση τιμών στις μεταβλητές της List (λύση). Καθώς το

κατηγόρημα χρησιμοποιεί κλασική αναδρομή σε λίστες, το labeling/1 υλοποιεί την ανάθεση τιμών με αναζήτηση κατά βάθος.

Συνήθως σε απλά προβλήματα καλούμε το κατηγόρημα labeling/1 σε όλες τις μεταβλητές του προβλήματος.

Δομή ενός Προγράμματος Περιορισμών

Έχοντας εξετάσει όλα τα παραπάνω, η μετατροπή του εισαγωγικού παραδείγματος που ήταν υλοποιημένο σε απλή Prolog, μετατρέπεται εύκολα σε CLP, όπως φαίνεται στον Πίνακα 12.2.

Πίνακας 12.2: *Οι δύο εκδόσεις (Prolog - CLP) του παραδείγματος*

Prolog	CLP (ic βιβλιοθήκη)

`solve(X,Y,Z):-`
`  member(X, [1,2,3,4,5,6,7,8,9,10]),`
`  member(Y, [1,2,3,4,5,6,7,8,9,10]),`
`  member(Z, [4,5,6,7,8,9,10,11]),`
`  X > 3, X < 9,`
`  Y > 2, Y < 9,`
`  X =:= Y + 1,`
`  Z > 5, Z < X.`
	`solve_clp(X,Y,Z):-`
	`  [X,Y] #:: [1..10],`
	`  Z #:: [4..11],`
	`  X #> 3, X #< 9,`
	`  Y #> 2, Y #< 9,`
	`  X #= Y + 1,`
	`  Z #> 5, Z #< X,`
	`  labeling([X,Y,Z]).`

Όπως φαίνεται στον πίνακα, οι κλήσεις στο κατηγόρημα member/2 έχουν αντικατασταθεί με μετατροπή των μεταβλητών σε μεταβλητές περιορισμών με δήλωση των αντίστοιχων πεδίων τιμών και οι κλασικές ανισο-ισότητες με κατάλληλους ακέραιους περιορισμούς. Ιδιαίτερη προσοχή θα πρέπει να δοθεί στο γεγονός, ότι στο Prolog πρόγραμμα γίνεται πρώτα η ανάθεση τιμών στις μεταβλητές και έπειτα ο έλεγχος των περιορισμών, ενώ στο CLP πρόγραμμα, δηλώνονται τα πεδία και οι περιορισμοί και έπειτα γίνεται η ανάθεση τιμών από το κατηγόρημα labeling/1.

Το παραπάνω απλό παράδειγμα δίνει και τη τυπική μορφή ενός CLP προγράμματος:

- Δήλωση των μεταβλητών και των αντίστοιχων πεδίων τους.
- Δήλωση των περιορισμών στις προηγούμενες μεταβλητές.
- Αναζήτηση λύσης.

12.3 Αναζήτηση σε μεγάλα προβλήματα: Το κατηγόρημα search/6

Ο κλασικός αλγόριθμος αναζήτησης κατά βάθος, μπορεί να απαιτήσει μεγάλο χρόνο εκτέλεσης σε προβλήματα μεγάλου μεγέθους. Το μέγεθος του δένδρου αναζήτησης επηρεάζεται από:

- το ποια θα είναι η επόμενη μεταβλητή στην οποία θα ανατεθεί τιμή (variable ordering),
- ποια τιμή θα ανατεθεί σε αυτή τη μεταβλητή (value ordering), και
- ποιος αλγόριθμος αναζήτησης θα χρησιμοποιηθεί.

Καθώς στη βιβλιογραφία έχει προταθεί ένας μεγάλος αριθμός αλγορίθμων αναζήτησης και ευρετικών μεθόδων επιλογής μεταβλητών και τιμών για τις μεταβλητές, η ECLiPSe περιλαμβάνει ένα γενικό κατηγόρημα αναζήτησης, το οποίο παραμετροποιείται ως προς τα τρία παραπάνω σημεία, ώστε να επιτρέπει την εφαρμογή αρκετών περισσότερο πολύπλοκων και αποδοτικών αλγορίθμων. Το κατηγόρημα αυτό είναι το search/6:

```
search(L, Arg, Select, Choice, Method, Option)
```

όπου:

- L, είναι η λίστα μεταβλητών περιορισμών στις οποίες θα ανατεθούν τιμές
- Arg, μια αριθμητική τιμή η οποία καθορίζει σε ποιο όρισμα των όρων της λίστας L είναι οι μεταβλητές περιορισμών προς ανάθεση. Χρησιμοποιείται στην περίπτωση όπου έχουμε σύνθετη

αναπαράσταση των στοιχείων του προβλήματος. Όταν η λίστα L περιέχει μόνο μεταβλητές, η τιμή είναι μηδέν (συνηθέστερη περίπτωση).

- Select, ένα άτομο το οποίο καθορίζει τον τρόπο με τον οποίο θα γίνει η επιλογή μεταβλητής για ανάθεση τιμής (variable ordering). Μερικές από τις διαθέσιμες τιμές του ορίσματος είναι input_order, με την οποία επιλέγεται πάντα η πρώτη ελεύθερη μεταβλητή στη λίστα, first_fail, με την οποία επιλέγεται η μεταβλητή με το μικρότερο σε μέγεθος πεδίο τιμών, smallest με την οποία επιλέγεται η μεταβλητή με την μικρότερη τιμή στο πεδίο της, και αρκετές άλλες.
- Choice, ένα άτομο το οποίο καθορίζει το ποια τιμή θα επιλεγεί από το πεδίο της μεταβλητής (value ordering). Και στην περίπτωση αυτή, υπάρχουν πολλές επιλογές, όπως για παράδειγμα indomain, το οποίο ορίζει ότι θα επιλεγούν διαδοχικά οι τιμές με αύξουσα σειρά, τα indomain_max, indomain_min, indomain_middle, όπου επιλέγονται τιμές με φθίνουσα, αύξουσα και από το "μέσο" του πεδίου, και αρκετές άλλες.
- Method, ένα άτομο το οποίο καθορίζει την μέθοδο (αλγόριθμο) αναζήτησης. Οι διαθέσιμες επιλογές είναι αρκετές και ξεφεύγουν από τα πλαίσια του παρόντος συγγράμματος. Ενδεικτικά αναφέρουμε την μέθοδο complete, που χρησιμοποιεί πλήρη αναζήτηση εξερευνώντας όλες τις διαθέσιμες εναλλακτικές και τη μέθοδο bbs(Steps:integer), (bounded backtracking search) που η αναζήτηση σταματά μετά από προκαθορισμένο (Steps) αριθμό οπισθοδρομήσεων.
- Option, μια λίστα από επιλογές που αφορούν στην αναζήτηση.

Το κατηγόρημα μπορεί να χρησιμοποιηθεί αντί του κατηγορήματος labeling/1, όπως φαίνεται στο ακόλουθο παράδειγμα:

```
solve_clp_ff(X,Y,Z):-
  [X,Y] #:: [1..10],
  Z #:: [4..11],
  X #> 3, X #< 9,
  Y #> 2, Y #< 9,
  X #= Y + 1,
  Z #> 5, Z #< X,
  search([X,Y,Z], 0, first_fail, indomain, complete,[]).
```

Στο παραπάνω, η κλασική αναζήτηση τιμών για τις μεταβλητές X, Y, Z αντικαθίσταται με ευρετική αναζήτηση (first fail), εξετάζοντας τις τιμές των πεδίων των μεταβλητών με την σειρά και χρησιμοποιώντας πλήρη αναζήτηση. Φυσικά, σε τόσο μικρά προβλήματα η διαφορά στην απόδοση είναι αμελητέα, όμως η χρήση των παραπάνω εναλλακτικών μεθόδων αναζήτησης ανάλογα με το μέγεθος του προβλήματος και την δομή του μπορεί να βελτιώσει σημαντικά την απόδοση του προγράμματος, όπως στο πρόγραμμα των N Βασιλισσών που παρουσιάζεται στην αντίστοιχη ενότητα αυτού του κεφαλαίου.

12.4 Περιορισμοί σε λίστες Ακεραίων

Η βιβλιοθήκη ic_global προσφέρει ένα πλήθος περιορισμών που αφορούν λίστες μεταβλητών περιορισμών. Οι περιορισμοί της βιβλιοθήκης έχουν μεγάλη πρακτική σημασία, καθώς επιτρέπουν την εύκολη μοντελοποίηση προβλημάτων. Μερικοί από αυτούς, είναι

- alldifferent(List), ο οποίος επιβάλλει στις μεταβλητές της λίστας List να έχουν διαφορετικές τιμές.
- maxlist(List,Max), ο οποίος επιβάλει η μεταβλητή Max να πάρει την μέγιστη τιμή από τις μεταβλητές περιορισμών της λίστας List,
- minlist(List,Min), αντίστοιχος με τον παραπάνω, με την μεταβλητή Min να έχει την μικρότερη τιμή, από τις τιμές των μεταβλητών της λίστας List.
- sumlist(List,Sum), το άθροισμα των τιμών των μεταβλητών της λίστας List, είναι η τιμή της Sum,
- atmost(N,List,Value), δηλώνει ότι το πολύ N μεταβλητές της λίστας List, μπορούν να πάρουν την τιμή Value,

- atleast(N,List,Value), δηλώνει τον περιορισμό ότι τουλάχιστον N μεταβλητές της λίστας List, θα πρέπει να έχουν την τιμή Value.

Για την χρήση των παραπάνω περιορισμών, είναι απαραίτητη η "φόρτωση" της βιβλιοθήκης ic_global με μια ερώτηση ?-lib(ic_global) στη γραμμή εντολών ή μια ντιρεκτίβα :-lib(ic_global) στο αρχείο. Για παράδειγμα, στην ακόλουθη ερώτηση, στη μεταβλητή W δίνεται η μεγαλύτερη τιμή των τριών μεταβλητών X, Y, Z.

```
?-[X,Y,Z]#::
[1..10],X#>Y,Y#>Z,ic_global:maxlist([X,Y,Z],W),labeling([X,Y,Z]).
X = 3
Y = 2
Z = 1
W = 3
Yes
```

Θα πρέπει να σημειωθεί ότι η ανάθεση της τιμής στην μεταβλητή W γίνεται δυναμικά, δηλαδή κατά τη διάρκεια της εκτέλεσης το πεδίο της W, διαμορφώνεται ανάλογα, μέχρις ότου να προκύψει με ασφάλεια ποια είναι η μέγιστη τιμή. Έτσι στην ερώτηση:

```
?- [X,Y,Z] #:: [1..10], X#>3, Y#<5, ic_global:maxlist([X,Y,Z],Max).
X = X{4..10}
Y = Y{1..4}
Z = Z{1..10}
Max = Max{4..10}
```

το πεδίο της Max διαμορφώνεται ανάλογα. Καθώς η Max είναι μεταβλητή περιορισμών, είναι δυνατό με τη σειρά της να συμμετέχει σε περιορισμούς:

```
?- [X,Y,Z] #:: [1..10], X#>3, Y#<5, ic_global:maxlist([X,Y,Z],Max),
Max #< 8.
X = X{4..7}
Y = Y{1..4}
Z = Z{1..7}
Max = Max{4..7}
```

Από το παραπάνω φαίνεται ότι το σύστημα λόγω του περιορισμού Max #<8, μείωσε αντίστοιχα τα πεδία των X, Y, Z, καθώς εφόσον η μέγιστη τιμή αυτών των μεταβλητών πρέπει να είναι μικρότερη του 8.

Στις παραπάνω ερωτήσεις ο περιορισμός maxlist/2 είχε το πρόθεμα "ic_global:", το οποίο δηλώνει απλά ότι χρησιμοποιείται η έκδοση του maxlist η οποία περιέχεται στη συγκεκριμένη βιβλιοθήκη. Περισσότερες λεπτομέρειες για τις βιβλιοθήκες δίνονται στην ενότητα 12.10.

Ο περιορισμός atmost/3 εφαρμόζεται σε μια λίστα μεταβλητών περιορισμών και θέτει ένα άνω όριο ως προς την εμφάνιση μιας ακέραιας τιμής στη λύση του προβλήματος περιορισμών. Ένα απλό παράδειγμα είναι η ακόλουθη ερώτηση, όπου η λύση (ανάθεση τιμών) για τις μεταβλητές X και Y, επιτρέπεται να έχει την τιμή 2 μόνο το πολύ μια φορά:

```
?- [X,Y] #:: [2..5], ic_global:atmost(1,[X,Y],2), labeling([X,Y]).
X = 2
Y = 3
Yes (0.00s cpu, solution 1, maybe more)
```

Αντίστοιχη λειτουργία έχει και ο περιορισμός atleast/3.

Ο περιορισμός element/3

Ο περιορισμός element/3 αφορά στη μοντελοποίηση περιορισμών πινάκων (tabular constraints). Η μορφή του περιορισμού είναι:

```
element(Index,List,Value)
```

και δηλώνει ότι η τιμή της μεταβλητής περιορισμών Value είναι η τιμή της λίστας ακεραίων List της θέσης Index (η νιοστή τιμή, όπου ν η Index). Τόσο η Index όσο και η Value είναι μεταβλητές περιορισμών, με την πρώτη να έχει σαν πεδίο τιμών το [1..Len] με Len το μέγεθος της λίστας List και την δεύτερη τις ακέραιες τιμές που εμφανίζονται στη λίστα. Για παράδειγμα:

```
?- element(I, [10, 20, 30], V).
I = I{1..3}
V = V{[10, 20, 30]}
Yes
```

Όπως φαίνεται στην παραπάνω ερώτηση, οι πιθανές τιμές της μεταβλητής I είναι από 1 έως και 3, όσες δηλαδή οι θέσεις του πίνακα (λίστα) και το πεδίο της μεταβλητής V είναι οι τιμές του πίνακα. Οι δύο μεταβλητές είναι συνδεδεμένες μέσω του περιορισμού, όπως φαίνεται στα ακόλουθα ερωτήματα:

```
?- element(I, [10, 20, 30], V), V = 30.
I = 3
V = 30
Yes
?- element(I, [10, 20, 30], V), I = 2.
I = 2
V = 20
Yes
```

Δηλαδή, οποιαδήποτε ανάθεση τιμών στην μία, οδηγεί στην αντίστοιχη ανάθεση τιμής στην άλλη μεταβλητή. Προφανώς, μεταβολές στο πεδίο οποιασδήποτε εκ των δύο μεταβλητών έχει αντίστοιχη επίδραση στο πεδίο της άλλης:

```
?- element(I, [10, 20, 30], V), I #> 1.
I = I{[2, 3]}
V = V{[20, 30]}
Yes
```

Ο περιορισμός είναι ιδιαίτερα χρήσιμος σε κάποιες κατηγορίες προβλημάτων. Κλασικό παράδειγμα τέτοιας κατηγορίας είναι τα προβλήματα ανάθεσης εργασιών (assignment problems), τα οποία αφορούν στην ανάθεση N εργασιών (tasks) σε N πράκτορες (agents). Κάθε πράκτορας μπορεί να αναλάβει μια εργασία, και οι πράκτορες έχουν διαφορετικό κόστος για την εκτέλεση μιας εργασίας, δηλαδή αν ο πράκτορας A εκτελέσει την εργασία 1, έχει κόστος 10, ενώ αν ο πράκτορας B εκτελέσει την ίδια εργασία, έχει κόστος 90.

Για παράδειγμα, έστω ότι σε ένα έργο πληροφορικής υπάρχουν 5 εργασίες που πρέπει να γίνουν και 5 ομάδες προγραμματιστών (πράκτορες). Κάθε ομάδα είναι ικανή να εκτελέσει ένα υποσύνολο από τις διαθέσιμες εργασίες και προφανώς σε κάθε ομάδα πρέπει να ανατεθεί μια εργασία. Το κόστος εκτέλεσης της κάθε εργασίας από μια ομάδα και η ικανότητα της ομάδας να εκτελέσει μια εργασία φαίνεται στον Πίνακα 12.3. Σε κάθε γραμμή του πίνακα σημειώνεται ποιες εργασίες και με ποιο κόστος μπορεί να εκτελέσει κάθε ομάδα. Τα κενά κελιά του πίνακα δηλώνουν ότι η συγκεκριμένη εργασία δεν μπορεί να γίνει από την αντίστοιχη ομάδα. Το ζητούμενο είναι να βρεθεί η ανάθεση των εργασιών στις ομάδες και το συνολικό κόστος της ανάθεσης.

Πίνακας 12.3: *Το πρόβλημα Ανάθεσης Εργασιών*

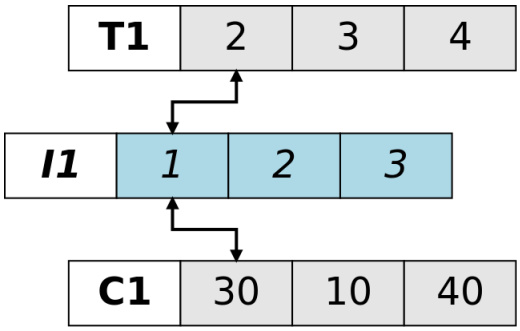
Ομάδα	**Κόστος Εργασίας 1**	**Κόστος Εργασίας 2**	**Κόστος Εργασίας 3**	**Κόστος Εργασίας 4**	**Κόστος Εργασίας 5**
1	-	30	10	40	-
2	70	-	-	20	140
3	-	10	-	80	100
4	20	-	40	40	-
5	40	10	-	10	-

Εφόσον υπάρχουν πέντε συνολικά ομάδες και σε κάθε ομάδα πρέπει να ανατεθεί μια εργασία, κάθε ομάδα μοντελοποιείται με μία μεταβλητή περιορισμών. Κατά συνέπεια η ομάδα 1 μοντελοποιείται με την μεταβλητή T1, της οποίας το πεδίο τιμών θα είναι το [2, 3, 4], δηλαδή η τιμή της μεταβλητής T1 θα δηλώνει ποια εργασία θα πάρει η ομάδα. Καθώς η ανάληψη εργασίας από κάθε ομάδα έχει διαφορετικό κόστος, η μοντελοποίηση του προβλήματος θα πρέπει να περιλαμβάνει και μια μεταβλητή η οποία θα εκφράζει το κόστος ανάληψης της εργασίας. Έτσι, για παράδειγμα, η μεταβλητή C1 μοντελοποιεί το κόστος ανάληψης μιας εργασίας από την ομάδα 1, με πεδίο τιμών το [30, 10, 40], δηλαδή τα κόστη των εργασιών [2, 3, 4], αντίστοιχα.

Προφανώς, αν η μεταβλητή T1 πάρει την τιμή 2, η C1 θα πρέπει να πάρει την τιμή 30, κοκ. Δηλαδή, αν η μεταβλητή T1 πάρει την πρώτη τιμή της λίστας [2, 3, 4] θα πρέπει η μεταβλητή C1 να πάρει αντίστοιχα την πρώτη τιμή της λίστας [30, 10, 40]. Η σχέση αυτή των μεταβλητών υλοποιείται μέσω του περιορισμού element/3:

```
element(I1,[2,3,4],T1),
element(I1,[30,10,40],C1),
```

Η κοινή μεταβλητή I1 στους παραπάνω περιορισμούς εξασφαλίζει την "σύνδεση" των δύο μεταβλητών, όπως φαίνεται στο Σχήμα 12.2. Έτσι αν η μεταβλητή T1 πάρει την τιμή 2, τότε λόγω του πρώτου περιορισμού element/3 η τιμή της I1 θα γίνει 1, και κατά συνέπεια λόγω του δεύτερου περιορισμού element/3 η τιμή της C1 θα γίνει 30.



Σχήμα 12.2: *Η σύνδεση των μεταβλητών T1 και C1, μέσω του περιορισμού element/3*

Έχοντας διατυπώσει τη σχέση μεταξύ εργασίας και κόστους για κάθε ομάδα, αυτό που απομένει είναι να εξασφαλίσουμε ότι θα ανατεθούν διαφορετικές εργασίες σε κάθε ομάδα. Αυτό επιτυγχάνεται επιβάλλοντας στις μεταβλητές T1, T2, ... T5 να έχουν διαφορετικές τιμές με το περιορισμό alldifferent/1. Το τελικό πρόγραμμα είναι το ακόλουθο:

```
:-lib(ic).

assign_tasks(Teams,Cost):-
  Teams = [T1,T2,T3,T4,T5],
     % περιορισμοί για την ομάδα 1
  element(I1,[2,3,4],T1),
  element(I1,[30,10,40],C1),
     % ... για την ομάδα 2
  element(I2,[1,4,5],T2),
  element(I2,[70,20,140],C2),
     % ... για την ομάδα 3
  element(I3,[2,4,5],T3),
  element(I3,[10,80,100],C3),
     % ... για την ομάδα 4
  element(I4,[1, 3,4],T4),
  element(I4,[20,40,40],C4),
     % ... για την ομάδα 5
  element(I5,[1,2,4],T5),
  element(I5,[40,10,10],C5),
     % διαφορετικές εργασίες σε κάθε ομάδα
```

```
    alldifferent(Teams),
       % συνολικό κόστος
    Cost #= C1 + C2 + C3 + C4 + C5,
       % ανάθεση τιμών
    labeling(Teams).
```

Η εκτέλεση του στην ECLiPSe Prolog φαίνεται στο ακόλουθο ερώτημα:

```
?- assign_tasks(Teams, Cost).
Teams = [2, 1, 5, 3, 4]
Cost = 250
Yes (0.00s cpu, solution 1, maybe more)
Teams = [2, 4, 5, 3, 1]
Cost = 230
Yes (0.01s cpu, solution 2, maybe more)
Teams = [2, 5, 4, 3, 1]
Cost = 330
Yes (0.01s cpu, solution 3, maybe more)
Teams = [3, 1, 5, 4, 2]
Cost = 230
Yes (0.02s cpu, solution 4, maybe more)
Teams = [3, 4, 5, 1, 2]
Cost = 160
Yes (0.05s cpu, solution 5, maybe more)
...
```

Όπως φαίνεται παραπάνω, παράγονται εναλλακτικές λύσεις με διαφορετικό κόστος. Στην συνηθέστερη περίπτωση αναζητούμε την ανάθεση εργασιών η οποία ελαχιστοποιεί το συνολικό κόστος. Μια τέτοια λύση βρίσκεται με την χρήση του αλγορίθμου αναζήτησης επέκτασης και οριοθέτησης (branch and bound).

12.5 Αναζήτηση Βέλτιστης Λύσης

Σε πολλές περιπτώσεις συνδυαστικών προβλημάτων, το ζητούμενο είναι να βρεθεί η βέλτιστη λύση σε σχέση με κάποιο κριτήριο. Για παράδειγμα, στο πρόβλημα της ανάθεσης εργασιών που εξετάστηκε στην προηγούμενη παράγραφο, συνήθως αναζητούμε τη λύση η οποία ελαχιστοποιεί στο συνολικό κόστος. Καθώς ο αριθμός των λύσεων σε αυτές τις περιπτώσεις είναι μεγάλος, είναι απαραίτητο να μειώνεται ο χώρος αναζήτησης ώστε να βρίσκεται η βέλτιστη λύση σε συντομότερο χρονικό διάστημα. Ο **αλγόριθμός επέκτασης και οριοθέτησης** (branch and bound) επιτυγχάνει τη μείωση δένδρου κλαδεύοντας μονοπάτια για τα οποία είναι βέβαιο ότι δεν μπορούν να οδηγήσουν σε βέλτιστη λύση.

Η λειτουργία του αλγορίθμου είναι η ακόλουθη:

- Βρίσκεται μια λύση χρησιμοποιώντας ένα τυφλό αλγόριθμο αναζήτησης και αποθηκεύεται μαζί με το κόστος της ως η τρέχουσα καλύτερη λύση.
- Η αναζήτηση συνεχίζεται "κλαδεύοντας" κάθε μερική λύση (ανάθεση τιμών σε υποσύνολο των μεταβλητών του προβλήματος), της οποίας το κόστος είναι μεγαλύτερο από το κόστος της τρέχουσας καλύτερης λύσης.
- Σε περίπτωση που βρεθεί μια πλήρης λύση (ανάθεση τιμών σε όλες τις μεταβλητές) με κόστος μικρότερο εκείνου της τρέχουσας καλύτερης, τότε η νέα αυτή λύση αποθηκεύεται ως η τρέχουσα καλύτερη.
- Η αναζήτηση σταματά όταν εξαντληθεί ο χώρος αναζήτησης.

Η ECLiPSe Prolog προσφέρει την βιβλιοθήκη branch_and_bound, η οποία παρέχει την υλοποίηση του παραπάνω αλγορίθμου. Το σημαντικότερο κατηγόρημα της βιβλιοθήκης είναι το

```
bb_min(+Goal, ?Cost, ?Options)
```

το οποίο βρίσκει εκείνη την λύση για το κατηγόρημα Goal, η οποία ελαχιστοποιεί την τιμή της μεταβλητής Cost. Το κατηγόρημα ανήκει στην κατηγορία των κατηγορημάτων ανώτερης τάξης, καθώς το όρισμα Goal

είναι ένας υποστόχος, ο οποίος στη συνηθέστερη περίπτωση αποτελεί στόχο ανάθεσης τιμών (πχ. labeling/1). Το τρίτο όρισμα Options δίνει την δυνατότητα να οριστούν διάφορες επιλογές σχετικές με τον αλγόριθμο αναζήτησης branch and bound, η επεξήγηση των οποίων ξεφεύγει από τα πλαίσια του παρόντος συγγράμματος. Με βάση τα παραπάνω, ο τελευταίος υποστόχος του κατηγορήματος assign_tasks/2 γίνεται:

```
bb_min(labeling(Teams),Cost,_).
```

Η δηλωτική περιγραφή του παραπάνω είναι "βρες μια ανάθεση τιμών στις μεταβλητές της λίστας Teams τέτοια ώστε να ελαχιστοποιεί την τιμή της μεταβλητής Cost". Ο κώδικας του νέου κατηγορήματος assign_tasks_bb/2, το οποίο βρίσκει τη βέλτιστη λύση, φαίνεται παρακάτω:

```
:-lib(ic).
:-lib(branch_and_bound).

assign_tasks_bb(Teams,Cost):-
  Teams = [T1,T2,T3,T4,T5],
  element(I1,[2,3,4],T1),
  element(I1,[30,10,40],C1),
     % ... (other constraints)
  element(I5,[1,2,4],T5),
  element(I5,[40,10,10],C5),
  alldifferent(Teams),
  Cost #= C1 + C2 + C3 + C4 + C5,
  bb_min(labeling(Teams),Cost,_).
```

Προσοχή θα πρέπει να δοθεί στο γεγονός ότι για να χρησιμοποιηθεί το κατηγόρημα bb_min/3 θα πρέπει να "φορτωθεί" η αντίστοιχη βιβλιοθήκη branch_and_bound. Η εκτέλεση του παραπάνω προγράμματος:

```
?- assign_tasks_bb(Teams, Cost).
Teams = [3, 4, 5, 1, 2]
Cost = 160
Yes (0.00s cpu)
```

επιστρέφει τη βέλτιστη λύση.

12.6 Χρονοπρογραμματισμός

Ο χρονοπρογραμματισμός είναι η διαδικασία ανάθεσης πόρων (resources) σε εργασίες (tasks) σε ένα χρονικό διάστημα, δεδομένων κάποιων περιορισμών και ενός κριτηρίου – στόχου. Ο όρος εργασία μπορεί να αναφέρεται σε διεργασίες υλοποίησης IT έργων, διεργασίες κατασκευής προϊόντων, εκτέλεση προγραμμάτων, μαθήματα, απογειώσεις / προσγειώσεις αεροπλάνων κλπ. Οι εργασίες απαιτούν πόρους όπως για παράδειγμα ομάδες προγραμματιστών, μηχανές παραγωγής, υπολογιστικούς πόρους, αίθουσες μαθημάτων, αεροδιάδρομους, κλπ. Συνήθως υπάρχουν περιορισμοί, όπως για παράδειγμα περιορισμοί καταλληλότητας (μια αίθουσα πρέπει να μπορεί να φιλοξενήσει τους φοιτητές ενός μαθήματος), διάταξης εργασιών, καταληκτικές ημερομηνίες, κλπ. Τέλος, σχεδόν πάντα υπάρχει ένα κριτήριο ως προς το οποίο κρίνεται η ποιότητα του παραγόμενου χρονοπρογράμματος, το οποίο μπορεί να είναι η ελαχιστοποίηση συνολικού χρόνου εκτέλεσης, η ελαχιστοποίηση συνολικού χρόνου αργοπορίας σε σχέση με καταληκτικές ημερομηνίες, η ολοκλήρωση διεργασιών εντός των ορίων της καταληκτικής ημερομηνίας, κλπ. Ο χρονοπρογραμματισμός επιτελεί σημαντικό ρόλο σε πλήθος βιομηχανικών εφαρμογών και για το λόγο αυτό είναι ένα από τα προβλήματα που έχουν μελετηθεί εκτενώς στη βιβλιογραφία.

Ο χρονοπρογραμματισμός αποτελεί μια από τις κλασικές εφαρμογές του προγραμματισμού περιορισμών και όπως θα εξετάσουμε στα ακόλουθα έχουν ενσωματωθεί στις πλατφόρμες λογικού προγραμματισμού με περιορισμούς εξειδικευμένοι περιορισμοί οι οποίοι επιτρέπουν την εύκολη μοντελοποίηση και αποδοτική επίλυση προβλημάτων της κατηγορίας. Τέτοιοι περιορισμοί είναι οι disjunctive/2 και cumulative/4, οι οποίοι θα εξεταστούν παρακάτω μέσω του παραδείγματος που ακολουθεί.

Έστω, ότι υπάρχουν 5 εργασίες και 2 ομάδες πρακτόρων για την διεκπεραίωσή τους. Κάθε εργασία έχει μια προκαθορισμένη διάρκεια και απαιτεί για την υλοποίησής της μια συγκεκριμένη ομάδα. Μεταξύ κάποιων εργασιών υπάρχουν περιορισμοί διάταξης (μερικής) και κάθε ομάδα μπορεί να εκτελέσει μια εργασία σε κάθε

χρονική στιγμή. Το ζητούμενο είναι να βρεθούν οι χρόνοι έναρξης των εργασιών ώστε να ελαχιστοποιείται η διάρκεια του συνολικού έργου. Ο Πίνακας 12.4 παρουσιάζει τις λεπτομέρειες του προβλήματος.

Πίνακας 12.4: *Πληροφορίες Εργασιών*

Εργασία	Διάρκεια	Διάταξη (ολοκλήρωση)	Ομάδα
1	5	πριν από την 2	A
2	4		B
3	7	πριν από την 5	B
4	1		A
5	9		A

Η μοντελοποίηση του προβλήματος απλοποιείται, αν θεωρήσουμε ότι ο χρόνος αναπαριστάται από χρονικές στιγμές, στο διάστημα από το 0 έως και το άπειρο, με τη χρονική στιγμή 0 να σηματοδοτεί την έναρξη του χρονοπρογράμματος. Αν θεωρήσουμε για παράδειγμα ότι St1 είναι η μια μεταβλητή που αναπαριστά τον χρόνο έναρξης της εργασίας 1, τότε η ανάθεση St1 = 0, σημαίνει ότι η εργασία 1 ξεκινά με την έναρξη του χρονοπρογράμματος (χρονική στιγμή 0), ενώ η ανάθεση St1 = 5, σημαίνει ότι η εργασία 1 ξεκινά την πέμπτη χρονική στιγμή (Σχήμα 12.3).

Σχήμα 12.3: *Αναπαράσταση του χρόνου σε προβλήματα χρονοπρογραμματισμού. Στο σχήμα εμφανίζεται μια εργασία με διάρκεια 5 χρονικές στιγμές.*

Κατά συνέπεια, ο χρόνος έναρξης της κάθε εργασίας μοντελοποιείται με μια ακέραια μεταβλητή περιορισμών (St1, St2, κλπ) και ο χρόνος λήξης της κάθε εργασίας είναι το άθροισμα του χρόνου έναρξης και της διάρκειάς της. Για παράδειγμα για την πρώτη εργασία, ισχύει ο περιορισμός:

```
End1 #= 5 + St1
```

όπου End1 είναι ο χρόνος λήξης της εργασίας. Προφανώς, παρόμοιοι περιορισμοί θα εκφράζουν την πληροφορία που αφορά τις υπόλοιπες εργασίες. Έχοντας μοντελοποιήσει την έναρξη και την λήξη των εργασιών με μεταβλητές περιορισμών, οι σχέσεις μερικής διάταξης προκύπτουν ως ανισότητες: ο περιορισμός ότι η εργασία 1 πρέπει να έχει ολοκληρωθεί πριν την έναρξη της εργασίας 2, εκφράζεται:

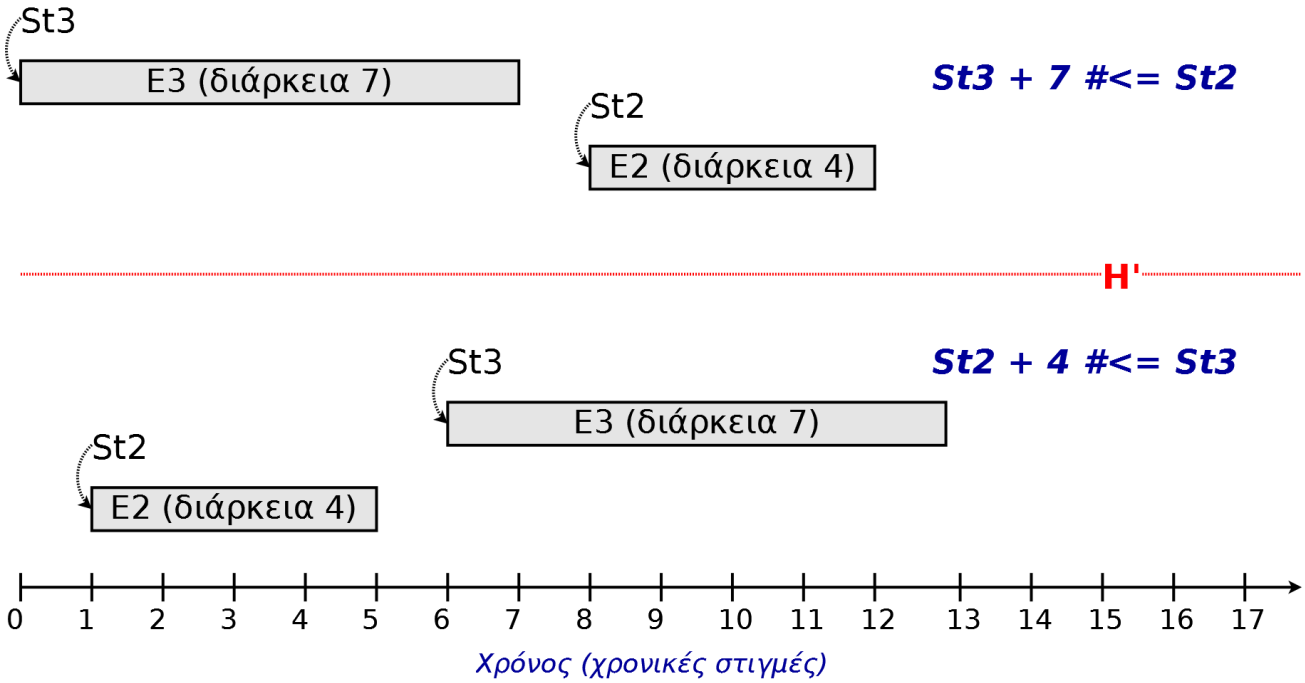
```
End1 #=< St2
```

Το χρονοπρόγραμμα ολοκληρώνεται με την ολοκλήρωση της τελευταίας εργασίας, άρα ο χρόνος λήξης του χρονοπρογράμματος, είναι ο χρόνος λήξης της τελευταίας εργασίας. Καθώς δεν είναι γνωστό ποια εργασία θα τελειώσει τελευταία, θα ορίσουμε ως χρόνο λήξης (MakeSpan) τη μέγιστη τιμή των μεταβλητών χρόνου λήξης των επιμέρους εργασιών End1, End2, ... End5, μέσω του περιορισμού maxlist/2 της βιβλιοθήκης ic_global:

```
ic_global:maxlist([End1,End2,End3,End4,End5],MakeSpan)
```

Ο τελευταίος περιορισμός του προβλήματος αφορά τη διεκπεραίωση των εργασιών από την κάθε ομάδα. Καθώς μια ομάδα μπορεί να εκτελέσει μόνο μια εργασία κάθε φορά, θα πρέπει τα διαστήματα εκτέλεσης των εργασιών της ομάδας να μην επικαλύπτονται, δηλαδή μια εργασία να έχει ολοκληρωθεί πριν την έναρξη της επόμενης εργασίας. Για παράδειγμα, στο πρόβλημα που εξετάζεται, η ομάδα Β αναλαμβάνει την εκτέλεση των εργασιών 2 και 3, και καθώς δεν υπάρχουν περιορισμοί διάταξης ορισμένοι στο πρόβλημα ανάμεσα στις δύο εργασίες, θα πρέπει να ισχύει (Σχήμα 12.4):

```
St2 + 4 #=< St3  ή  St3 + 7 #=< St2
```



Σχήμα 12.4: *Παράδειγμα μη επικάλυψης δύο εργασιών που επιβάλλεται από την διάζευξη St2 + 4 #=< St3 ή St3 + 7 #=< St2*

Όμως, η εισαγωγή διαζεύξεων αυξάνει δραματικά το χρόνο αναζήτησης της λύσης καθώς εισάγει σημεία οπισθοδρόμησης. Καθώς ο περιορισμός εμφανίζεται πάρα πολύ συχνά σε προβλήματα χρονοπρογραμματισμού, έχουν αναπτυχθεί ειδικοί αλγόριθμοι διάδοσης περιορισμών της κατηγορίας, οι οποίοι είναι υλοποιημένοι στην βιβλιοθήκη της ECLiPSe ic_edge_finder. Ένας τέτοιος περιορισμός είναι ο disjunctive/2:

```
disjunctive(StartTimes, Durations)
```

όπου StartTimes είναι μια λίστα με μεταβλητές περιορισμών που μοντελοποιούν τους χρόνους έναρξης των εργασιών, και Durations οι αντίστοιχες διάρκειες των εργασιών (λίστα ακεραίων). Ο περιορισμός επιβάλλει την μη επικάλυψη των εργασιών της λίστας. Για την ομάδα Β, η οποία πρέπει να εκτελέσει τις εργασίες 2 και 3 με χρόνους έναρξης St2, St3 και διάρκειες 4 και 7 αντίστοιχα έχουμε:

```
disjunctive([St2,St3], [4,7])
```

Τέλος, καθώς αναζητούμε εκείνο το χρονοπρόγραμμα το οποίο ελαχιστοποιεί τη συνολική διάρκεια του έργου θα χρησιμοποιήσουμε τον αλγόριθμο επέκτασης και οριοθέτησης με κριτήριο την τιμή της μεταβλητής MakeSpan:

```
bb_min(labeling(Starts), MakeSpan, bb_options{strategy:restart})
```

Ο τελικός κώδικας φαίνεται παρακάτω:

```
%%% Libraries Required.
:-use_module(library(ic)).
:-use_module(library(ic_global)).
:-use_module(library(ic_edge_finder)).
:-use_module(library(branch_and_bound)).
```

```
schedule_start_times(teamA([St1,St4,St5]),teamB([St2,St3]),MakeSpan):
-
  Starts = [St1,St2,St3,St4,St5],
  Ends = [End1,End2,End3,End4,End5],
  Starts #:: 0..inf,
  Ends #:: 0..inf,
     % Start End Times and Durations,
  End1 #= 5 + St1, End2 #= 4 + St2,
  End3 #= 7 + St3, End4 #= 1 + St4,
  End5 #= 9 + St5,
     % Ordering Constraints
  End1 #<= St2, End3 #<= St5,
  ic_global:maxlist(Ends,MakeSpan),
     % Assignment to teams and no overlapping constraints
     % team A
  disjunctive([St1,St4,St5],[5,1,9]),
     % team B
  disjunctive([St2,St3],[4,7]),
     % Search for a Solution
  bb_min(labeling(Starts),MakeSpan,bb_options{strategy:restart}).
```

Η εκτέλεση του οποίου είναι:

```
?- schedule_start_times(T1, T2, M).
T1 = teamA([0, 5, 7])
T2 = teamB([7, 0])
M = 16
Yes (0.00s cpu)
```

Η παραπάνω λύση αντιστοιχεί στο χρονοπρόγραμμα που φαίνεται στο Σχήμα 12.5.

Σχήμα 12.5: *Το χρονοπρόγραμμα του παραδείγματος*

Ο περιορισμός cumulative/4

Σε κάποια προβλήματα χρονοπρογραμματισμού, οι πόροι έχουν συγκεκριμένη χωρητικότητα και οι εργασίες μπορούν να καταναλώσουν μέρος της χωρητικότητας των πόρων αυτών. Έστω για παράδειγμα ότι στο παραπάνω πρόβλημα ότι η ομάδα Α έχει 5 μέλη, και η ομάδα Β έχει 4 μέλη και ότι οι εργασίες απαιτούν για την διεκπεραίωσή τους:

- η εργασία 1, ένα μέλος
- οι εργασίες 2 και 3 από δύο μέλη,
- οι εργασίες 4 και 5 από τρία μέλη

Από τα παραπάνω είναι προφανές ότι οι ομάδες μπορούν να εκτελέσουν περισσότερες από μια εργασίες κάθε χρονική στιγμή. Έτσι η ομάδα Α μπορεί να εκτελέσει ταυτόχρονα τις εργασίες 1 και 4 ή 1 και 5, αλλά όχι τις

εργασίες 4 και 5 καθώς η ταυτόχρονη εκτέλεσή τους απαιτεί 6 μέλη (3+3). Η διατύπωση των περιορισμών στην περίπτωση αυτή είναι πολύπλοκη, απλοποιείται όμως σημαντικά με τον περιορισμό cummulative/4:

```
cumulative(StartTimes, Durations, Resources, ResourceLimit)
```

όπου StartTimes και Durations είναι οι χρόνοι έναρξης των εργασιών και οι αντίστοιχες διάρκειες τους, Resources οι απαιτήσεις σε πόρους της κάθε εργασίας (λίστα ακεραίων) και ResourceLimit η συνολική χωρητικότητα του πόρου. Ο περιορισμός εξασφαλίζει ότι δεν αλληλεπικαλύπτονται εργασίες για τις οποίες το άθροισμα των πόρων ξεπερνά τη συνολική χωρητικότητα. Έτσι για την ομάδα Α, ο περιορισμός disjunctive/4 αντικαθίσταται από τον περιορισμό:

```
cumulative([St1,St4,St5], [5,1,9], [1,3,3], 5)
```

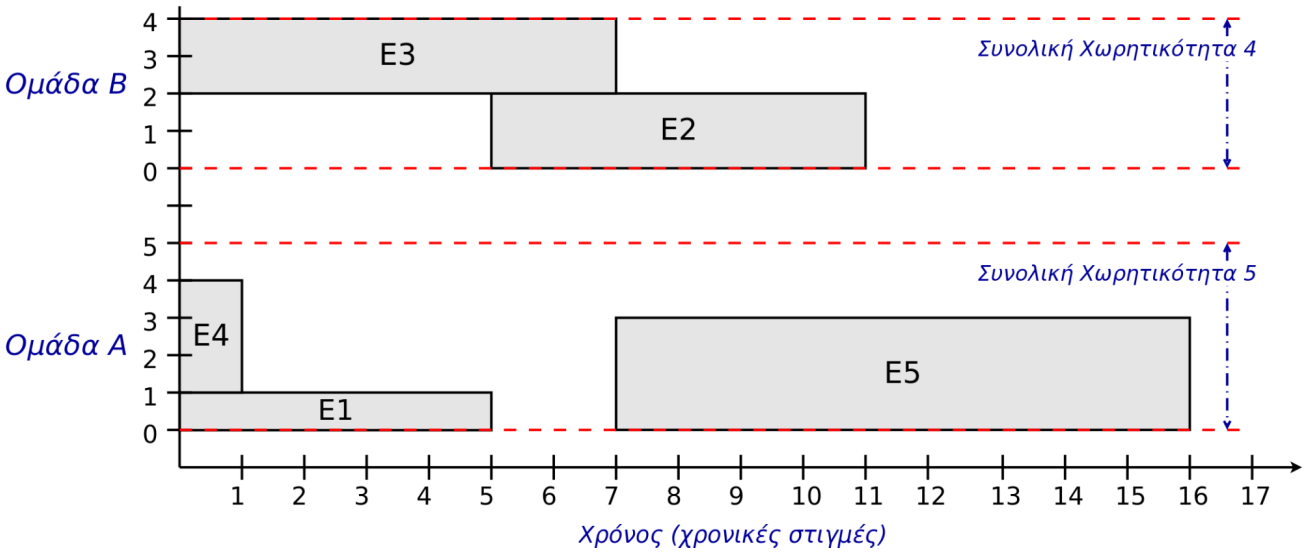
ενώ για την ομάδα Β ο περιορισμός disjunctive/4 αντικαθίσταται από τον:

```
cumulative([St2,St3], [4,7], [2,2], 4)
```

Αυτές είναι και οι μόνες αλλαγές οι οποίες απαιτούνται για το νέο πρόβλημα. Το χρονοπρόγραμμα που προκύπτει δίνεται από την ερώτηση:

```
?- schedule_start_times_r(T1, T2, M).
T1 = teamA([0, 0, 7])
T2 = teamB([5, 0])
M = 16
```

Στο Σχήμα 12.6 παρουσιάζεται η απάντηση. Το πλάτος κάθε εργασίας αναπαριστά την απαίτηση σε πόρους, ενώ οι διακεκομμένες γραμμές την συνολική χωρητικότητα (μέλη) της κάθε ομάδας. Αν και η εργασία 5 θα μπορούσε να εκτελεστεί νωρίτερα στο πρόγραμμα, ο περιορισμός διάταξης ο οποίος επιβάλλει να ξεκινήσει μετά το τέλος της εργασίας 3, δεν επιτρέπει κάτι τέτοιο.



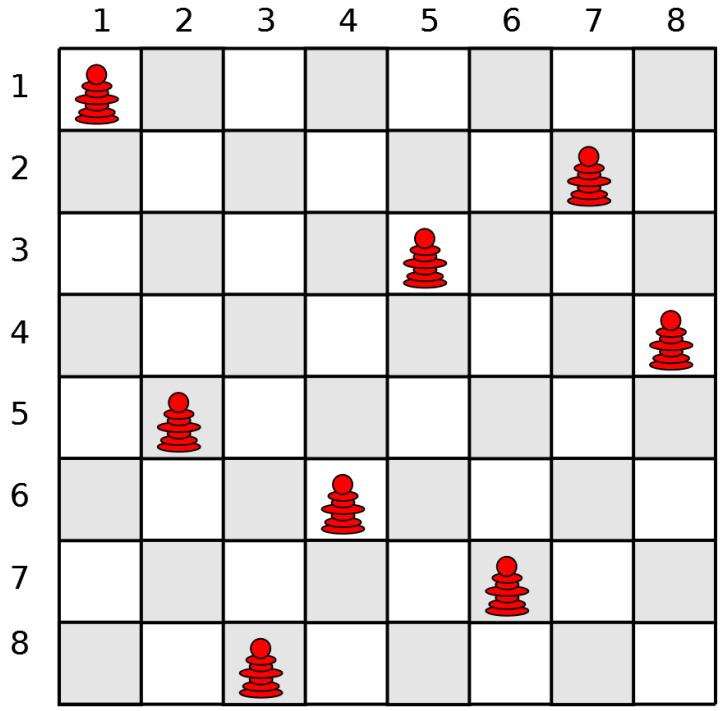
Σχήμα 12.6: *Χρονοπρόγραμμα με πόρους και εργασίες διαφορετικής χωρητικότητας*

Από τα παραπάνω, είναι προφανές ότι προβλήματα χρονοπρογραμματισμού που ανήκουν σε αυτήν την κατηγορία μοντελοποιούνται και επιλύονται πολύ εύκολα με τη χρήση αυτών των εξειδικευμένων αυτών περιορισμών.

12.7 Παράδειγμα Εφαρμογής: Το πρόβλημα των N Βασιλισσών

Το πρόβλημα της τοποθέτησης 8 βασιλισσών σε μια σκακιέρα, το οποίο παρουσιάστηκε στο Κεφάλαιο 8, μπορεί να γενικευτεί σε σκακιέρες οποιουδήποτε μεγέθους. Έτσι, στην γενική του μορφή το πρόβλημα είναι να τοποθετηθούν N βασίλισσες σε μια σκακιέρα NxN, με τέτοιο τρόπο ώστε να μην επιτίθεται η μια στην άλλη. Αποτελεί ένα από τα πλέον γνωστά προγράμματα δοκιμής (benchmark) στον προγραμματισμό με περιορισμούς, και είναι ένα κλασικό συνδυαστικό πρόβλημα, δηλαδή η δυσκολία επίλυσής του αυξάνει εκθετικά όσο μεγαλώνει το μέγεθος της σκακιέρας (N).

Καθώς είναι δεδομένο ότι οι βασίλισσες θα τοποθετηθούν σε διαφορετικές στήλες της σκακιέρας, στη υλοποίηση που ακολουθεί, η λύση είναι μια λίστα N ακεραίων, καθένας από τους οποίους δηλώνει τη γραμμή στην οποία θα τοποθετηθεί η αντίστοιχη βασίλισσα. Για παράδειγμα η λύση [1, 5, 8, 6, 3, 7, 2, 4] για την κλασική σκακιέρα μεγέθους 8, δηλώνει ότι η βασίλισσα της πρώτης στήλης θα τοποθετηθεί στην πρώτη γραμμή, η βασίλισσα της δεύτερης στήλης στη γραμμή 5 κοκ. Η λύση φαίνεται στο Σχήμα 12.7.



Σχήμα 12.7: *Η λύση [1, 5, 8, 6, 3, 7, 2, 4] στο πρόβλημα των 8 βασιλισσών*

Κάθε θέση (γραμμή) μιας βασίλισσας μοντελοποιείται σαν μια μεταβλητή περιορισμών με πεδίο τις τιμές 1..N και οι γραμμές στις οποίες θα τοποθετηθούν πρέπει να είναι διαφορετικές μεταξύ τους. Εφόσον η υλοποίηση που εξετάζεται υποστηρίζει διαφορετικό μέγεθος σκακιέρας N, θα πρέπει να δημιουργείται μια λίστα μεταβλητών μεγέθους N, και να επιβάλλονται οι προηγούμενοι περιορισμοί. Η λίστα N μεταβλητών (Rows) δημιουργείται με το κατηγόρημα length/2:

```
...
length(Rows,N),
Rows #:: 1..N,
alldifferent(Rows),
...
```

Για κάθε βασίλισσα θα πρέπει να εξασφαλιστεί ότι δεν υπάρχει άλλη βασίλισσα σε επόμενες στήλες πάνω στις δύο διαγώνιους της. Αν QueenR είναι είναι η γραμμή μιας βασίλισσας, τότε η γραμμή της αμέσως επόμενης βασίλισσας της λίστας (NextQueenR) δεν μπορεί να πάρει τις τιμές QueenR + 1, QueenR - 1, και η βασίλισσα μετά από αυτή δεν μπορεί να τοποθετηθεί στις γραμμές QueenR + 2, QueenR - 2, κοκ. Η τεχνική που θα χρησιμοποιηθεί εφόσον δεν γνωρίζουμε το μήκος της λίστας είναι η αναδρομή. Παρακάτω παρουσιάζεται το κατηγόρημα apply_constraints/1 και το βοηθητικό κατηγόρημα αυτού safe/3 τα οποία υλοποιούν το παραπάνω:

```
apply_constraints([]).
apply_constraints([QueenR|Rest]):-
  safe(QueenR,Rest,1),
  apply_constraints(Rest).

safe(_,[],_).
safe(QueenR,[NextQueenR|Rest],Pos):-
  NextQueenR #\= QueenR+Pos,
```

```
    NextQueenR #\= QueenR-Pos,
    NewPos is Pos + 1,
    safe(QueenR,Rest,NewPos).
```

Η μεταβλητή Pos στο κατηγόρημα safe/3, ξεκινά από την τιμή 1 και για κάθε επόμενη βασίλισσα αυξάνει κατά 1, επιτρέποντας έτσι την υλοποίηση των περιορισμών που περιγράφηκαν παραπάνω.

Τέλος, θα πρέπει να γίνει ανάθεση τιμών στις μεταβλητές. Η απλή προσέγγιση είναι να χρησιμοποιηθεί το κατηγόρημα labeling/1, όμως η δυσκολία του προβλήματος αυξάνει εκθετικά με το μέγεθος της σκακιέρας και κατά συνέπεια η εφαρμογή ευρετικών τεχνικών αναζήτησης μειώνει δραματικά το χρόνο εύρεσης λύσης. Για να είναι δυνατή η σύγκριση των δύο λύσεων υλοποιείται το βοηθητικό κατηγόρημα search_for_solution/2, όπου το πρώτου του όρισμα (QRows) είναι οι μεταβλητές στις οποίες θα ανατεθούν οι τιμές και το δεύτερο καθορίζει τον τρόπο με τον οποίο θα γίνει η ανάθεση. Οι τιμές του δεύτερου ορίσματος μπορεί να είναι το άτομο labeling, ή μια λίστα δύο τιμών με την πρώτη να καθορίζει την επιλογή μεταβλητής (variable ordering) και τη δεύτερη την επιλογή τιμής (value ordering), όπως περιγράφονται στην ενότητα 12.3:

```
search_for_solution(QRows,labeling):-
  labeling(QRows).

search_for_solution(QRows,[VarOrdering,ValueOrdering]):-
  search(QRows,0,VarOrdering,ValueOrdering,complete, []).
```

Έχοντας τα παραπάνω κατηγορήματα, το κατηγόρημα το οποίο βρίσκει λύση στο πρόβλημα των N βασιλισσών είναι το queens/3, με το πρώτο όρισμα να είναι το μέγεθος της σκακιέρας, το δεύτερο η λύση και το τρίτο ο τρόπος με τον οποίο θα γίνει η ανάθεση τιμών:

```
queens(N,QRows,Strategy):-
  length(QRows,N),
  QRows #:: 1..N,
  alldifferent(QRows),
  apply_constraints(QRows),
  search_for_solution(QRows,Strategy).
```

Και η εύρεση μιας λύσης για σκακιέρα μεγέθους 8x8 χρησιμοποιώντας το κατηγόρημα labeling/1 είναι:

```
?- queens(8, S, labeling).
S = [1, 5, 8, 6, 3, 7, 2, 4]
Yes
```

Τι διαφορά όμως κάνει η χρήση ευρετικών μεθόδων στην αναζήτηση λύσης; Οι ακόλουθες ερωτήσεις αναζητούν λύσεις σε μια σκακιέρα 30x30, στην πρώτη περίπτωση χρησιμοποιώντας το κατηγόρημα labeling/1 και στην δεύτερη χρησιμοποιώντας την ευρετική συνάρτηση first fail για την επιλογή μεταβλητών:

```
?- queens(30, S, labeling).
S = [1, 3, 5, 2, 4, 9, 11, 13, 15, 7, 23, 26, 28, 25, 22, 24, ...]
Yes (178.83s cpu, solution 1, maybe more)

?- queens(30, S, [first_fail, indomain]).
S = [1, 3, 5, 24, 26, 4, 23, 7, 28, 16, 18, 15, 6, 22, 20, ...]
Yes (0.01s cpu, solution 1, maybe more)
```

Όπως φαίνεται στις δύο εκτελέσεις, για μέγεθος σκακιέρας 30x30, η διαφορά στο χρόνο εκτέλεσης είναι δραματική.

12.8 Παράδειγμα Εφαρμογής: Διέλευση Οχημάτων από Γέφυρα

Έξι φορτηγά αυτοκίνητα που έχουν διαφορετικές ταχύτητες και διαφορετικό βάρος πρέπει να περάσουν μια γέφυρα, η οποία μπορεί να αντέξει μέχρι 20 τόνους το πολύ. Τα χαρακτηριστικά των αυτοκινήτων είναι τα ακόλουθα:

- Name, το χαρακτηριστικό όνομα του αυτοκινήτου,

- Speed, ο χρόνος που απαιτείται για να διέλθει το αυτοκίνητο τη γέφυρα εκφρασμένος σε χρονικές στιγμές, και
- Weight, το βάρος του φορτηγού αυτοκινήτου σε τόνους.

Οι πληροφορίες για τα οχήματα δίνονται ως Prolog γεγονότα, όπως φαίνεται στο κώδικα που ακολουθεί, όπου το πρώτο όρισμα του γεγονότος να είναι το όνομά του, το δεύτερο το βάρος του και το τρίτο ο χρόνος διέλευσης:

```
%%% car/3
%%% car(Name, Weight, Speed)
car(alpha, 10, 4).
car(beta, 13,5).
car(gamma, 8, 3).
car(delta, 5, 4).
car(ephilon, 7, 1).
car(zita, 9, 3).
car(eta, 11, 6).
```

Το ζητούμενο είναι να βρεθεί το πότε πρέπει να ξεκινήσει κάθε αυτοκίνητο την διάσχιση, ώστε για να διέλθουν όλα με ασφάλεια την γέφυρα, δηλαδή το άθροισμα των βαρών των αυτοκινήτων που είναι πάνω στην γέφυρα ταυτόχρονα να μην ξεπερνά τους 20 τόνους. Αναζητείται η λύση στην οποία η διέλευση όλων των αυτοκινήτων γίνεται στον ελάχιστο δυνατό χρόνο.

Η μοντελοποίηση γίνεται με απλό τρόπο αν θεωρήσουμε ότι η διέλευση του κάθε αυτοκινήτου είναι μια εργασία χρονοπρογραμματισμού, με διάρκεια ίση με τον χρόνο διέλευσης (Speed), άρα για κάθε αυτοκίνητο θα υπάρχει ένας περιορισμός:

```
S + Speed #= E
```

όπου S ο χρόνος έναρξης της διέλευσης και E ο χρόνος λήξης. Εφόσον η πληροφορία για τα αυτοκίνητα είναι με την μορφή γεγονότων, θα χρησιμοποιήσουμε το κατηγόρημα findall/3 για να συλλέξουμε σε τρεις λίστες την απαραίτητη πληροφορία, όπως φαίνεται στο ακόλουθο μέρος κώδικα:

```
...
findall(C,car(C,_,_),Cars),
findall(W,car(_,W,_),Weights),
findall(S,car(_,_,S),Speeds),
...
```

Εφόσον υπάρχουν N αυτοκίνητα, πρέπει να δημιουργήσουμε N μεταβλητές έναρξης, μια για κάθε αυτοκίνητο, το οποίο εξασφαλίζεται με την χρήση του κατηγορήματος length/2:

```
...
length(Cars,N),
length(Starts,N),
Starts #:: 0..inf,
...
```

Ο περιορισμός Starts #:: 0..inf εξασφαλίζει ότι οι χρόνοι έναρξης θα είναι μεγαλύτεροι ή ίσοι με το 0, καθώς το χρονικό διάστημα ξεκινά με τη χρονική στιγμή 0. Για τον ορισμό των περιορισμών του χρόνου έναρξης και λήξης της κάθε διέλευσης θα χρησιμοποιηθεί αναδρομή, εφόσον οι χρόνοι έναρξης είναι σε λίστα (Starts). Έτσι το κατηγόρημα state_crossing_times/3 θα έχει τρία ορίσματα, με το πρώτο να είναι η λίστα Starts, το δεύτερο η λίστα των χρόνων διέλευσης (Weights) και το τρίτο μια λίστα με τους χρόνους λήξης κάθε εργασίας (Ends):

```
state_crossing_times([],[],[]).
state_crossing_times([S|Starts], [Sp|Speeds], [E|Ends]):-
  S + Sp #= E,
  state_crossing_times(Starts,Speeds,Ends).
```

Όπως και το παράδειγμα του χρονοπρογραμματισμού εργασιών που δόθηκε παραπάνω, ο χρόνος διέλευσης ολόκληρου του στόλου των αυτοκινήτων (MakeSpan) είναι ο χρόνος λήξης της διέλευσης του τελευταίου αυτοκινήτου, άρα:

```
...
maxlist(Ends,MakeSpan),
...
```

Αν θεωρήσουμε ότι η γέφυρα είναι ο μοναδικός πόρος του προβλήματος με χωρητικότητα 20, και κάθε διέλευση καταναλώνει μέρος αυτού του πόρου ανάλογα με το βάρος του οχήματος, τότε η ασφαλής διέλευση όλων των οχημάτων εξασφαλίζεται με τον περιορισμό cumulative/4:

```
...
cumulative(Starts, Speeds, Weights, 20)
...
```

Τέλος, η βέλτιστη λύση είναι εκείνη που ελαχιστοποιεί την μεταβλητή MakeSpan:

```
...
bb_min(labeling(Starts), MakeSpan, bb_options{strategy:restart})
...
```

Ο ολοκληρωμένος κώδικας δίνεται παρακάτω. Το κατηγόρημα state_crossing_times/3 είναι εκείνο που παρουσιάστηκε προηγουμένως:

```
car_crossing(Starts,MakeSpan):-
  findall(C,car(C,_,_),Cars),
  findall(W,car(_,W,_),Weights),
  findall(S,car(_,_,S),Speeds),
  length(Cars, N),
  length(Starts, N),
  Starts #:: 0..inf,

  state_crossing_times(Starts, Speeds, Ends),

  maxlist(Ends, MakeSpan),
  cumulative(Starts, Speeds, Weights, 20),
  bb_min(labeling(Starts), MakeSpan, bb_options{strategy:restart}).
```

Και η εκτέλεση του αντίστοιχου ερωτήματος:

```
?- car_crossing(S, M).
S = [0, 4, 0, 3, 7, 9, 9]
M = 15
Yes (0.01s cpu)
```

όπου τα στοιχεία της λίστας του πρώτου ορίσματος είναι οι χρόνοι έναρξης διέλευσης των οχημάτων alpha, beta, gamma κλπ αντίστοιχα, και M ο χρόνος ολοκλήρωσης της διέλευσης όλων των οχημάτων.

12.9 Περιορισμοί ως Δεδομένα

Σε κάποιες περιπτώσεις είναι χρήσιμο αντί να εισάγουμε ένα περιορισμό σαν κλασικό υποστόχο σε ένα πρόβλημα, να αναθέσουμε την τιμή αλήθειας του σε μια ακέραια μεταβλητή, η οποία έχει πεδίο τιμών το διάστημα 0..1, με 0 να αναπαριστά το λογική τιμή ψευδές (false) και 1 την τιμή αληθές (true). Οι περιορισμοί αυτής της κατηγορίας, αναφέρονται στη βιβλιογραφία ως *περιορισμοί ως δεδομένα* (reified constraints) και είναι παθητικοί (passive) με την έννοια ότι δεν επιφέρουν άμεσα αλλαγές στα πεδία των μεταβλητών. Αυτό που συμβαίνει είναι ότι αν η αλήθεια τους προκύπτει λογικά από την τρέχουσα κατάσταση του προβλήματος, συνεπάγεται ότι η αντίστοιχη μεταβλητή παίρνει την τιμή 1, ενώ αν προκύπτει η αποτυχία του περιορισμού, η μεταβλητή παίρνει την τιμή 0. Για παράδειγμα στην ακόλουθη ερώτηση η μεταβλητή T έχει σαν τιμή τη λογική τιμή του περιορισμού X #> 2:

```
?- X :: 1..10, T #= (X #> 2).
X = X{1..10}
T = T{[0, 1]}
There is 1 delayed goal.
Yes
```

Καθώς από το τρέχον πεδίο της μεταβλητής X δεν προκύπτει αν ο περιορισμός X#>2 είναι απαραίτητα αληθής ή ψευδής η τιμή της T ανήκει στο διάστημα 0..1. Ας σημειωθεί ότι στο πεδίο της X δεν έγινε καμία αλλαγή. Αντίθετα στην ακόλουθη ερώτηση, η μεταβλητή T παίρνει την τιμή 0, καθώς από το τρέχον πεδίο της X συνάγεται ότι ο περιορισμός X#>20 είναι ψευδής:

```
?- X :: 1..10, T #= (X #> 20).
X = X{1..10}
T = 0
Yes (0.00s cpu)
```

Οι reified περιορισμοί μπορούν να συνδυαστούν με τους κλασικούς λογικούς τελεστές σύζευξης (and), διάζευξης (or), άρνησης (neg) και συνεπαγωγής (=>). Για παράδειγμα:

```
?- X #:: [1..10], B #= (X #>= 1 and X #=< 10).
X = X{1..10}
B = 1
Yes (0.00s cpu)
```

Τα λογικά συνδετικά επιτρέπουν την μοντελοποίηση πολύπλοκων περιορισμών. Για παράδειγμα ο περιορισμός element/3:

```
element(I,[20,30,40],V)
```

μπορεί να εκφραστεί σαν ένα σύνολο συνεπαγωγών:

```
(I #=1) => (V#=30),
(V#=30) => (I#=1),
...
(I #=3)=> (V#=40),
(V#=40) => (I#=3),
```

Αν στη reified μεταβλητή αποδοθεί τιμή τότε ο περιορισμός ενεργοποιείται και αλλάζει το πεδίο τιμών των μεταβλητών που εμπλέκονται σε αυτόν. Για παράδειγμα:

```
?- X :: 1..5, B #= (X #> 3).
X = X{1..5}
B = B{[0, 1]}
```

αλλά αν η B ενοποιηθεί με την τιμή 1, τότε το πεδίο της X μεταβάλλεται ανάλογα:

```
?- X :: 1..5, B #= (X #> 3), B #= 1.
X = X{[4, 5]}
B = 1
```

Τέλος, ο ακόλουθος reified περιορισμός αφορά τα πεδία των μεταβλητών

```
::(Var,Domain,Bool)
```

Η λειτουργία του είναι η ακόλουθη:

- Αν το τρέχον πεδίο της Var, είναι υποσύνολο του πεδίου Domain, τότε η μεταβλητή Bool γίνεται 1,
- Αν τα δύο πεδία δεν έχουν κοινά στοιχεία τότε η μεταβλητή Bool, γίνεται 0.
- Σε οποιαδήποτε άλλη περίπτωση το πεδίο της B είναι το διάστημα ακεραίων 0..1

Οι ακόλουθες ερωτήσεις δείχνουν την λειτουργία του περιορισμού:

```
?- X #:: [1..10], ::(X, 1..15, B).
X = X{1..10}
B = 1
Yes
?- X #:: [1..10], ::(X, 11..15, B).
X = X{1..10}
B = 0
Yes
?- X #:: [1..10], ::(X, 1..5, B).
X = X{1..10}
B = B{[0, 1]}
There is 1 delayed goal.
```

```
Yes (0.00s cpu)
?- X #:: [1..10], ::(X, 1..5, B), B = 1.
X = X{1..5}
B = 1
Yes (0.00s cpu)
```

12.10 Σημαντικές Παρατηρήσεις

Προσοχή στους περιορισμούς

Στην περίπτωση των ακέραιων περιορισμών, η ECLiPSe θεωρεί ότι κάθε μέρος της αριθμητικής έκφρασης του περιορισμού έχει μια ακέραια τιμή. Έτσι για παράδειγμα αν και σύμφωνα με τα μαθηματικά οι εκφράσεις X/2 + Y/2 = 3 και X + Y = 6 θα έπρεπε να έχουν τις ίδιες λύσεις στο διάστημα 1..10 για τις μεταβλητές X και Y, αν διατυπώσουμε τον πρώτο περιορισμό σαν Prolog ερώτηση, θα έχουμε:

```
?- [X, Y] #:: [1..10], X / 2 + Y / 2 #= 3, labeling([X, Y]).
X = 2
Y = 4
Yes (0.00s cpu, solution 1, maybe more)
X = 4
Y = 2
Yes (0.00s cpu, solution 2)
```

ενώ για τον δεύτερο περιορισμό:

```
?- [X, Y] #:: [1..10], X + Y #= 6, labeling([X, Y]).
X = 1
Y = 5
Yes (0.00s cpu, solution 1, maybe more)
X = 2
Y = 4
Yes (0.00s cpu, solution 2, maybe more)
X = 3
Y = 3
Yes (0.00s cpu, solution 3, maybe more)
X = 4
Y = 2
Yes (0.00s cpu, solution 4, maybe more)
X = 5
Y = 1
Yes (0.00s cpu, solution 5)
```

Για να γίνει κατανοητή η παραπάνω συμπεριφορά, θα πρέπει να λάβουμε υπόψη ότι ο όρος X/2 επιβάλει στη μεταβλητή X να πάρει μόνο άρτιες τιμές, έτσι ώστε το ενδιάμεσο αποτέλεσμα της πρόσθεσης να είναι ακέραιο. Το ίδιο συμβαίνει και για τον όρο Y/2.

Στόχοι σε αναβολή

Σε κάποιες από τις ερωτήσεις, η ECLiPSe Prolog επιστρέφει εκτός από την απάντηση και ένα μήνυμα στο οποίο αναφέρεται ότι υπάρχουν στόχοι σε αναβολή (delayed goals). Για παράδειγμα στην ακόλουθη ερώτηση η ECLiPSe Prolog απαντά:

```
?- [X, Y] #:: [1..10], X #> Y.
X = X{2..10}
Y = Y{1..9}
There is 1 delayed goal.
Yes (0.01s cpu)
```

Όπως αναφέρθηκε, ένας στόχος σε αναβολή σημαίνει ότι υπάρχει ένας περιορισμός ο οποίος είναι ακόμη ενεργός, δηλαδή θα μπορούσε να μειώσει ακόμη περισσότερο τα πεδία. Αυτό σημαίνει ότι η γλώσσα δεν εγγυάται βάσει των περιορισμών ότι μπορεί να υπάρξει λύση, αλλά μόνο το γεγονός ότι αν υπάρχει λύση θα

είναι μέσα στα παραπάνω πεδία. Εξετάζοντας την κλασική περίπτωση τριών μεταβλητών, διαφορετικών μεταξύ τους, με κοινό πεδίο τιμών μεγέθους 3, παρατηρούμε ότι ενώ γνωρίζουμε ότι δεν υπάρχει λύση η Prolog απαντά θετικά:

```
?- [X, Y, Z] #:: [1..2], X #\= Y, Y #\= Z, Z #\= X.
X = X{[1, 2]}
Y = Y{[1, 2]}
Z = Z{[1, 2]}
There are 3 delayed goals.
Yes (0.00s cpu)
```

Η ύπαρξη στόχων σε αναβολή, μπορεί να δείχνει ότι σε κάποιες μεταβλητές δεν έχει γίνει απόδοση τιμής. Έτσι στο παραπάνω πρόβλημα, μπορεί να διαγνωστεί η ασυνέπεια αν προστεθεί ο στόχος labeling/1:

```
?- [X, Y, Z] #:: [1..2], X #\= Y, Y #\= Z, Z #\= X, labeling([X, Y,
Z]).
No (0.00s cpu)
```

Αν εξασφαλιστεί ότι δεν υπάρχουν μεταβλητές χωρίς τιμές, και οι στόχοι σε αναβολή είναι λίγοι, τότε τα μηνύματα μπορούν να αγνοηθούν με σχετική ασφάλεια.

Βιβλιοθήκες και κατηγορήματα

Σε μερικές περιπτώσεις, ένα κατηγόρημα έχει δύο εκδόσεις που ανήκουν στις βιβλιοθήκες ic και ic_global. Για παράδειγμα, το κατηγόρημα maxlist/2, το οποίο παρουσιάστηκε παραπάνω έχει δύο υλοποιήσεις που ανήκουν στις δύο βιβλιοθήκες που αναφέρθηκαν. Αν σε ένα πρόγραμμα φορτωθούν και οι δύο βιβλιοθήκες τότε απαιτείται να διευκρινίσουμε σε ποια από τις δύο εκδόσεις του κατηγορήματος αναφερόμαστε στον κώδικα. Αυτό γίνεται απλά δηλώνοντας την βιβλιοθήκη πριν το κατηγόρημα. Έτσι ο υποστόχος:

```
ic:maxlist(List,Max)
```

αναφέρεται στο κατηγόρημα maxlist/2 της βιβλιοθήκης ic, ενώ ο υποστόχος:

```
ic_global:maxlist(List,Max)
```

στο κατηγόρημα με το ίδιο όνομα της βιβλιοθήκης ic_global.

Αν και ίδιο όνομα κατηγορήματος σημαίνει ότι σημασιολογικά τα δύο είναι όμοια, με την έννοια ότι επιβάλουν τον ίδιο περιορισμό, ο τρόπος με τον οποίο αυτός επιβάλλεται, δηλαδή ο αλγόριθμος ο οποίος αφαιρεί τιμές από τα πεδία είναι διαφορετικός. Για παράδειγμα, ο περιορισμός alldifferent/1, ο οποίος δηλώνει ότι οι μεταβλητές στη λίστα του ορίσματός του πρέπει να πάρουν διαφορετικές τιμές, στις βιβλιοθήκες ic και ic_global υλοποιείται με διαφορετικό τρόπο. Στην πρώτη περίπτωση επιβάλλεται ο περιορισμός #\= στις μεταβλητές:

```
?-[X,Y,Z]#::[1..2], ic:alldifferent([X,Y,Z]).
X = X{[1,2]}
Y = Y{[1,2]}
Z = Z{[1,2]}
There are 3 delayed goals.
Yes (0.00s cpu)
```

ενώ η βιβλιοθήκη ic_global εξετάζει τον περιορισμό σε πιο γενικό επίπεδο:

```
?- [X,Y,Z] #::[1..2], ic_global:alldifferent([X,Y,Z]).
No (0.00s cpu)
```

Το παραπάνω δεν σημαίνει ότι η μια βιβλιοθήκη είναι καλύτερη από την άλλη σε όλες τις περιπτώσεις. Η ic_global βρίσκει την ασυνέπεια συντομότερα, αλλά απαιτεί περισσότερο πολύπλοκους υπολογισμούς (περισσότερο χρόνο στην εκτέλεσή της). Αντίθετα, η έκδοση του κατηγορήματος ic, λόγω των περιορισμών #\= που επιβάλλει απαιτεί την ανάθεση τιμών για να διαγνώσει τη μη ύπαρξη λύσης:

```
?-[X,Y,Z]#::[1..2], ic:alldifferent([X,Y,Z]), labeling([X,Y,Z]).
No (0.00s cpu)
```

Και στις δύο περιπτώσεις η λύση που προκύπτει είναι ορθή.

Βιβλιογραφία

Αν και στα κλασικά βιβλία της Prolog αφιερώνεται συνήθως ένα κεφάλαιο στον λογικό προγραμματισμό με περιορισμούς, τα βιβλία τα οποία είναι αποκλειστικά αφιερωμένα στο πεδίο είναι τα (Apt and Wallace 2006) και (Marriot and Stuckey 1998). Ειδικότερα το πρώτο αφορά στο σύστημα ECLiPSe και παρουσιάζει τις τεχνικές μοντελοποίησης προβλημάτων σε βάθος και με αρκετά μεγάλο πλήθος παραδειγμάτων. Κλασικό σύγγραμμα που αφορά γενικά την περιοχή του προγραμματισμού με περιορισμούς είναι το (Apt 2003). Η πρώτη ερευνητική εργασία η οποία εισήγαγε την έννοια του αλγορίθμου συνέπειας τόξου είναι η (Mackworth 1977) ενώ μετά από αυτήν έχουν ακολουθήσει πλήθος παραλλαγών του αλγορίθμου. Τέλος, οι αλγόριθμοι συνέπειας καθώς και οι γενικές αρχές του πεδίου τους προγραμματισμού με περιορισμούς παρουσιάζονται συνοπτικά στα αντίστοιχα κεφάλαια βιβλίων της Τεχνητής Νοημοσύνης, όπως για παράδειγμα τα (Russel and Norvig 2009) και (Βλαχάβας κα. 2011) και στα ίδια βιβλία υπάρχει και μια περισσότερο αναλυτική περιγραφή του αλγορίθμου επέκτασης και οριοθέτησης, ο οποίος αποτελεί κλασικό αλγόριθμο εύρεσης βέλτιστης λύσης.

Η ECLiPSe Prolog είναι ελεύθερο λογισμικό το οποίο είναι διαθέσιμο από την ιστοσελίδα http://eclipseclp.org/. Στον ιστότοπο, εκτός από τα κλασικά εγχειρίδια της γλώσσας όπου περιγράφονται με λεπτομέρεια τόσο οι βιβλιοθήκες που παρουσιάστηκαν παραπάνω, όσο και το μεγάλο πλήθος των υπολοίπων βιβλιοθηκών του συστήματος, περιέχεται σημαντικό εκπαιδευτικό υλικό (tutorials).

Apt, K. R. (2003). *Principles of Constraint Programming*, 1st Edition Cambridge University Press

Apt, K. R. and Wallace M. (2006). *Constraint Logic Programming using Eclipse*, Cambridge University Press

Mackworth, A.K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8:99-118.

Mariott, K. and Stuckey P. (1998). *Programming with Constraints: an Introduction*, MIT Press

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. 3rd edition. Prentice Hall.

Βλαχάβας, Ι. και Κεφαλάς, Π. και Βασιλειάδης, Ν. και Κόκκορας, Φ. και Σακελλαρίου Η. (2011). *Τεχνητή Νοημοσύνη*. Γ' Έκδοση. ISBN: 978-960-8396-64-7. Εκδόσεις Πανεπιστημίου Μακεδονίας.

Άλυτες Ασκήσεις

12.1 Κλασικά προβλήματα στο CLP είναι τα κρυπταριθμητικά παζλ. Ένα κλασσικό τέτοιο puzzle αφορά την ανάθεση τιμών από 0..9 στα γράμματα που υπάρχουν, ώστε κάθε γράμμα να πάρει μια διαφορετική τιμή και να ικανοποιείται η ακόλουθη σχέση:

```
  D O N A L D
+ G E R A L D
-------------
= R O B E R T
```

Η λύση είναι η ακόλουθη:

```
[D,O,N,A,L,G,E,R,B,T]
[5,2,6,4,8,1,9,7,3,0]
```

Να υλοποιήσετε ένα κατηγόρημα donald/1, που μοντελοποιεί το παραπάνω πρόβλημα και επιστρέφει στην μεταβλητή L, τη λύση στο puzzle. (υποθέτουμε ότι σε κάθε θέση της λίστας αντιστοιχεί ένα γράμμα, όπως φαίνεται παραπάνω). Για παράδειγμα:

```
?- donald(L).
L = [5, 2, 6, 4, 8, 1, 9, 7, 3, 0]
Yes
```

12.2 Σε ένα εστιατόριο οι τιμές των προσφερόμενων πιάτων έχουν διαμορφωθεί ως ακολούθως:

```
item(pizza,12).
```

```
item(burger,14).
item(kingburger,18).
item(platSurprise,15).
```

Ένας πελάτης που επισκέφτηκε το εστιατόριο ζήτησε να μάθει ποια πιάτα μπορεί να παραγγείλει ώστε το συνολικό ποσό που θα πληρώσει να είναι σε κάθε περίπτωση ακριβώς 50 ευρώ. Κάθε πιάτο μπορεί να παραγγελθεί περισσότερες από μια φορές, και ο πελάτης θα ήθελε να μάθει όλους τους δυνατούς συνδυασμούς. Γράψτε ένα Prolog πρόγραμμα το οποίο επιλύει το παραπάνω πρόβλημα. Στο πρόγραμμά σας θα πρέπει να έχετε ένα κατηγόρημα menu(Amount, Order), το οποίο για δοθέν ποσό Amount, επιστρέφει στην λίστα Order τα πιάτα και τον αριθμό από κάθε πιάτο που μπορεί να παραγγελθεί. Το πρόγραμμα θα πρέπει να είναι γενικό με την έννοια ότι θα πρέπει να μπορεί να δουλεύει με οποιαδήποτε λίστα πιάτων, αλλάζοντας φυσικά τα αντίστοιχα γεγονότα της Prolog. Για παράδειγμα:

```
?- menu(50, L).
L = [0 - pizza, 1 - burger, 2 - kingburger, 0 - platSurprise]
L = [3 - pizza, 1 - burger, 0 - kingburger, 0 - platSurprise]
No (0.01s cpu)

?- menu(49, L).
No (0.01s cpu)
```

(**Υπόδειξη:** Μπορείτε πρώτα να υπολογίσετε τις ποσότητες από κάθε πιάτο και μετά να συνδυάσετε την ποσότητα με το όνομα του πιάτου).

12.3 Σε ένα πανεπιστήμιο θα πρέπει να ανατεθούν 10 διπλωματικές εργασίες σε αντίστοιχους 10 φοιτητές. Ο κάθε φοιτητής μπορεί να δηλώσει μέχρι 5 εργασίες με σειρά προτίμησης. Για παράδειγμα ο φοιτητής 1, δήλωσε ότι επιθυμεί να αναλάβει τις εργασίες 4,1,3,5,6, με σειρά προτίμησης αυτή που εμφανίζεται. Οι επιλογές που έκαναν οι φοιτητές φαίνονται παρακάτω:

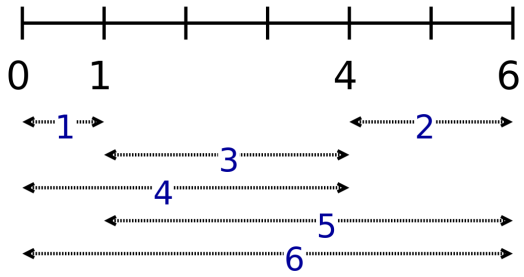
```
student(alex,[4,1,3,5,6]).
student(nick,[6,3,5,2,4]).
student(jack,[8,4,5,7,10]).
student(helen,[3,7,8,9,1]).
student(maria,[7,1,5,6,4]).
student(evita,[8,4,7,9,5]).
student(jacky,[5,6,7,4,10]).
student(peter,[2,6,10,8,3]).
student(john,[1,3,10,9,6]).
student(mary,[1,6,7,9,10]).
```

Να αναπτύξετε ένα Prolog πρόγραμμα (με top-level κατηγόρημα το solve(S, Cost)) το οποίο κάνει την καλύτερη ανάθεση διπλωματικών εργασιών στους φοιτητές δεδομένων των επιλογών τους. Όλοι οι φοιτητές είναι ίσοι, και οι επιλογές που γίνονται θα πρέπει να δίνουν (όσο είναι δυνατό) την εργασία που προτιμά ο φοιτητής περισσότερο. Παράδειγμα εκτέλεσης είναι το ακόλουθο:

```
?- solve(Students, C).
Students = [(alex, 4), (nick, 6), (jack, 8), (helen, 3), (maria, 7),
(evita, 9), (jacky, 5), (peter, 2), (john, 10), (mary, 1)]
C = 15
Yes (0.01s cpu)
```

12.4 Μια σειρά από κεραίες πρέπει να τοποθετηθούν στη σειρά έτσι ώστε η απόσταση μεταξύ δύο κεραιών να είναι διαφορετική από την απόσταση μεταξύ δύο οποιονδήποτε άλλων κεραιών. Αν θεωρήσουμε ότι η πρώτη κεραία είναι στη θέση 0, τότε ποια είναι η ελάχιστη απόσταση που μπορεί να τοποθετηθεί η τελευταία

κεραία? Για παράδειγμα, το Σχήμα 12.8 δείχνει την τοποθέτηση 4 κεραιών σύμφωνα με τον παραπάνω κανόνα:



Σχήμα 12.8: *Τοποθέτηση 4 κεραιών της άσκησης 12.4.*

Να υλοποιήσετε ένα πρόγραμμα το οποίο βρίσκει τις θέσεις των κεραιών για διαφορετικούς αριθμούς κεραιών και επιστρέφει την ελάχιστη απόσταση. Για παράδειγμα:

```
?- antennas(4, Max, L).
Max = 6
L = [0, 1, 4, 6]
Yes

?- antennas(5, Max, L).
Max = 11
L = [0, 1, 4, 9, 11]
Yes

?- antennas(7, Max, L).
Max = 25
L = [0, 1, 4, 10, 18, 23, 25]
```

12.5 Σε ένα πλοίο μεταφοράς ιστιοπλοϊκών πλοιαρίων πρέπει να φορτωθεί ένα σύνολο από πλοιάρια με τα ακόλουθα χαρακτηριστικά:

- πλοιάριο Α, πλάτος 150cm και μήκος 6m,
- πλοιάριο Β, πλάτος 180cm και μήκος 8m,
- πλοιάριο Γ, πλάτος 230cm και μήκος 11m,
- πλοιάριο Δ, πλάτος 130cm και μήκος 5m,
- πλοιάριο Ε, πλάτος 210cm και μήκος 9m,

Αν το πλάτος του χώρου φόρτωσης είναι 400cm, ζητούμενα είναι (a) το ελάχιστο μήκος που θα καταλαμβάνουν τα παραπάνω πλοιάρια δεδομένου ότι μπορούν να τοποθετηθούν χωρίς κενά μεταξύ τους και (b) η τοποθέτησή τους. Για την τελευταία απαιτούνται τα μέτρα απόστασης τοποθέτησης της πλώρης του κάθε πλοιαρίου από την αρχή (σημείο 0) του χώρου φόρτωσης. Να γράψετε ένα κατηγόρημα ship_loading/2 σε CLP το οποίο να επιστρέφει την απόσταση τοποθέτησης του κάθε πλοιαρίου από την αρχή του χώρου φόρτωσης και το ελάχιστο μήκος που καταλαμβάνουν όλα τα πλοιάρια. Για παράδειγμα:

```
?- ship_loading(Placement, L).
Placement = [0, 11, 0, 6, 11]
L = 20
```

12.6 Σε μια εταιρεία, πρέπει να μεταφερθούν κάποια από τα 10 κιβώτια διαφορετικού βάρους ιδίου όγκου, από δύο φορτηγά. Το φορτηγό Α μπορεί να μεταφέρει 3 κιβώτια με μέγιστο βάρος 1200kg, ενώ το δεύτερο 4 κιβώτια με μέγιστο βάρος 1350kg. Τα βάρη των κιβωτίων δίνονται από γεγονότα της μορφής:

```
box(1,140).
```

```
box(2,200).
...
box(9,600).
box(10,650).
```

Να αναπτύξετε ένα CLP πρόγραμμα το θα έχει ένα κατηγόρημα load_trucks/4 το οποίο να βρίσκει ποια κιβώτια θα πρέπει να μεταφέρει κάθε φορτηγό και το συνολικό βάρος των κιβωτίων, για να εκμεταλλευόμαστε όσο το δυνατό περισσότερο το ωφέλιμο φορτίο τους. Για παράδειγμα, στην ακόλουθη εκτέλεση, TA είναι τα κιβώτια που θα μεταφέρει το φορτηγό Α, και LA το βάρος τους, ενώ TB και LB τα αντίστοιχα μεγέθη για το φορτηγό Β:

```
?- load_trucks(TA,LA,TB,LB).
TA = [2, 4, 6]
LA = 1200
TB = [3, 5, 8, 10]
LB = 1345
Yes
```

12.7 Σε μια πανεπιστημιακό τμήμα , πρέπει να γίνουν οι ακόλουθες εξετάσεις :

- εξέταση μαθήματος Α, διάρκειας 150' και φοιτητές 120,
- εξέταση μαθήματος Β, διάρκειας 180' και φοιτητές 200,
- εξέταση μαθήματος Γ, διάρκειας 230' και φοιτητές 190,
- εξέταση μαθήματος Δ, διάρκειας 130' και φοιτητές 250,
- εξέταση μαθήματος Ε, διάρκειας 210' και φοιτητές 70,

Υποθέστε ότι

- υπάρχει ένας μεγάλος χώρος εξετάσεων χωρητικότητας 400 ατόμων και ότι
- δύο ή περισσότερα μαθήματα μπορούν να εξεταστούν ταυτόχρονα σε αυτό τον χώρο,
- οι χρόνοι έναρξης και τέλους κάθε εξέτασης είναι ανεξάρτητοι για κάθε μάθημα, ακόμη και αν γίνονται στην ίδια αίθουσα.
- Το μάθημα Α δεν μπορεί να συμπίπτει με το μάθημα Γ.

Ζητούμενα είναι (α) ο ελάχιστος χρόνος που θα πρέπει να χρησιμοποιηθεί ο χώρος εξετάσεων για να ολοκληρωθούν όλες οι εξετάσεις και (β) η χρονική στιγμή που θα πρέπει να ξεκινήσει η κάθε εξέταση (από την χρονική στιγμή 0, όπου και ξεκινούν όλες οι εξετάσεις. Να γράψετε ένα κατηγόρημα exams/2 σε CLP το οποίο να επιστρέφει τον χρόνο έναρξης της κάθε εξέτασης και το συνολικό χρόνο που θα χρησιμοποιηθεί η αίθουσα. Για παράδειγμα:

```
?- exams(L, Duration).
L = [0, 0, 150, 380, 180]
Duration = 510
```

12.8 Σε ένα πανεπιστήμιο θα πρέπει να γίνουν εξετάσεις σε ένα σύνολο από αίθουσες διαφορετικής χωρητικότητας. Η χωρητικότητα των αιθουσών και οι ανάγκες σε θέσεις (φοιτητές) του κάθε μαθήματος δίνονται με την μορφή γεγονότων:

Πληροφορία για τις αίθουσες

```
room(1,20).
room(2,40).
room(3,50).
   ...
room(10,60).
```

Πληροφορία για τις εξετάσεις

```
class(exam1,10).
class(exam2,60).
class(exam3,70).
   ...
class(exam10,30).
```

Αν (α) όλες οι εξετάσεις λαμβάνουν χώρα την ίδια στιγμή, (β) κάθε εξέταση θα πρέπει να γίνει σε διαφορετική αίθουσα και (γ) προφανώς κάθε εξέταση πρέπει να γίνει σε μια αίθουσα με ικανοποιητική χωρητικότητα ώστε να χωρέσουν οι φοιτητές του μαθήματος, να αναπτύξετε ένα πρόγραμμα που να βρίσκει σε ποια αίθουσα θα γίνει η εξέταση του κάθε μαθήματος. Για παράδειγμα (το cl(exam1,1) σημαίνει ότι η εξέταση του μαθήματος exam1, θα γίνει στην αίθουσα 1):

```
?- exam_schedule(List).
List = [cl(exam1, 1), cl(exam2, 10), cl(exam3, 8), cl(exam4, 3),
cl(exam5, 2), cl(exam6, 5), cl(exam7, 4), cl(exam8, 7), cl(exam9, 9),
cl(exam10, 6)]
```

12.9 Ένα μουσείο μπορεί να δεχθεί το πολύ 100 επισκέπτες σε κάθε χρονική στιγμή, ανεξάρτητα από τον αριθμό των ομάδων (group) στις οποίες θα είναι οργανωμένοι. Κάθε ομάδα θέλει να επισκεφτεί το μουσείο για διαφορετικό χρονικό διάστημα (διαφορετική διάρκεια επίσκεψης). Έστω ότι υπάρχουν:

- ομάδα A, με 60 επισκέπτες, που θα επισκεφτεί το μουσείο για 2 ώρες
- ομάδα B, με 30, επισκέπτες, που θα επισκεφτεί το μουσείο για 1 ώρες
- ομάδα C, με 50, επισκέπτες, που θα επισκεφτεί το μουσείο για 2 ώρες
- ομάδα D, με 40, επισκέπτες, που θα επισκεφτεί το μουσείο για 5 ώρες

Αν όλες οι επισκέψεις μπορούν να ξεκινήσουν από τη χρονική στιγμή 0 και μετά (συμπεριλαμβανομένης της χρονικής στιγμής 0), να γράψετε ένα κατηγόρημα museum/1 σε CLP το οποίο να επιστρέφει τους χρόνους έναρξης κάθε επίσκεψης των ομάδων A, B ,C, D σε λίστα, ελαχιστοποιώντας το συνολικό χρόνο που απαιτείται για την ολοκλήρωση όλων των επισκέψεων. Για παράδειγμα:

```
?- museum(S).
S = [0, 2, 3, 0]
```

ΚΕΦΑΛΑΙΟ 13: Επεξεργασία Γλώσσας και Γραμματικές Οριστικών Προτάσεων

Λέξεις Κλειδιά:
Φυσική Γλώσσα, Τεχνητές Γλώσσες, Γραμματικές, Ιεραρχία Chomsky, Γραμματικές Οριστικών Προτάσεων (DCG), Μεταγλωττιστές, Ορισμός και χρήση Τελεστών

Περίληψη
Η επεξεργασία φυσικής γλώσσας αποτέλεσε μια από τις πρώτες εφαρμογές του λογικού προγραμματισμού. Το κεφάλαιο παρουσιάζει το πως η Prolog, υποστηρίζει την ανάπτυξη εφαρμογών της κατηγορίας, μέσω γραμματικών οριστικών προτάσεων, και παραθέτει ένα παράδειγμα ανάπτυξης ενός απλού μεταγλωττιστή στην Prolog. Το κεφάλαιο επίσης παρουσιάζει τους τελεστές στην Prolog, και τον τρόπο τον ορισμού νέων τελεστών.

Μαθησιακοί Στόχοι

Με την ολοκλήρωση της θεωρίας και την επίλυση των ασκήσεων αυτού του κεφαλαίου, ο αναγνώστης θα είναι ικανός να:

- γνωρίζει τις βασικές αρχές γλωσσών και γραμματικών,
- μπορεί να υλοποιήσει μία απλή γραμματική οριστικών προτάσεων,
- κατανοήσει τις βασικές τεχνικές για την απόδοση σημασιολογίας στη γλώσσα,
- μπορεί να σχεδιάσει και να υλοποιήσει ένα απλό μεταγλωττιστή για μία τεχνητή γλώσσα.

Παράδειγμα κίνητρο

Η διεπαφή ανθρώπου-μηχανής αποκτά όλο και περισσότερα ανθρώπινα χαρακτηριστικά. Για παράδειγμα το ποντίκι αρχίζει να δίνει τη θέση του στην αφή, η πληκτρολόγηση δίνει τη θέση της στη αναγνώριση κειμένου από προφορικό λόγο (φωνητική πληκτρολόγηση) κλπ. Η τεχνολογία μας επιτρέπει να σχεδιάζουμε και να υλοποιούμε εφαρμογές Τεχνητής Νοημοσύνης όπως:

- η αυτόματη μετάφραση/διερμηνεία,
- η φωνητική προσπέλαση μηχανών αναζήτησης και βάσεων δεδομένων,
- η αυτόματη περίληψη και κατηγοριοποίηση κειμένων,
- η διεπαφή και διάλογος σε φυσική γλώσσα μεταξύ χρήστη και μηχανής,
- η κατανόηση περιεχομένου ιστοσελίδων και υπηρεσιών στον σημασιολογικό παγκόσμιο ιστό.

Βασική προϋπόθεση για την επιτυχή υλοποίηση όλων αυτών είναι η επεξεργασία και κατανόηση της γλώσσας. Η Prolog μας δίνει τη δυνατότητα να υλοποιήσουμε σχετικά εύκολα προγράμματα που αναλύουν συντακτικά τις προτάσεις μιας γλώσσας και επιτρέπει τη εύκολη διασύνδεση τους με τη σημασιολογία αυτών των προτάσεων. Για παράδειγμα, ένα από τα σύγχρονα επιτεύγματα της Τεχνητής Νοημοσύνης είναι ο υπολογιστής της IBM με το όνομα Watson ο οποίος χρησιμοποιεί Γραμματικές Οριστικών Προτάσεων σε Prolog για να κατανοήσει την ερώτηση που εκφωνεί ο παρουσιαστής του παιχνιδιού Jeopardy.

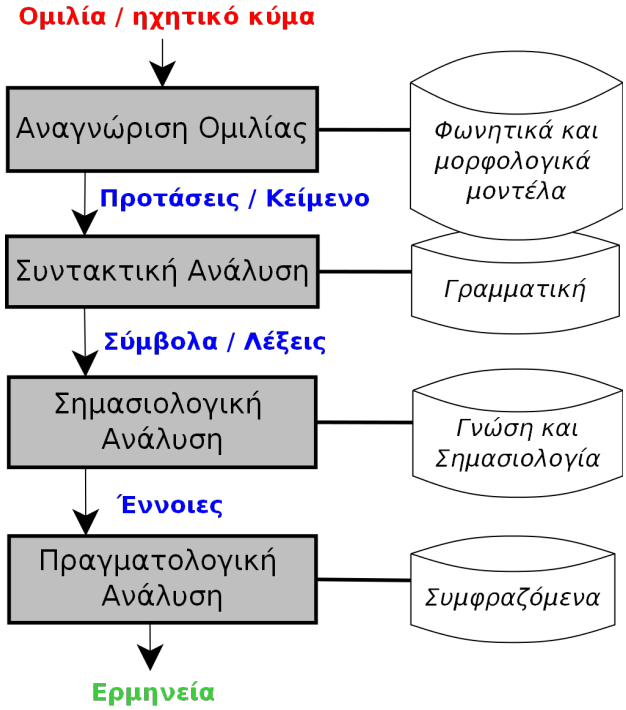
Μονοπάτι Μάθησης

Οι ενότητες αυτού του κεφαλαίου είναι πέντε. Οι βασικές αρχές των DCG σε Prolog δίνονται στην 13.2 και θεωρούμε ότι ο αναγνώστης θα διαλέξει μία από τις δύο βασικές εφαρμογές είτε για φυσική (13.3) ή για τεχνητή γλώσσα (13.4). Η εισαγωγική ενότητα απευθύνεται σε όσους θέλουν να γνωρίσουν πολύ περιληπτικά

τη θεωρία γλωσσών. Η τελευταία ενότητα αφορά στον ορισμό τελεστών για την καλύτερη αναγνωσιμότητα λογικών προγραμμάτων ώστε να έχουν στοιχεία από τη φυσική γλώσσα.

13.1 Επεξεργασία και Κατανόηση της Φυσικής Γλώσσας

Η επεξεργασία και κατανόηση της φυσικής προφορικής γλώσσας είναι μια πολύπλοκη διαδικασία η οποία αρχίζει με την εξεργασία ενός ηχητικού κύματος και έχει ως αποτέλεσμα την ερμηνεία του (Σχήμα 13.1). Τα ηχητικά σήματα μετατρέπονται σε φθόγγους και στη συνέχεια από αυτούς παράγονται λέξεις και τελικά προτάσεις κειμένου. Ακολουθεί η συντακτική ανάλυση η οποία ξεχωρίζει τις λέξεις (τερματικά σύμβολα) των προτάσεων προτάσεων και τις επεξεργάζεται βάσει της γραμματικής της γλώσσας. Παράγεται το δένδρο σύνταξης και γίνεται μετατροπή των προτάσεων σε άλλες εσωτερικές δομές αναπαράστασης χρησιμοποιώντας τη νοηματική σημασία των λέξεων μέσω της σημασιολογικής ανάλυσης. Τέλος επιχειρείται η πραγματολογική ανάλυση, δηλαδή η ένταξη της πρότασης μέσα στο γενικότερο νοηματικό πλαίσιο των συμφραζομένων, λαμβάνοντας υπόψη τις συνθήκες μέσα στις οποίες αυτή ειπώθηκε. Στο γραπτό λόγο που αφορά είτε στη φυσική ή σε τεχνητές γλώσσες (πχ γλώσσες αναπαράστασης ή προγραμματισμού κλπ), το στάδιο επεξεργασίας ηχητικού κύματος δεν υπάρχει. Αντίστοιχη διαδικασία προς την αντίθετη φορά γίνεται για την παραγωγή φυσικής γλώσσας. Η λεπτομερής περιγραφή της διαδικασίας δεν αποτελεί αντικείμενο αυτού του συγγράμματος. Ο ενδιαφερόμενος μπορεί να ανατρέξει σε άλλα συγγράμματα Τεχνητής Νοημοσύνης ή εν γένει Υπολογιστικής Γλωσσολογίας. Στο κεφάλαιο αυτό, θα ασχοληθούμε μόνο με το κομμάτι που αφορά στον ορισμό γλωσσών, είτε φυσικής ή τεχνητών, και στη συντακτική ανάλυση μέσω Γραμματικών Οριστικών Προτάσεων (DCG) της Prolog. Αναφορά θα γίνει και στη σημασιολογική ανάλυση φυσικής γλώσσας τόσο όσο να αναδείξει τη χρησιμότητα των DCG της Prolog. Τέλος, δείχνουμε τη δυνατότητα ανάπτυξης μεταγλωττιστών για τεχνητές γλώσσες μέσω DCG.



Σχήμα 13.1: *Τα στάδια επεξεργασίας και κατανόησης της φυσικής γλώσσας*

13.2 Τύποι Γλωσσών και Επεξεργασία Φυσικής Γλώσσας

Ορισμός της Γλώσσας και της Γραμματικής

Για την μελέτη των γλωσσών είναι απαραίτητο να ορίσουμε με αυστηρό τρόπο (μαθηματικά) τι αποτελεί γλώσσα και πως μπορεί να παραχθεί. Έτσι στα ακόλουθα παρουσιάζονται οι έννοιες της τυπικής γλώσσας (formal language) και των παραγωγικών τους μοντέλων, δηλαδή των τυπικών γραμματικών.

Μια **Γλώσσα (Language)** ορίζεται ως το σύνολο των συμβολοσειρών (φράσεων) που παράγεται από μία πεπερασμένη αλφάβητο συμβόλων. Τα σύμβολα τα οποία απαρτίζουν τις συμβολοσειρές ονομάζονται τερματικά σύμβολα. Μια **Γραμματική (Grammar)** ορίζεται ως ο μηχανισμός που μπορεί να παράγει όλες τις συμβολοσειρές (φράσεις) μιας γλώσσας. Μια **Τυπική Γραμματική (Formal Grammar)** είναι η τετράδα $G = (N, \Sigma, P, S)$ όπου:

- N είναι ένα πεπερασμένο σύνολο από **μη-τερματικά σύμβολα (non-terminals)**
- Σ είναι ένα πεπερασμένο σύνολο από **τερματικά σύμβολα (terminals)** το οποίο είναι διαφορετικό από το N
- P είναι ένα πεπερασμένο σύνολο από **κανόνες παραγωγής (production rules)** της μορφής $\alpha \rightarrow \beta$ όπου το $\alpha \in (N \cup \Sigma)^* \, N \, (N \cup \Sigma)^*$ και το $\beta \in (N \cup \Sigma)^*$, είναι δηλαδή της μορφής $(N \cup \Sigma)^* \, N \, (N \cup \Sigma)^* \rightarrow (N \cup \Sigma)^*$.
- $\Sigma \in N$ είναι το **αρχικό σύμβολο (initial symbol)**

Το σύμβολο → διαβάζεται και ερμηνεύεται ως "αποτελείται από" ή "παράγει". Το σύμβολο * (ονομάζεται και κλείσιμο Kleene) υποδηλώνει ότι υπάρχουν μηδέν ή περισσότερες επαναλήψεις από το σύμβολο αυτό. Για παράδειγμα, ο κανόνας $\alpha \rightarrow \beta \, \gamma^*$ διαβάζεται ότι "η έκφραση α αποτελείται από ένα β και μηδέν ή περισσότερες επαναλήψεις του γ" ή εναλλακτικά ότι "η έκφραση α παράγει συμβολοσειρές που ξεκινούν με το β και αποτελούνται από μηδέν ή περισσότερα γ."

Η διαδικασία παραγωγής μιας συμβολοσειράς η οποία ανήκει στη γλώσσα, ξεκινά με το αρχικό σύμβολο, και με την επαναληπτική εφαρμογή των κανόνων παραγωγής, παράγει στο τέλος συμβολοσειρές οι οποίες περιέχουν μόνο τερματικά σύμβολα. Αυτές οι συμβολοσειρές ανήκουν στη γλώσσα σύμφωνα με τη γραμματική. Το συνηθισμένο πρόβλημα το οποίο πρέπει να απαντηθεί για μια γλώσσα είναι αν μια συμβολοσειρά ανήκει σε αυτή και η απάντηση σε αυτό το ερώτημα δίνεται από ένα αυτόματο (automaton), όπως για παράδειγμα μια μηχανή Turing ή ένα αυτόματο πεπερασμένων καταστάσεων.

Ιεραρχία Chomsky

Οι Noam Chomsky και Marcel Schützenberger όρισαν τέσσερις τύπους γλωσσών ανάλογα με τις δυνατότητες που έχει η κάθε μία να περιγράψει σύνθετες ή απλούστερες φράσεις μέσω της γραμματικής τους. Η ιεραρχία αυτή έγινε γνωστή ως ιεραρχία Chomsky (Πίνακας 13.1).

Πίνακας 13.1: *Η Ιεραρχία Γλωσσών κατά Chomsky*

Τύπος Γλώσσας	Ονομασία Γλώσσας	Μηχανή που αναγνωρίζει τη Γλώσσα αυτή
Τύπος-0	Αναδρομικά Απαριθμήσιμες Γλώσσες (Recursively Enumerable)	Μηχανή Turing
Τύπος-1	Με συμφραζόμενα (Context Sensitive)	Μηχανή Turing με περιορισμένο μήκος ταινίας
Τύπος-2	Χωρίς συμφραζόμενα (Context Free)	Αυτόματα Στοίβας

Τύπος-3	Κανονικές (Regular)	Αυτόματα Πεπερασμένων Καταστάσεων

Οι **γλώσσες Τύπου-0** ορίζονται από γραμματικές που αναγνωρίζονται από μία **Μηχανή Turing**. Οι γλώσσες αυτές ονομάζονται **Αναδρομικά Απαριθμήσιμες Γλώσσες (Recursively Enumerable ή Unrestricted)**. Οι γλώσσες αυτές μπορούν να παράγουν τα πάντα. Οι κανόνες είναι της μορφής:

- $\alpha \rightarrow \beta$

όπου:

- α και β είναι τυχαίες συμβολοσειρές ενός λεξιλογίου, δηλαδή $\alpha,\beta \in (N \cup \Sigma)^*$ και
- $\alpha \neq \varepsilon$

Οι **γλώσσες Τύπου-1** ορίζονται από γραμματικές που αναγνωρίζονται από μία **μη-ντετερμινιστική μηχανή Turing με πεπερασμένο μήκος ταινίας (Linear-bounded Automata)**. Οι γλώσσες αυτές μπορούν να χειριστούν διαφορετικά συμφραζόμενα επειδή γνωρίζουν τι ειπώθηκε ήδη και δεν έχουν προκαθορισμένο τι θα ειπωθεί μετά **(Context Sensitive)**. Οι κανόνες είναι της μορφής:

- $\alpha A \beta \rightarrow \alpha B \beta$, *ή*
- $S \rightarrow \varepsilon$

όπου:

- $\alpha,\beta,B \in (N \cup \Sigma)^*$
- $A, S \in N$, με S το αρχικό σύμβολο της γραμματικής.
- $B \neq \varepsilon$, δηλαδή πρέπει να περιέχει τουλάχιστον ένα στοιχείο (τερματικό ή μη τερματικό).

Οι **γλώσσες Τύπου-2** ορίζονται από γραμματικές που αναγνωρίζονται από **Αυτόματα Στοίβας (Push-down Automata)**. Οι γλώσσες αυτές μπορούν να χειριστούν διαφορετικά συμφραζόμενα αλλά μόνο τα πιο πρόσφατα επειδή γνωρίζουν τι ειπώθηκε μόλις πριν και δεν έχουν όλην την εικόνα του τι ειπώθηκε γενικά **(Context Free**). Η φυσική γλώσσα έχει τη δομή μιας γλώσσας Τύπου-2. Οι κανόνες είναι της μορφής:

- $A \rightarrow \alpha$

όπου:

- $A \in N$ και
- $\alpha \in (N \cup \Sigma)^*$

Οι **γλώσσες Τύπου-3** ορίζονται από γραμματικές που αναγνωρίζονται από **Αυτόματα Πεπερασμένων Καταστάσεων (Finite State Automata)**. Οι γλώσσες αυτές ονομάζονται **Κανονικές (Regular)** και δεν έχουν γνώση για το τι έχει ήδη ειπωθεί. Είναι οι πιο απλές γλώσσες και οι κανόνες τους είναι της μορφής:

- $A \rightarrow \varepsilon$
- $A \rightarrow \alpha$
- $A \rightarrow \alpha B$

όπου:

- $A, B \in N$
- $\alpha \in \Sigma$

Παράδειγμα: Μία απλή γραμματική φυσικής γλώσσας

Τα Ελληνικά, όπως και κάθε φυσική γλώσσα, έχουν πολύπλοκους γραμματικούς κανόνες που διέπουν τη δημιουργία φράσεων. Εδώ παραθέτουμε μια απλή γραμματική των Ελληνικών για συγκεκριμένες φράσεις

που περιέχουν μία φράση ουσιαστικού την οποία ακολουθεί μία φράση ρήματος. Ως τυπική γραμματική λοιπόν, η απλή αυτή γλώσσα ορίζεται με την τετράδα (N, Σ, P, S) όπου:

- το σύνολο από μη-τερματικά σύμβολα N είναι

  N= *{φράση, φράση ρήματος, φράση ουσιαστικού, ουσιαστικό, ρήμα, άρθρο}*

- το σύνολο από τερματικά σύμβολα Σ είναι

  Σ = *{γάτα, σκύλος, άνθρωπος, μαϊμού, μπανάνα, κυνηγάει, τρώει,, ο, του, τον, η, της, την, το}*

- το αρχικό σύμβολο S είναι

  S = *φράση*

- το σύνολο από τους κανόνες παραγωγής P είναι

  P = *{*

  φράση→ φράση ουσιαστικού φράση ρήματος ,

  φράση ουσιαστικού → άρθρο ουσιαστικό,

  φράση ρήματος → ρήμα φράση ουσιαστικού,

  ουσιαστικό →γάτα,

  ουσιαστικό →σκύλος,

  ουσιαστικό →άνθρωπος,

  ουσιαστικό →μαϊμού,

  ουσιαστικό →μπανάνα

  άρθρο→ ο,

  άρθρο→ του,

  άρθρο→ τον,

  άρθρο→ η,

  άρθρο→ της,

  άρθρο→ την ,

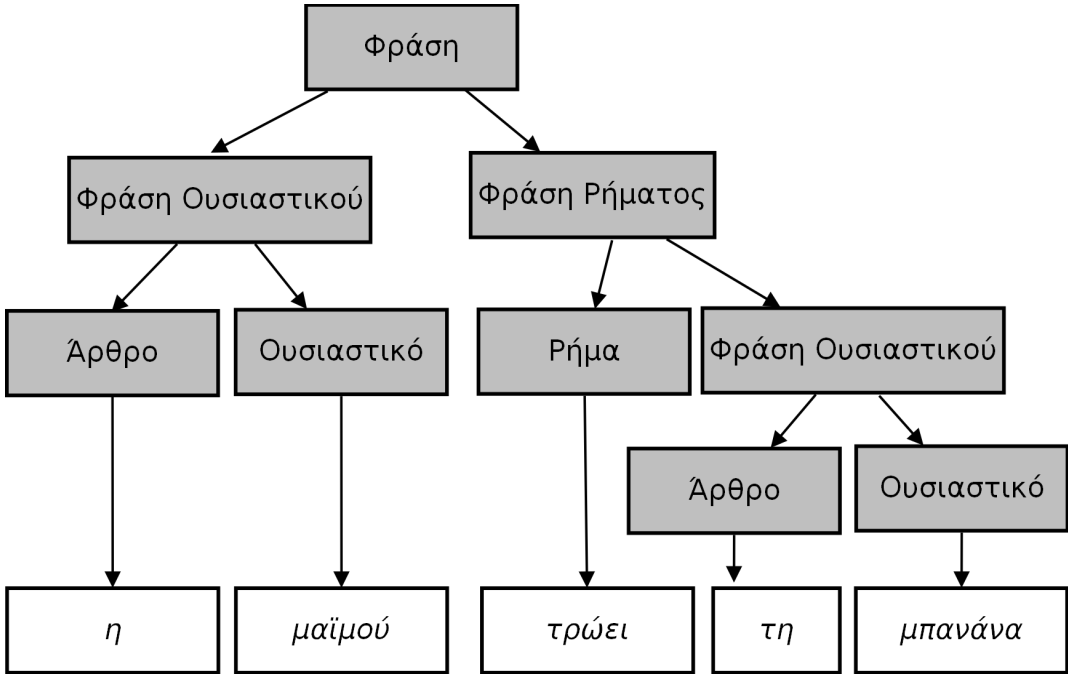
  άρθρο→ το ,

  ρήμα→ κυνηγάει,

  ρήμα→ τρώει

  }

Η παραπάνω γραμματική μπορεί να παράγει και να αναγνωρίσει φράσεις της μορφής "*η μαϊμού τρώει τη μπανάνα*", "*ο σκύλος κυνηγάει την γάτα*" κλπ, οι οποίες έχουν ακριβώς τη δομή των φράσεων που ορίζουν οι κανόνες της. Αυτό φαίνεται καθαρά από μία απλή γραμματική ανάλυση η οποία παράγει το λεγόμενο **Δένδρο Σύνταξης (Syntax Tree)** μιας συγκεκριμένης φράσης. Ένα παράδειγμα δένδρου σύνταξης απεικονίζεται στο Σχήμα 13.2.



Σχήμα 13.2: *Δένδρο Σύνταξης μίας φράσης της απλής γραμματικής*

Στη γραμματική αυτή δεν εξετάζουμε προς στιγμή τη συμφωνία αριθμού, πτώσης ουσιαστικού, άρθρου, κλίση ρήματος κλπ. Έτσι, για παράδειγμα, η φράση "*την άνθρωπος κυνηγάει ο μπανάνα"* είναι καθ' όλα αποδεκτή από τη γραμματική. Επίσης, είναι φανερό ότι η γραμματική δεν αντιμετωπίζει τη σημασία των φράσεων καθώς δεν εμπεριέχει σημασιολογία. Έτσι, για παράδειγμα, η φράση "*η μπανάνα τρώει τη γάτα"* είναι επίσης καθ' όλα αποδεκτή από τη γραμματική.

13.3 Γραμματικές Οριστικών Προτάσεων

Ορισμός των Γραμματικών Οριστικών Προτάσεων (DCG) στην Prolog

Η Prolog έχει έναν ειδικό τρόπο (σύνταξη) για να μπορεί κανείς να ορίσει γραμματικές χωρίς συμφραζόμενα (Context Free Grammars). Οι γραμματικές αυτές στην Prolog ονομάζονται **Γραμματικές Οριστικών Προτάσεων (Definite Clause Grammars)** ή απλά **DCGs**. Στο ειδικό αυτό συντακτικό:

- τα μη-τερματικά σύμβολα είναι κατηγορήματα που εμφανίζονται στο αριστερό μέρος ενός κανόνα παραγωγής και μπορούν να χρησιμοποιηθούν στο δεξιό μέρος οποιουδήποτε κανόνα,
- τα τερματικά σύμβολα είναι άτομα που εμφανίζονται μέσα σε τετράγωνες αγκύλες "[ ]" ,
- οι κανόνες παραγωγής χρησιμοποιούν τον ειδικό ενσωματωμένο τελεστή "**-->**",
- δεν υπάρχει αρχικό σύμβολο (στην πράξη εκφράζεται με την ερώτηση της Prolog όπως θα δούμε παρακάτω).

Για παράδειγμα, ο κανόνας που δηλώνει ένα μη τερματικό σύμβολο,

noun → man

αναγράφεται σε DCG ως:

```
noun -->[man].
```

και ο κανόνας παραγωγής που δηλώνει ένα συντακτικό κανόνα:

sentence → noun_phrase verb_phrase (φράση→ φράση ουσιαστικού φράση ρήματος)

αναγράφεται σε DCG ως:

```
sentence --> noun_phrase, verb_phrase.
```

Παράδειγμα: Μία απλή γραμματική φυσικής γλώσσας σαν DGC

Το σύνολο της απλής γραμματικής που ορίστηκε στην προηγούμενη ενότητα αναγράφεται σε μορφή DCG ως:

```
sentence --> noun_phrase, verb_phrase.
noun_phrase-->article,noun.
verb_phrase-->verb,noun_phrase.
noun-->[cat].
noun-->[dog].
noun-->[man].
noun-->[monkey].
noun-->[banana].
article-->[a].
article-->[the].
verb-->[chases].
verb-->[eats].
```

Ο τρόπος αυτός είναι πολύ εκφραστικός και εύκολος στην συγγραφή γραμματικών κανόνων. Στην πραγματικότητα όμως η Prolog έχει τη δική της αναπαράσταση. Για παράδειγμα, η φράση DCG:

```
sentence --> noun_phrase, verb_phrase.
```

αναπαριστάται εσωτερικά ως φράση Prolog ως εξής:

```
sentence(ListOfTerminals, Result) :-
  noun_phrase(ListOfTerminals, TempList),
  verb_phrase(TempList, Result).
```

Το κατηγόρημα sentence/2 έχει δύο ορίσματα: το πρώτο δηλώνει τη λίστα των τερματικών συμβόλων που αποτελούν μία πρόταση και το δεύτερο η λίστα των τερματικών συμβόλων που απομένει όταν όλα αυτά τα σύμβολα διαβαστούν. Αυτή είναι η κλασσική προσέγγιση του ορισμού ενός κατηγορήματος με **λίστες διαφορών (difference lists)**. Οι λίστες διαφορών είναι μια προηγμένη τεχνική στην οποία μία λίστα εμφανίζεται σαν ζευγάρι ([L|X],X) ή [L|X]-X έτσι ώστε κατηγορήματα να μπορούν με μία ενοποίηση (και όχι με πολλαπλές αναδρομικές κλήσεις) να διαχειριστούν μία λίστα, με αποτέλεσμα τη μείωση της υπολογιστικής πολυπλοκότητας. Συνεπώς, τα τερματικά σύμβολα αναπαριστώνται εσωτερικά ως λίστα διαφορών. Για παράδειγμα, η φράση DCG:

```
noun-->[cat].
```

αναπαριστάται εσωτερικά ως φράση Prolog ως εξής:

```
noun([cat|X],X).
```

που σημαίνει μετά τη συμμετοχή του cat στη λίστα των τερματικών συμβόλων απομένει η λίστα X.

Από την παραπάνω εσωτερική αναπαράσταση μίας DCG προκύπτει ότι η ερώτηση για το αν μία πρόταση, για παράδειγμα “the monkey eats a banana”, ανήκει στη γραμματική που ορίστηκε είναι:

```
?- sentence([the,monkey,eats,a,banana],[]).
```

με αποτέλεσμα επιτυχία. Αντίθετα η ερώτηση για την πρόταση:

```
?- sentence([a,dog,eats,chases,the,cat],[]).
```

αποτυγχάνει γιατί περιέχει ένα παραπάνω ρήμα, ενώ η ερώτηση:

```
?- sentence([the,woman,chases,a,man],[]).
```

αποτυγχάνει γιατί το τερματικό σύμβολο woman δεν ανήκει στα τερματικά σύμβολα της εν λόγω γραμματικής. Ενδιαφέρον έχει η ερώτηση:

```
?- sentence(S,[]).
```

η οποία παράγει όλες τις προτάσεις που είναι αποδεκτές από τη γραμματική, καθώς και η ερώτηση της μορφής:

```
?- sentence([the,Who,chases,a,Whom],[]).
```

η οποία επιστρέφει τα τερματικά σύμβολα με τα οποία η πρόταση μπορεί να γίνει αποδεκτή.

Εδώ μπορείτε να δείτε ένα σύντομο βίντεο με τη υλοποίηση και δοκιμή του προγράμματος.

Επιπλέον ορίσματα στις φράσεις μίας DCG: Παράδειγμα με Δένδρο Σύνταξης

Οι φράσεις μίας γραμματικής DCG μπορούν και αυτές να πάρουν επιπλέον ορίσματα. Αυτό είναι στην πραγματικότητα που κάνει τις DCGs ικανές να εκφράσουν μια επέκταση των γραμματικών χωρίς συμφραζόμενα, τις **κατηγορικές γραμματικές (attributed grammars)** ή γραμματικές ιδιοτήτων. Συνοπτικά, οι κατηγορικές γραμματικές επιτρέπουν την προσκόλληση ιδιοτήτων στα σύμβολα της γραμματικής και την ύπαρξη των λεγόμενων σημασιολογικών κανόνων, οι οποίοι ορίζουν τις τιμές των ιδιοτήτων και πιθανά περιορισμούς σε αυτές. Οι γραμματικές ιδιοτήτων χρησιμοποιούνται ευρύτατα στην κατασκευή μεταγλωττιστών και επιτρέπουν, μέσω των DCG, να παράγεται από το συντακτικό δένδρο επιπλέον πληροφορία. Τα επιπλέον ορίσματα προστίθενται σαν πρώτα στη εσωτερική Prolog αναπαράσταση των ορισμών.

Τη δυνατότητα αυτή μπορούμε να την εκμεταλλευτούμε έτσι ώστε η ερώτηση, εκτός από την επιτυχία στη αποδοχή μιας πρότασης της γραμματικής, να επιστρέφει και το δένδρο σύνταξης. Έτσι, η DCG διαμορφώνεται ως εξής:

```
sentence( s( NP, VP ) ) -->
  noun_phrase( NP ), verb_phrase( VP ).
noun_phrase( np( A, N ) ) -->
  article( A ), noun( N ).
verb_phrase( vp( V, NP ) ) -->
  verb( V ), noun_phrase( NP ).
noun( n(cat) ) --> [cat].
noun( n(dog) ) --> [dog].
noun( n(man) ) --> [man].
noun( n(monkey) ) --> [monkey].
noun( n(banana) ) --> [banana].
article( art(a) ) --> [a].
article( art(the) ) --> [the].
verb( v(chases) ) --> [chases].
verb( v(eats) ) --> [eats].
```

Η ερώτηση:

```
?- sentence( SyntaxTree, [the, monkey, eats, a, banana], [] ).
```

επιστρέφει επιτυχία αλλά και το δένδρο σύνταξης με μορφή ενός σύνθετου όρου:

```
SyntaxTree = s(np(art(the),n(monkey)),
vp(v(eats),np(art(a),n(banana))))
```

ο οποίος αν αναλυθεί γραφικά σχηματίζει το δένδρο σύνταξης όπως παρουσιάστηκε παραπάνω.

Εδώ μπορείτε να δείτε ένα σύντομο βίντεο με τη υλοποίηση και δοκιμή του προγράμματος.

Ανάμιξη Prolog με φράσεις DCG

Πολλές φορές είναι επιθυμητό να εμπλέξουμε κατηγορήματα Prolog που εκτελούν κάποιες διαδικασίες μέσα σε κανόνες DCG, δηλαδή να υλοποιούμε σημασιολογικούς κανόνες μέσω Prolog κατηγορημάτων. Αυτό είναι εφικτό με τη χρήση της ανοικτής και κλειστής αγκύλης στο δεξιό μέρος μιας φράσης DCG:

μη τερματικό σύμβολο -->

σειρά τερματικών ή μη τερματικών συμβόλων, {κλήσεις σε Prolog κατηγορήματα}

Για παράδειγμα, ο παρακάτω κανόνας επιστρέφει την αριθμητική τιμή ενός λεκτικού (πχ έξι εκατοντάδες):

```
numberword(N) --> digitword(D), [hundred], {N is D*100}.
digitword(1) --> one.
…
digitword(6) --> six.
…
```

Στο παράδειγμα ο σημασιολογικός κανόνας {N is D*100} υπολογίζει την τιμή της "ιδιότητας" N του συμβόλου numberword, βάσει της ιδιότητας D του συμβόλου digitword. Η ερώτηση που επιστρέφει την τιμή του λεκτικού six hundred είναι:

```
?- numberword(N, [six, hundred],[]).
X = 600
```

Στην παραπάνω γραμματική, οι πολλαπλές φράσεις noun/3, μπορούν να αντικατασταθούν με μία:

```
noun( n(X) ) --> [X],{member(X,[cat,fog,man,...]}.
```

Αντί του member/2, η κλήση θα μπορούσε να παραπέμψει σε ένα πλήρες λεξικό ουσιαστικών.

Ένα άλλο παράδειγμα είναι οι κανόνες DGC που ορίζουν ποιοι χαρακτήρες είναι επιτρεπτοί σε προσδιοριστικά (identifiers), για παράδειγμα ονόματα μεταβλητών ή σταθερών μιας γλώσσας προγραμματισμού. Στη συγκεκριμένη περίπτωση οι επιτρεπτοί χαρακτήρες είναι όλα τα πεζά γράμματα, τα ψηφία 0 έως 9 και ο ειδικός χαρακτήρας "_":

```
allowedchar(L)-->    letter(L).
allowedchar(D)-->    digit(D).
allowedchar(S)-->    special_symbol(S).

letter(X)--> [X],
  {member(X,[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z])}.

digit(X)-->  [X],
  {member(X,[0,1,2,3,4,5,6,7,8,9])}.

special_symbol('_')-->    ['_'].
```

Λεκτικός Αναλυτής (Tokenizer) στην Prolog

Πολλές εφαρμογές αποδέχονται κανονικό κείμενο (μια ακολουθία από χαρακτήρες), και όχι λίστα από λέξεις. Αυτό σημαίνει ότι πρέπει να υπάρχει ένα ενδιάμεσο στάδιο μεταξύ της εισαγωγής του κειμένου και της επεξεργασίας του από μια DCG. Αυτό το βήμα ονομάζεται **Λεκτική Ανάλυση (Lexical Analysis)** και σκοπός του είναι να μετατρέψει μία πρόταση που απαρτίζεται από μια ακολουθία ASCII χαρακτήρων, οι οποίοι μπορεί να είναι γράμματα ή ψηφία, σημεία στίξης και άλλα σύμβολα χωρισμένα με κενά, σε μία λίστα από λέξεις, σημεία στίξης και σύμβολα (tokens). Δε θα παρουσιάσουμε με λεπτομέρεια κάποιον λεκτικό αναλυτή, καθώς οι βασικές ιδέες υλοποίησης έχουν παρουσιαστεί στο Κεφάλαιο 9 (ενότητα 9.5), και υπάρχουν πολλοί διαθέσιμοι ελεύθερα στον ιστό. Εμείς θα χρησιμοποιήσουμε έναν παλαιότερο του Clocksin & Mellish που δουλεύει σε οποιαδήποτε έκδοση της Prolog. Το κατηγόρημα ονομάζεται readin/1 και επιστρέφει τη λίστα με τις λέξεις και τα σύμβολα, τις οποίες πληκτρολογεί ο χρήστης. Για παράδειγμα:

```
?-readin(X).
|: this is a sentence with punctuation like ,
            and symbols like + and #.
[this,is,a,sentence,with,punctuation,like,',',
            and,symbols,like,'+',and,'#','.']
```

Στην περίπτωση της απλής γραμματικής, είναι θεμιτό να ορίσουμε ένα κατηγόρημα το οποίο θα διαβάζει μία πρόταση, θα αποχωρίζεται την τελεία και θα καλεί τη γραμματική DCG (κατηγόρημα sentence/3) για τον έλεγχο των τερματικών συμβόλων. Το κατηγόρημα parse_sentence/1 ορίζεται ως εξής:

```
parse_sentence(SyntaxTree):-
  readin(Tokens),
  sentence(SyntaxTree, S, ['.']).
```

Το κατηγόρημα καλείται μέσω της ερώτησης:

```
?-parse_sentence(SyntaxTree).
|:  the monkey eats a banana.
SyntaxTree = s(np(art(the), n(monkey)), vp(v(eats),
            np(art(a), n(banana))))
```

και έχει ως αποτέλεσμα τη λεκτική και συντακτική ανάλυση της εισαγόμενης πρότασης.

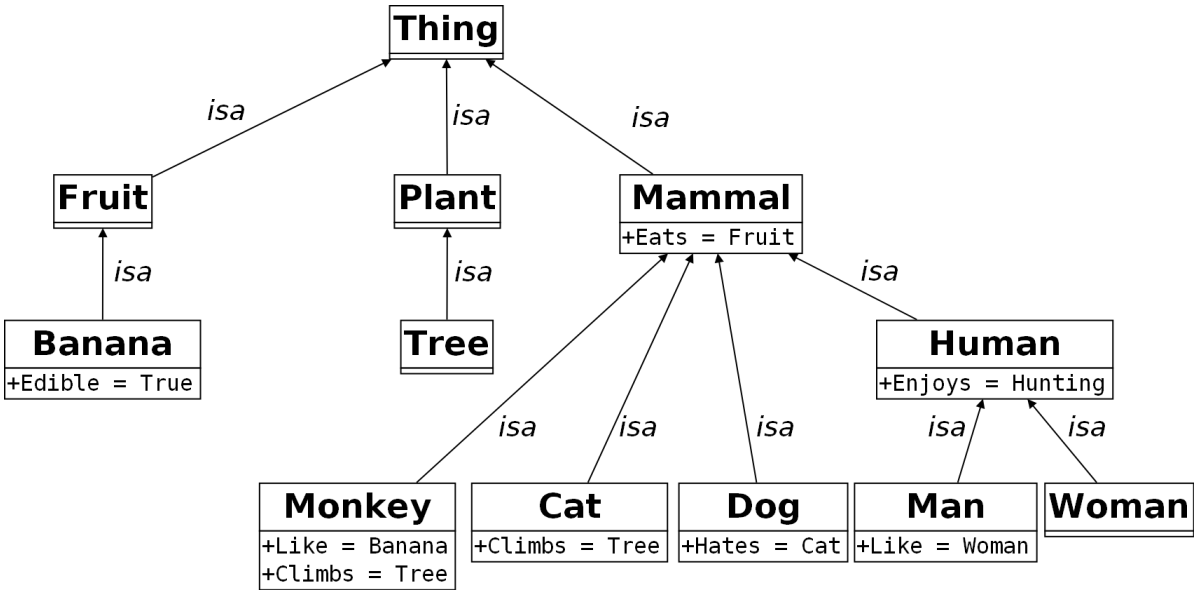
13.4 Έλεγχος της Σημασιολογίας μέσω Αναπαράστασης Γνώσης

Μέχρι τώρα παρουσιάστηκε η συντακτική ανάλυση των προτάσεων μίας γραμματικής, χωρίς να δοθεί έμφαση σε στοιχεία της γλώσσας που αφορά σε συμφωνία άρθρου, αριθμού, πτώσης, χρόνου κλπ. Το πρόβλημα αυτό αντιμετωπίζεται με περισσότερους κανόνες οι οποίοι φέρνουν σε συμφωνία μεταβλητές που αφορούν σε αυτά τα χαρακτηριστικά. Σε αυτήν την ενότητα, θα αντιμετωπίσουμε μερικώς το πρόβλημα της σημασιολογίας. Αυτό προκύπτει όταν συντακτικά σωστές προτάσεις της φυσικής γλώσσας δεν έχουν καμία εύλογη σημασία, όπως για παράδειγμα η πρόταση *"το δέντρο έφαγε τη μαϊμού"*.

Δομημένη αναπαράσταση γνώσης

Στην Τεχνητή Νοημοσύνη, η σημασιολογία είναι άμεσα συνδεδεμένη με την **Αναπαράσταση της Γνώσης (Knowledge Representation)**. Υπάρχουν πολλές μέθοδοι αναπαράστασης γνώσης που αναφέρονται με λεπτομέρεια σε συγγράμματα Τεχνητής Νοημοσύνης. Μία από τις πιο δημοφιλείς είναι τα **Πλαίσια (Frames)** που αποτέλεσε και προάγγελο του αντικειμενοστραφούς προγραμματισμού. Περιληπτικά, κάθε πλαίσιο αναπαριστά μία **κλάση (class)** ή ένα συγκεκριμένο **αντικείμενο (object)**. Κάθε πλαίσιο έχει **ιδιότητες (attributes**) με αντίστοιχες **τιμές (values)**. Το πλεονέκτημα των πλαισίων είναι ότι αυτά οργανώνονται σε ιεραρχική δομή όπου μια κλάση συνδέεται με άλλες υποκλάσεις με δεσμούς **"είναι" (is a ή ISA)**, **"είναι τύπου" (a kind of ή AKO)** και **"είναι στιγμιότυπο" (instance of)**. Αυτό δημιουργεί μια **κληρονομικότητα (inheritance)** ιδιοτήτων από μία υπερκλάση σε μία υποκλάση της.

Για παράδειγμα, το Σχήμα 13.3, απεικονίζει μία ιεραρχική δομή πλαισίων μόνο με δεσμούς ISA. Σε ορισμένα πλαίσια αναγράφονται ιδιότητες με τις τιμές τους. Θα χρησιμοποιήσουμε αυτή τη γνώση για να συνδέσουμε τις συντακτικά σωστές προτάσεις την απλής γραμματικής της προηγούμενης ενότητας με τη σημασία τους.



Σχήμα 13.3: *Αναπαράσταση γνώσης με μία ιεραρχική δομή πλαισίων*

Στην Prolog είναι σχετικά απλό να αναπαραστήσουμε αυτή τη δομή με τους παρακάτω ορισμούς των κατηγορημάτων frame/1 (πλαίσιο) και frame_prop/3 (πλαίσιο έχει ιδιότητα και τιμή):

```
frame(thing).
frame(mammal).
frame(plant).
...
frame_prop(mammal, isa, thing).
frame_prop(plant, isa, thing).
frame_prop(fruit, isa, thing).
```

```
....
frame_prop(monkey, likes, banana).
frame_prop(man, likes, woman).
frame_prop(mammal, eats, fruit).
frame_prop(banana, edible, true).
frame_prop(dog, hates, cat).
frame_prop(human, enjoys, hunting).
```

Ανάκτηση γνώσης σε Ιεραρχική δομή Πλαισίων

Στα πλαίσια, η γνώση (τιμή μίας ιδιότητας ενός πλαισίου) δεν είναι άμεσα προσβάσιμη. Υπάρχει, όμως, ένας μηχανισμός αναζήτησης της γνώσης μέσα στην ιεραρχία των πλαισίων, που μοιάζει πολύ με τους αναδρομικούς αλγορίθμους αναζήτησης διαδρομής σε γράφο (Κεφάλαιο 11). Ο αλγόριθμος αυτός εντοπίζει καταρχήν το πλαίσιο του οποίου την τιμή μιας ιδιότητας αναζητούμε. Αν βρεθεί σε αυτό το πλαίσιο, η τιμή επιστρέφεται, αν όχι αναζητούμε την τιμή σε υπερκλάση (κληρονομικότητα). Η υλοποίηση σε Prolog είναι:

```
property(O,A,V):-
  frame_prop(O,A,V),!.
property(O,A,V):-
  frame_prop(O,isa,SuperClass),!,
property(SuperClass, A, V),!.
```

Κανόνες Σημασιολογίας

Όμως στα πλαίσια η γνώση δεν εκφράζεται υπό προϋποθέσεις. Οι **Κανόνες EAN-TOTE (IF-THEN Rules)** έχουν αυτή τη δυνατότητα. Οι κανόνες είναι της μορφής:

ΕΑΝ *προϋπόθεση*$_1$ ***ΚΑΙ*** *...* ***ΚΑΙ*** *προϋπόθεση*$_N$ ***ΤΟΤΕ*** *συμπέρασμα*

Για παράδειγμα, κάποιος μπορεί να εκφράσει με κανόνες EAN-TOTE την παρακάτω γνώση για τις κλάσεις πραγμάτων που αναφέρθηκαν παραπάνω:

ΕΑΝ *το Χ είναι θηλαστικό* ***ΚΑΙ*** *στο Χ αρέσει το Υ* ***ΚΑΙ*** *το Υ είναι βρώσιμο* ***ΤΟΤΕ*** *το Χ τρώει το Υ*

ΕΑΝ *το Χ είναι θηλαστικό* ***ΚΑΙ*** *το Υ είναι θηλαστικό* ***ΚΑΙ*** *στο Χ αρέσει το Υ* ***ΚΑΙ*** *στο Χ αρέσει το κυνήγι* ***ΤΟΤΕ*** *το Χ κυνηγάει το Υ*

ΕΑΝ *το Χ είναι θηλαστικό* ***ΚΑΙ*** *το Χ μισεί το Υ* ***ΤΟΤΕ*** *το Χ κυνηγάει το Υ*

Σε συνδυασμό με τον αλγόριθμο ανάκτησης γνώσης από τα πλαίσια (property/3), οι κανόνες αναγράφονται σε Prolog ως ο ορισμός ενός κατηγορήματος (eligible_action/4), που αληθεύει αν δύο αντικείμενα Χ και Υ μπορούν να συνδεθούν με μια ενέργεια, πχ "το Χ τρώει το Υ" ή "το Χ κυνηγάει το Υ". Το τελευταίο όρισμα είναι ένας σύνθετος όρος που αναπαριστά την αιτιολογία του αποτελέσματος και θα χρησιμοποιηθεί αργότερα.

```
eligible_action(X,eats,Y,[X,isa,mammal]+[X,likes,Y]+
[Y,edible,true]):-
  property(X,isa,mammal),
  property(X,likes,Y),
  property(Y,edible,true).

eligible_action(X,chases,Y,[X,isa,mammal]+[Y,isa,mammal]+[X,likes,Y]+
[X,enjoys,hunting]):-
  property(X,isa,mammal),
  property(Y,isa,mammal),
  property(X,likes,Y),
  property(X,enjoys,hunting).

eligible_action(X,chases,Y,[X,isa,mammal]+[X,hates,Y]):-
  property(X,isa,mammal),
  property(X,hates,Y).
```

Τελική μορφή Γραμματικής με ενσωματωμένη Σημασιολογία

Έτσι λοιπόν, οι προτάσεις που είναι συντακτικά σωστές μπορούν να ελεγχθούν για σωστή σημασιολογία καλώντας το κατηγόρημα eligible_action/4 που αφορά στα δύο ουσιαστικά της πρότασης και το ρήμα που τα συνδέει:.

```
semantics(S):-
  S=[_,Object1,Verb,_,Object2],
  sentence(S, []),
  eligible_action(Object1, Verb, Object2, Justification),
  writelist(['Sentence: ',S,' is semantically correct',nl]),
  writelist(['Because ',Justification,nl,nl]).
```

Η παρακάτω ερώτηση μπορεί να δημιουργήσει όλες τις σημασιολογικά σωστές προτάσεις της απλής γραμματικής με βάση τη γνώση που υπάρχει για τα ουσιαστικά και τα ρήματα:

```
?-semantics(S).
Sentence: [a, dog, chases, a, cat] is semantically correct
Because [dog, isa, mammal]+[dog, hates, cat]

Sentence: [a, man, chases, a, woman] is semantically correct
Because [man, isa, mammal]+[woman, isa, mammal]+
            [man, likes, woman]+[man, enjoys, hunting]

Sentence: [a, monkey, eats, a, banana] is semantically correct
Because [monkey, isa, mammal]+[monkey, likes, banana]
           +[banana, edible, true]

Sentence: [the, dog, chases, a, cat] is semantically correct
Because [dog, isa, mammal]+[dog, hates, cat]

...
```

Το Σημασιολογικό Δίκτυο

Αυτά που αναφέρθηκαν υπαινίσσονται μία αφαιρετική εικόνα της δομής του **Σημασιολογικού Ιστού (Semantic Web)**. Ο Σημασιολογικός Ιστός βασίζεται στο μοντέλο δεδομένων **RDF (Resource Description Framework)**, το οποίο απεικονίζεται λεκτικά ως προτάσεις με την τριάδα υποκείμενο - κατηγόρημα - αντικείμενο, κάτι που αντιστοιχεί ευθέως με τον τρόπο που αναπαραστήσαμε προηγουμένως τις τιμές των ιδιοτήτων των πλαισίων με το κατηγόρημα frame_prop/3. Το ρόλο των πλαισίων στο Σημασιολογικό Ιστό παίζουν οι **Οντολογίες (Ontologies)** που αποτελούν ένα φορμαλισμό μίας κοινής κατανόησης πραγμάτων, εννοιών, συσχετίσεων, ιδιοτήτων και περιορισμών. Οι οντολογίες παρέχουν τύπους ή ομάδες αντικειμένων (κλάσεις) και επιτρεπτές ιδιότητες και σχέσεις μεταξύ τους, αλλά μόνο διμερείς σχέσεις, εκφρασμένες είτε με τη γλώσσα RDF Schema, με την οποία εκφράζονται απλές ιεραρχίες κλάσεων και ιδιοτήτων, καθώς επίσης και πεδία ορισμού / τιμών των ιδιοτήτων, ή με τη γλώσσα **OWL (Web Ontology Language)**, η οποία είναι ικανή να εκφράσει περισσότερους λογικούς περιορισμούς σχετικούς με τα αντικείμενα και τις ιδιότητές τους, συμπεριλαμβανομένης της λογικής άρνησης. Η γλώσσα OWL ανήκει στις λεγόμενες **περιγραφικές λογικές (description logics)** οι οποίες αποτελούν υποσύνολα της κατηγορηματικής λογικής πρώτης τάξης που παρουσιάστηκε στο Κεφάλαιο 3.

Τέλος, για τους κανόνες EAN-TOTE υπάρχουν διάφορες υποψήφιες πρότυπες γλώσσες κανόνων, όπως για παράδειγμα η γλώσσα SWRL (Semantic Web Rule Language), η οποία επεκτείνει τις οντολογίες OWL με **λογικούς** ή **συνεπαγωγικούς κανόνες** (**deductive rules**) χωρίς άρνηση. Είναι σημαντικό ότι κάποιες εκδόσεις της Prolog όπως η SWI-Prolog παρέχουν βιβλιοθήκες για την επεξεργασία RDF δεδομένων και Οντολογιών. Επίσης, είναι σημαντικό το γεγονός πως ο λογικός προγραμματισμός έχει επηρεάσει σημαντικά τον Σημασιολογικό Ιστό, σε σημείο που να έχουν οριστεί υποσύνολα της γλώσσας OWL, όπως για παράδειγμα η OWL 2 RL (rule language), τα οποία έχουν την δυνατότητα να εκφράσουν την σημασιολογία των οντολογιών χρησιμοποιώντας εξ' ολοκλήρου ένα σύνολο κανόνων λογικού προγραμματισμού.

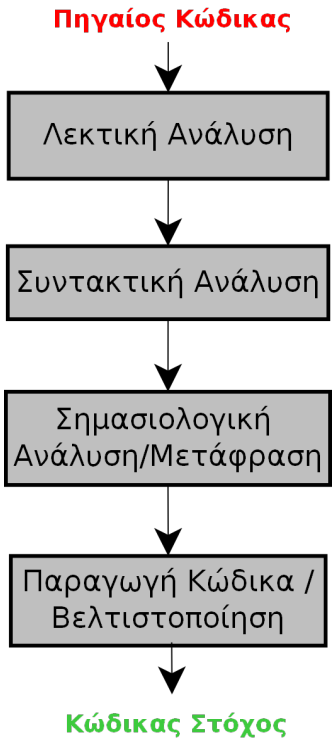
13.5 Μεταγλωττιστές

Τεχνητές Γλώσσες και Μεταγλωττιστές

Πολλές φορές είναι επιθυμητό να ορίσουμε μία γλώσσα ειδικού σκοπού με την οποία να κωδικοποιούμε προβλήματα με ευκολότερο τρόπο. Έτσι εξασφαλίζεται η καλύτερη διεπαφή, η εκφραστικότητα, η αναγνωσιμότητα, η ικανότητα γρήγορης τροποποίησης κλπ., όλα δηλαδή τα χαρακτηριστικά που θα ήθελε κάποιος από μια γλώσσα προγραμματισμού. Αν λοιπόν κάποιος θελήσει να δημιουργήσει τη δική του γλώσσα, είναι προφανές ότι πρέπει να κάνει επιλογές ως προς:

- τα σύμβολα και τις λέξεις κλειδιά και τη σημασία τους,
- τις ενσωματωμένες διαδικασίες, και
- τη γραμματική της γλώσσας.

Πρακτικά θα πρέπει να δημιουργήσει έναν μεταγλωττιστή, δηλαδή ένα πρόγραμμα που θα δέχεται πηγαίο κώδικα στη νέα γλώσσα (source code) και θα το μετατρέπει σε κώδικα μιας γλώσσας στόχο (target code) ο οποίος θα είναι άμεσα εκτελέσιμος. Τα στάδια μετατροπής του πηγαίου κώδικα σε κώδικα στόχο απεικονίζονται στο Σχήμα 13.4. Υπάρχουν πολλά εργαλεία που βοηθούν στην υλοποίηση ενός μεταγλωττιστή, όπως για παράδειγμα τα κλασικά εργαλεία lex/yacc, flex/bison, το ANTRL, αλλά εδώ θα εξετάσουμε τη χρήση των DCGs σε ένα απλό παράδειγμα.



Σχήμα 13.4: *Η δομή ενός μεταγλωττιστή*

Παράδειγμα: Μια γλώσσα για Μηχανές Πεπερασμένων Καταστάσεων

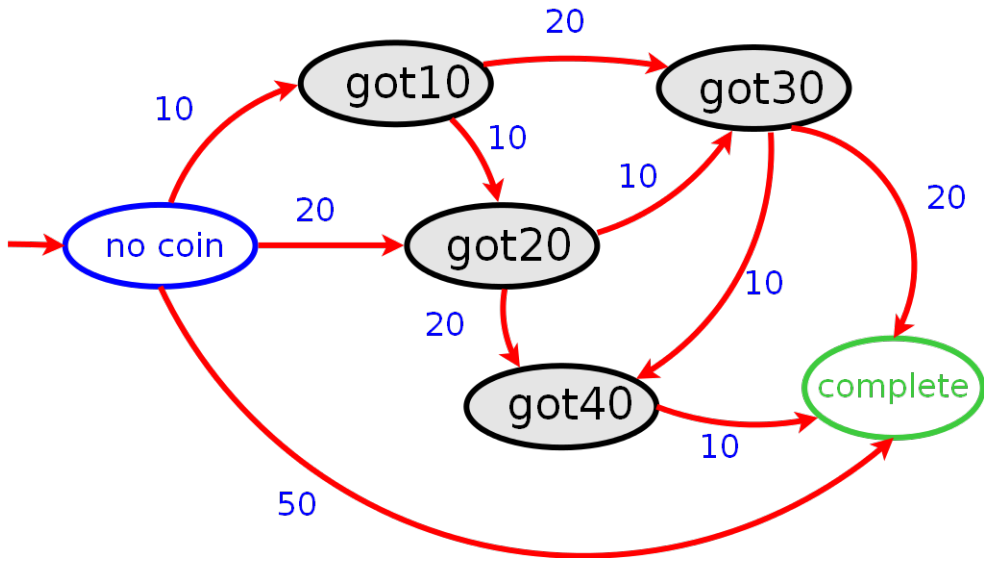
Μια ντετερμινιστική Μηχανή Πεπερασμένων Καταστάσεων (Finte State Machine) ορίζεται από:

- ένα πεπερασμένο σύνολο καταστάσεων (states),
- ένα πεπερασμένο σύνολο συμβόλων εισόδου (input symbols),
- την αρχική κατάσταση (initial state),

- κάποιες τελικές καταστάσεις (final states),
- ένα σύνολο από συναρτήσεις μεταβάσεων που δέχονται μία κατάσταση και ένα σύμβολο εισόδου και επιστρέφουν μία νέα κατάσταση (transition functions).

Για παράδειγμα, το Σχήμα 13.5 απεικονίζει μία μηχανή πεπερασμένων καταστάσεων για την αποδοχή 50 λεπτών από έναν αυτόματο πωλητή. Η αρχική κατάσταση είναι η no coin ενώ η τελική κατάσταση η complete στην οποία έχουν εισαχθεί συνολικά 50 λεπτά. Στις ενδιάμεσες καταστάσεις η μηχανή έχει δεχτεί κάποια κέρματα των 10,20 και 50 λεπτών (σύμβολα εισόδου). Προς στιγμή και για λόγους απλότητας παραλείπουμε στον ορισμό μεταβάσεις που οδηγούν σε απόρριψη κάποιου νομίσματος, για παράδειγμα την εισαγωγή 50 λεπτών στην κατάσταση got30.

Σκοπός μας είναι να δημιουργήσουμε μία γλώσσα περιγραφής τέτοιων μηχανών πεπερασμένων καταστάσεων, έτσι ώστε κάποιος με ευκολία να τα ορίζει. Αυτός θα είναι και ο πηγαίος κώδικας ο οποίος με μία διαδικασία μεταγλώττισης που θα ορίσουμε θα μετατρέπεται σε κώδικα Prolog που θα εκτελείται άμεσα. Στη συγκεκριμένη περίπτωση, εκτέλεση είναι η είσοδος μίας πεπερασμένης ακολουθίας συμβόλων που θα καταλήξει ή όχι σε μία τελική κατάσταση. Για παράδειγμα η ακολουθία <10,20,20> γίνεται αποδεκτή ενώ η <20,20> όχι. Η γλώσσα αυτή, ας την ονομάσουμε FSML, θα είναι ικανή να ορίσει όλα τα στοιχεία μίας μηχανής με εκφραστικό τρόπο. Δε θα υπεισέλθουμε στις λεπτομέρειες σχεδιασμού της γλώσσας, αφού οι σχεδιαστικές επιλογές είναι πολλές και η διαδικασία αυτή δεν εμπίπτει στο σκοπό αυτού του κεφαλαίου.



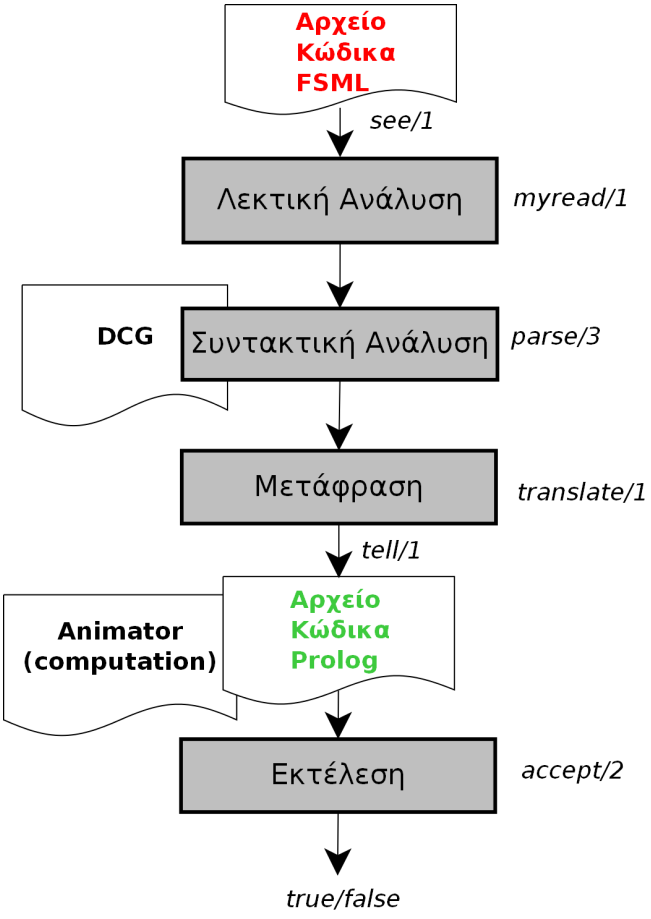
Σχήμα 13.5: *Μια Μηχανή Πεπερασμένων Καταστάσεων που αναπαριστά την αποδοχή 50 λεπτών σε έναν αυτόματο πωλητή.*

Ο πηγαίος κώδικας FSML αρχίζει με τη δήλωση του ονόματος της μηχανής και τελειώνει με το #accept και το #end. Η σειρά των υπολοίπων προτάσεων που ορίζουν τα στοιχεία της μηχανής δεν είναι σημαντική. Ας υποθέσουμε ότι ο πηγαίος κώδικας σε FSML για τη μηχανή του Σχήματος 13.5 είναι:

```
#fsm coin_machine.

#states = {no_coin, got10, got20, got30, got40, complete}.

#initial_state = no_coin.

#final_states = {complete}.

#input = {10, 20, 50}.

#transition from no_coin with 10 to got10.
#transition from no_coin with 20 to got20.
```

```
#transition from no_coin with 50 to complete.
#transition from got10 with 10 to got20.
#transition from got10 with 20 to got30.
#transition from got20 with 10 to got30.
#transition from got20 with 20 to got40.
#transition from got30 with 10 to got40.
#transition from got30 with 20 to complete.
#transition from got40 with 10 to complete.

#accept <10,10,20,10>.
#accept <10,20,10>.
#accept <10,10,10>.

#end.
```

Αυτός ο κώδικας πρέπει να διαβαστεί και να χωριστεί σε προτάσεις. Κάθε πρόταση πρέπει να ελεγχθεί για συντακτικά λάθη, δηλαδή αν υπακούει στην γραμματική της γλώσσας FSML, να δημιουργηθεί ένας ενδιάμεσος κώδικας που θα βοηθήσει στον έλεγχο των στοιχείων (πχ αν μια κατάσταση στις μεταβάσεις δεν έχει οριστεί στο σύνολο των καταστάσεων κλπ). Τέλος πρέπει να δημιουργηθεί ο κώδικας στόχος σε Prolog. Η διαδικασία της μεταγλώττισης απεικονίζεται στο Σχήμα 13.6. Στο κεφάλαιο αυτό θα δείξουμε μόνο τη DCG της γλώσσας FSML και κάποια μέρη κώδικα από τον υπόλοιπο μεταγλωττιστή.



Σχήμα 13.6: *Ένας μεταγλωττιστής για τη μετάφραση μιας γλώσσας που αναπαριστά Αυτόματα Πεπερασμένων Καταστάσεων (FSML) σε εκτελέσιμο κώδικα (Prolog), ο μηχανισμός εκτέλεσης και τα αντίστοιχα κατηγορήματα.*

Το κατηγόρημα parse/3 κάνει το συντακτικό έλεγχο μία πρότασης S (λίστας από σύμβολα που προέκυψαν από τον λεκτικό αναλυτή) και τη μετάφραση στη γλώσσα στόχο. Ο ορισμός του parse/3 έχει τόσες

περιπτώσεις όσες και οι διαφορετικού τύπου δηλώσεις της FSML και μία επιπλέον σε περίπτωση συντακτικού λάθους όταν ο συντακτικός έλεγχος αποτύχει:

```
parse(_,S,[]):-
  model(ModelName,S,[]),!,
  translate(fsm(ModelName)),!.

parse(_,S,[]):-
  init_state(InitState,S,[]),!,
  translate(initial_state(InitState)).

parse(_,S,[]):-
  final_states(FinalStates,S,[]),!,
  translate(final_states(FinalStates)).

parse(_,S,[]):-
  states(StateList,S,[]),!,
  translate(states(StateList)).

parse(_,S,[]):-
  transition(S1,F,S2,S,[]),!,
  translate(transition(S1,F,S2)).

parse(_,S,[]):-
  input(I,S,[]),!,
  translate(input(I)).

parse(_,S,[]):-
  accept(Query,S,[]),!,
  translate(Query).

parse(L,_,[]):-
  writelist([nl,'****** Syntax Error in ******',nl]),
  writelist([L,nl]),
  writelist(['****************************',nl,nl]),
  retract(errors(X)), X1 is X+1, assert(errors(X1)),!.
```

Οι παρακάτω φράσεις σε DCG ορίζουν τη γραμματική της γλώσσας FSML:

```
model(ModelName) --> ['#fsm'], constant(ModelName).

states(States)--> ['#states'], ['='], set(States).

init_state(InitState)--> ['#initial_state'], ['='],
constant(InitState).

final_states(FinalStates)--> ['#final_states'], ['='],
set(FinalStates).

input(InputSet)--> ['#input'], ['='], set(InputSet).

transition(S1,I,S2) -->
  ['#transition'], ['from'],  constant(S1), ['with'],
  constant(I), ['to'],  constant(S2).

accept(accept(QueryParameter))--> ['#accept'],
sequence(QueryParameter).
```

Κάποιοι από τους κανόνες DCG παραπέμπουν σε άλλα μη τερματικά σύμβολα, όπως το constant, set, sequence κλπ. που ορίζονται αντίστοιχα. Για παράδειγμα το set/1 ορίζεται μια σειρά από σταθερές constant/1 χωρισμένες με κόμμα και εγκλεισμένες ανάμεσα σε δύο αγκύλες {}:

```
set(L) --> ['{'],  set_elements(L), ['}'].

set_elements([Elem])--> constant(Elem).
set_elements([Head|Tail])--> constant(Head), [','],
set_elements(Tail).
```

Όπως αναφέραμε, ο κώδικας στόχος είναι η γλώσσα Prolog. Για παράδειγμα, ο πηγαίος κώδικας της FSML θα πρέπει να μετατραπεί στον παρακάτω κώδικα Prolog:

```
fsm(coin_machine).

states(coin_machine, [no_coin, got10, got20, got30, got40,
complete]).

initial_state(coin_machine, no_coin).

final_states(coin_machine,[complete]).

input(coin_machine,[10, 20, 50]).

transition(coin_machine, no_coin ,10, got10).
transition(coin_machine, no_coin ,20, got20).
transition(coin_machine, no_coin ,50, complete).
transition(coin_machine, got10 ,10, got20).
transition(coin_machine, got10 ,20, got30).
transition(coin_machine, got20 ,10, got30).
transition(coin_machine, got20 ,20, got40).
transition(coin_machine, got30 ,10, got40).
transition(coin_machine, got30 ,20, complete).
transition(coin_machine, got40 ,10, complete).

:-accept(coin_machine,[10,10,20,10]).
:-accept(coin_machine,[10,20,10]).
:-accept(coin_machine,[10,10,10]).
```

Το κατηγόρημα accept/2 δέχεται μία λίστα από σύμβολα και αληθεύει αν η μηχανή πεπερασμένων καταστάσεων αποδέχεται τη λίστα αυτή, δηλαδή αν μετά τις μεταβάσεις καταλήγει σε μία τελική κατάσταση.

Μετά το συντακτικό έλεγχο, τη μετάφραση αναλαμβάνει το κατηγόρημα translate/1 που σε αυτή την περίπτωση είναι ένα απλό write του όρου που έχει επιστρέψει η γραμματική:

```
translate(accept(Q)):-!,
  writelist([':-',accept(Q),'.',nl]),!.

translate(X):-
  write(X),write('.'),nl,!.
```

Τέλος, όλη η διαδικασία απαιτεί:

- διαχείριση αρχείων (άνοιγμα και διάβασμα από το αρχείο του πηγαίου κώδικα, άνοιγμα και γράψιμο στο αρχείο του κώδικα στόχου),
- λεκτική ανάλυση των προτάσεων της FSML σε σύμβολα,
- μεταγλώττιση μία προς μία όλων των προτάσεων του πηγαίου κώδικα έως τη λέξη κλειδί #end, και
- αναφορά συντακτικών λαθών.

Το κυρίως κατηγόρημα compile/0 ορίζεται ως εξής:

```
compile:-
  writelist([nl,'   ** FSML Compiler **',nl,nl]),
  writelist(['Input Filename and extention (in single quotes) :']),
  read(IF),
  writelist(['Compiling ', IF, '......', nl,nl]),
```

```
  initialise_errors,
  writelist(['Target Filename (.pl) :']),
  read(OF),
  see(IF),
  tell(OF),
  writelist([':-[fsmlanimator].', nl,nl]),
  compile_code,
  told,
  report_errors,!.

compile_code:-
  repeat,
  myread(Statement,S),
  (S = ['#end'] ->
   (seen,!);
   (parse(Statement,S,[]),
    fail)).
```

Θα παρατηρήσατε ότι στον κώδικα στόχο τοποθετείται ως πρώτη γραμμή η εντολή:

```
:-[fsmlanimator].
```

Αυτή είναι απαραίτητη ώστε ο κώδικας στόχος (Prolog κώδικας που αναπαριστά μία μηχανή πεπερασμένων καταστάσεων) να εκτελεστεί, δηλαδή για παράδειγμα η ερώτηση:

```
:-accept(coin_machine,[10,10,20,10]).
```

η οποία καθορίζει αν η λίστα συμβόλων εισόδου είναι αποδεκτή από τη μηχανή. Το κατηγόρημα accept/2 υλοποιεί το **μηχανισμό εκτέλεσης (computation)** μιας μηχανής πεπερασμένων καταστάσεων:

```
accept(Name,Sequence):-
  fsm(Name),
  write('Accepting for '), write(Name),
  write(' Sequence: '), write(Sequence),nl,
  initial_state(I),
  accept(I,Sequence).

accept(S,[]):-
  final_states(FS),
  member(S,FS),!,
  write('Sequence Accepted'),nl,nl.

accept(S,[I|Rest]):-
  transition(S,I,NextS),!,
  accept(NextS,Rest).

accept(_,_):-
  write('Sequence NOT Accepted'),nl,nl.
```

Όπως αναφέρθηκε, συνήθως στη μεταγλώττιση υπάρχει και ένα στάδιο που ελέγχει και τη συνέπεια μεταξύ των φράσεων της γλώσσας. Για παράδειγμα στην FSML θα μπορούσαμε να ελέγχουμε αν οι καταστάσεις που ορίζονται στο #transition έχουν δηλωθεί στο #states. Αν όχι, αυτό δεν αποτελεί μεν συντακτικό λάθος αλλά "λογικό" λάθος. Κάτι τέτοιο θα απαιτούσε ένα δεύτερο πέρασμα από έναν ενδιάμεσο κώδικα που θα δημιουργούνταν για αυτόν ακριβώς το λόγο πριν το στάδιο της τελικής μεταγλώττισης.

13.6 Προχωρημένα Θέματα: Ορισμός και Χρήση Τελεστών

Το κατηγόρημα και ο όρος κατέχουν βασική θέση στο συντακτικό της Prolog. Όπως αναφέρθηκε, ένα κατηγόρημα και ένας όρος έχουν το ίδιο συντακτικό:

```
functor_name(argument1, …, argumentn)
```

και διαφέρουν μόνο στο ότι το κατηγόρημα έχει τιμή αληθείας (true ή false) ενώ ο όρος είναι απλά μία δομή δεδομένων. Ακόμη και παραστάσεις όπως:

```
a:-b,c
X is Y+5 mod 2
```

είναι στην πραγματικότητα της μορφής

```
:-(a, ,(b,c))
is(X, mod(+(Y,5),2)
```

αντίστοιχα. Μόνο που η γλώσσα μας επιτρέπει να τις γράφουμε σε μία πιο αναγνώσιμη μορφή επειδή τους έχει ορίσει σαν ενθεματικούς τελεστές, όπου τα ορίσματα αναγράφονται εκατέρωθεν του τελεστή. Όλα τα σύμβολα στην Prolog έχουν οριστεί ως τελεστές και έτσι η γλώσσα απέκτησε την τελική της μορφή ως προς το επιτρεπτό συντακτικό της.

Κάθε χρήστης, όμως, μπορεί να ορίσει τους δικούς του τελεστές, έτσι ώστε το πρόγραμμα (ή τουλάχιστον ένα κομμάτι του) να δείχνει πιο ευανάγνωστο και να πλησιάζει τη φυσική γλώσσα. Για παράδειγμα, ο κώδικας που είδαμε στο Κεφάλαιο 4:

```
follows(ilias,petros).
follows(petros, ilias).
...
friends(X,Y) :- follows(X,Y), follows(Y,X).
```

θα μπορούσε να γραφεί και ως:

```
ilias follows petros.
petros follows ilias.
...
X friend_with Y :- X follows Y, Y follows X.
```

Οι ερωτήσεις θα ήταν της μορφής:

```
?- ilias friend_with Whom.
Whom = petros
```

Η Prolog προσφέρει το κατηγόρημα write_canonical/1, το οποίο δέχεται στο όρισμά του μια παράσταση εκφρασμένη με τη χρήση τελεστών και εμφανίζει στην οθόνη την εσωτερική αναπαράσταση της σαν Prolog όρο. Για παράδειγμα:

```
?-write_canonical(petros follows ilias).
follows(petros, ilias).
true.
```

Ορισμός νέων τελεστών

Η Prolog προσφέρει την δυνατότητα ορισμού νέων τελεστών, με την εντολή op/3, η οποία έχει τη μορφή:

```
:-op(Τελεστής, Τύπος, Προτεραιότητα)
```

όπου:

- **Τελεστής** είναι το σύμβολο ή το άτομο που ορίζεται ως τελεστής,
- **Τύπος** είναι ένα από τα **xf**, **yf**, **xfx**, **xfy**, **yfx**, **fy** ή **fx**,
- **Προτεραιότητα** είναι ένας αριθμός, συνήθως από το 0 έως το 1200 (ο μικρότερος αριθμός δηλώνει υψηλότερη προτεραιότητα).

Το γράμμα f δηλώνει τη θέση του τελεστή, δηλαδή αν είναι:

- **προθεματικός (prefix)**, δηλαδή πριν από το όρισμα που δέχεται,
- **επιθεματικός (postfix)**, δηλαδή μετά το όρισμα που δέχεται, ή
- **ενθεματικός (infix)**, δηλαδή ανάμεσα σε δύο ορίσματα..

Τα γράμματα **x** και **y** ορίζουν την προσεταιριστικότητα (associativity) του τελεστή σημαίνουν ότι:

- το όρισμα που βρίσκεται στη θέση του x έχει χαμηλότερη προτεραιότητα,

- το όρισμα που βρίσκεται στη θέση του y έχει το πολύ ίση προτεραιότητα.

Ο τύπος και η προτεραιότητα των τελεστών καθορίζουν πως εκλαμβάνεται ή υπολογίζεται μία έκφραση. Ας υποθέσουμε ότι οι τελεστές - και * ορίζονται ως:

```
:-op(-,yfx,500).
:-op(*,yfx,400).
```

Εφόσον ο τελεστής - είναι τύπου yfx, άρα η έκφραση 9-4-3 είναι ισοδύναμη με την (9-4)-3 και όχι με την 9-(4-3), καθώς ο τελεστής είναι αριστερά προσεταιριστικός, δηλαδή στα αριστερά του μπορεί να υπάρχει έκφραση το πολύ ίσης προτεραιτότητας. Επίσης, η έκφραση 9-4*3 σημαίνει 9-(4*3) και όχι (9-4)*3 επειδή ο τελεστής * έχει μεγαλύτερη προτεραιότητα από τον τελεστή -. Εννοείται ότι ο τρόπος υπολογισμού των πράξεων σε μια παράσταση μπορεί να αλλάξει με τη χρήση παρενθέσεων, όπως αναφέρθηκε.

Χρήση τελεστών για βελτίωση της αναγνωσιμότητας

Για παράδειγμα, στο πρόγραμμα υπολογισμού του παραγοντικού η εντολή:

```
:-op(500,xfx,!).
```

ορίζει ένα νέο ενθεματικό τελεστή με όνομα ! (παραπέμπει στο μαθηματικό σύμβολο του ορισμού του παραγοντικού), έτσι ώστε ο ορισμός:

```
N ! F :- factorial(N,F).
```

να είναι συντακτικά αποδεκτός και να ενεργοποιείται με την ερώτηση:

```
?- 4 ! Result.
Result=24
```

Στο παράδειγμα των πλαισίων που δόθηκε παραπάνω, η χρήση των τελεστών μπορεί να κάνει τη δήλωση των ιδιοτήτων περισσότερο ευανάγνωστη, Αν οριστούν οι τελεστές:

```
:-op(60,fx,attribute).
:-op(50,xfy,of).
:-op(50,xfx,has_value).
```

δίνουν τη δυνατότητα να οριστεί η φράση:

```
attribute A of O has_value V :-
  property(O,A,V).
```

που καλεί το κατηγόρημα propety/3 που αναζητά τιμές ιδιοτήτων σε μια ιεραρχική δομή πλαισίων (ενότητα 13.4). Έτσι η ερωτήσεις μπορούν να γίνουν με πιο "φυσικό" τρόπο, χωρίς παρενθέσεις. Για παράδειγμα:

```
?- attribute eats of monkey has_value V.
V = banana
```

Σαν τελευταίο παράδειγμα, θα αναφέρουμε τη δυνατότητα που παρέχουν οι τελεστές για να οριστούν κανόνες EAN-TOTE, μία δημοφιλής μέθοδος αναπαράστασης γνώσης, με πιο αναγνώσιμη μορφή από αυτήν της Prolog,. Ένας κανόνας EAN-TOTE έχει τη μορφή:

N: ***if*** *Condition*$_1$ ***and*** *Condition*$_2$ ***or*** *....* ***and*** *Condition*$_N$ ***then*** *Conclusion.*

όπου *N* είναι ο αριθμός του κανόνα, *Condition*$_i$ οι συνθήκες του κανόνα (σύζευξη, διάζευξη κατηγορημάτων ή συνδυασμός) και *Conclusion* το συμπέρασμα του κανόνα (ένα κατηγόρημα). Όλα τα σύμβολα που συμμετέχουν στη σύνταξη του κανόνα (:, if, and, or, then) μπορούν να οριστούν ως τελεστές:

```
:-op(800,fx,if).
:-op(700,xfx,then).
:-op(400,xfy,and).
:-op(300,xfy,or).
:-op(200,fy,not).
```

Ας υποθέσουμε ότι θα θέλαμε να αναπαραστήσουμε τη γνώση που περιγράφεται το παρακάτω εγχειρίδιο ασφάλειας μιας αεροπορικής εταιρίας για το ποια αντικείμενα επιτρέπεται να πάρει ένας επιβάτης μαζί του στην καμπίνα του αεροπλάνου:

"Personal oxygen canisters also known as "canned oxygen," "recreational oxygen" and "flavoured oxygen" are forbidden in aircraft cabin and carry-on baggage, and checked baggage. Personal care items,

such as cologne and hairspray, are allowed in checked baggage without prior approval, as long as the volume is less than 16 ounces. Personal care items in smaller quantities, which comply with Transport Security Administration rules may also be placed in carry-on baggage. One butane curling iron per passenger is permitted in carry-on baggage. No refills are allowed and the safety cover must be on the curling iron. Butane curling irons are not permitted when travelling to Switzerland".

Οι κανόνες, με βάση τους τελεστές που ορίστηκαν παραπάνω, αναγράφονται στην Prolog ως εξής:

```
1: if item(X,canned_oxygen) or item(X,recreational_oxygen)
   or item(X,flavoured_oxygen) then item(X,oxygen_canister).
2: if item(X,oxygen_canisters) then allowed(X,carry_on,no).
3: if item(X,oxygen_canisters) then allowed(X,cabin,no).
4: if item(X,oxygen_canisters) then allowed(X,checked_baggage,no).
5: if item(X,cologne) or item(X,hairspray)
   then item(X,personal_care).
6: if item(X,personal_care) and volume(X,V) and prolog(V<16)
   then allowed(X,checked_baggage,yes).
7: if item(X,personal_care) and volume(X,V) and complies(V,tsa)
   then allowed(X,carry_on,yes).
8: if item(X,butane_curling_iron) and numberofitems(X,1)
   and item(X,refill)
   and safety_cover_on(X) and travels(X,C)
   and test(C\=switzerland)
   then allowed(X,carry_on,yes).
```

Το πρόβλημα φυσικά είναι ότι οι κανόνες αυτοί δεν είναι άμεσα εκτελέσιμοι σε Prolog. Χρειάζεται η υλοποίηση ενός μετα-διερμηνέα, όπως αυτοί που αναφέρθηκαν στο Κεφάλαιο 10. Αυτό δεν είναι βέβαια μόνο αναγκαίο αλλά πολλές φορές και επιθυμητό, γιατί ένας μετα-διερμηνέας δίνει τη δυνατότητα μαζί με την εκτέλεση να υπάρχει και επεξήγηση γιατί παράγεται κάποιο αποτέλεσμα ως αποτέλεσμα κάποιας συλλογιστικής. Οι επεξηγήσεις είναι γνωστές στα συστήματα γνώσης και ως ερωτήσεις why ή how.

Βιβλιογραφία

Η σημαντικότερη αναφορά στις γραμματικές είναι αυτή του Noam Chomsky που θεωρείται και ο "πατέρας" της μοντέρνας γλωσσολογίας (Chomsky, 1958 και Chomsky and Schützenberger ,1963). Όλα τα βιβλία Prolog αφιερώνουν μία τουλάχιστον ενότητα στις Γραμματικές Οριστικών Προτάσεων αλλά εδώ θα αναφέρουμε αυτό του Van Le (1993) το οποίο δεν είναι ευρέως γνωστό, αλλά περιέχει ενδιαφέροντα παραδείγματα και εφαρμογές. Μία τέτοια εφαρμογή για μεταγλωττιστές Μηχανών Πεπερασμένων Καταστάσεων με παρουσία Μνήμης παρουσιάζεται στο (Kefalas et al., 2003). Ενδιαφέρον παρουσιάζει η αποκάλυψη της IBM ότι η διασύνδεση φυσικής γλώσσας με τον υπολογιστή Watson είναι βασισμένη σε DCG και υλοποιημένη σε Prolog. Οι (Meditskos G. and Bassiliades N., 2008) παρουσιάζουν μία προσπάθεια για συλλογιστική μέσω κανόνων στον Σημασιολογικό Ιστό, έννοια που καθιέρωσε ο "ιδρυτής" του παγκόσμιου ιστού Tim Berners Lee (Berners-Lee T. et al., 2001).

Van Le, T. (1993). *Techniques of Prolog Programming with Implementation of Logical Negation and Quantified Goals,* John Wiley & Son.

Chomsky, N. (1959). On Certain Formal Properties of Grammars, *Information and Control,* Vol 2, pp.137-167.

Chomsky, N. and Schützenberger, M. (1963). The algebraic theory of context free languages, *Computer Programming and Formal Languages, North Holland*, pp.118-161.

ACM Learning Center (2013), *IBM Watson: Beyond Jeopardy!* [Available at: http://learning.acm.org/webinar/lally.cfm].

Lally A. and Fodor P. (2003), *Natural Language Processing With Prolog in the IBM Watson System,* Association for Logic Programming, [Available at: http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/].

Kefalas, P. and Eleftherakis, G. and Sotiriadou, A. (2003). Developing tools for formal methods. *Proceedings of the 9th Panhellenic Conference in Informatics.* pp.625-639.

Berners-Lee, T. and Hendler, J. and Lassila, O. (2001). The Semantic Web. *Scientific American,* Vol. 284, pp.34-43.

Meditskos, G. and Bassiliades, N. (2008). A Rule-based Object-Oriented OWL Reasoner, *IEEE Transactions on Knowledge and Data Engineering, IEEE,* Vol. *20*, No.3, pp. 397-410.

Άλυτες Ασκήσεις

13.1 Να ενισχυθεί η απλή γραμματική σε DCG της ενότητας 13.3 ώστε να είναι δυνατή και η αποδοχή πιο σύνθετων προτάσεων όπως αυτών που περιέχουν συνδετικά, όπως and, or, but.

13.2 Στη γραμματική υπάρχουν μεταβατικά (transitive) και αμετάβατα (intransitive) ρήματα. Στην απλή γραμματική DCG που ορίστηκε στην ενότητα 13.3 υπάρχουν μόνο μεταβατικά ρήματα. Να επεκτείνετε τη γραμματική αυτή ώστε να δέχεται προτάσεις με αμετάβατα ρήματα, για παράδειγμα "*η γάτα τρέχει*", "*ο σκύλος κοιμάται*" κλπ.

13.3 Στη φυσική γλώσσα, η συμφωνία ρήματος και αριθμού είναι σημαντική είτε στην Αγγλική ή στην Ελληνική. Για παράδειγμα, οι φράσεις "*οι σκύλοι κυνηγούν τη γάτα*" ή "*the dogs chase the cat*" έχουν συμφωνία μεταξύ του αριθμού (πληθυντικός) και του ρήματος ("κυνηγούν" αντί "κυνηγάει" και "chase" αντί "chases"). Να επεκτείνετε τη γραμματική της ενότητας 13.3 αυτή ώστε να δέχεται προτάσεις με συμφωνία αριθμού και ρήματος. Θα μπορούσατε να ξεκινήσετε από κάτω προς τα πάνω ορίζοντας τα τερματικά σύμβολα ως εξής:

```
article(singular) --> [a].
article(_)        --> [the].
...
noun(singular) --> [cat].
noun(plural)   --> [cats].
...
verb(singular) --> [chases].
verb(plural)   --> [chase].
...
```

13.4 Στην ενότητα 13.5 παρουσιάστηκε ένας μεταγλωττιστής της FSML. Στη μεταγλώττιση υπάρχει και ένα στάδιο που ελέγχει και συνέπεια μεταξύ των φράσεων μιας γλώσσας. Για παράδειγμα στην FSML θα έπρεπε να ελέγχουμε αν οι καταστάσεις που ορίζονται στο #transition έχουν δηλωθεί στο #states. Αν όχι, αυτό δεν αποτελεί μεν συντακτικό λάθος αλλά "λογικό" λάθος. Να επεκτείνετε το μεταγλωττιστή ώστε να κάνει ένα τέτοιο έλεγχο. Αυτό θα απαιτούσε:

- τη δημιουργία ενός ενδιάμεσου κώδικα (μέσω του ορισμού translate/1 όπου ο κώδικας στόχος θα γίνεται καταρχήν assert αντί write)
- τον έλεγχο λογικών λαθών
- τη δημιουργία του τελικού κώδικα στόχου διαγραφή (retract) μίας προς μία των προτάσεων του ενδιάμεσου κώδικα.

13.5 Στην ενότητα 13.5 παρουσιάστηκε ένας μεταγλωττιστής της FSML. Οι Μηχανές Πεπερασμένων Καταστάσεων ενδέχεται να έχουν και έξοδο (output) σε κάθε μετάβαση. Για παράδειγμα, όταν η μηχανή δέχεται ένα σύμβολο εισόδου και γίνεται μετάβαση από μία κατάσταση σε μία νέα κατάσταση, υπάρχει ταυτόχρονα και η έξοδος ενός συμβόλου, Να επεκταθεί η γλώσσα FSML, ο μεταγλωττιστής της και το πρόγραμμα της εκτέλεσης (accept/1) ώστε να ενσωματωθούν τα σύμβολα εξόδου.

13.6 Να οριστούν οι τελεστές follows και friends ώστε να είναι συντακτικά αποδεκτοί οι ορισμοί που παρουσιάστηκαν στην ενότητα 13.6.

13.7 Να οριστούν κατάλληλοι τελεστές (frame, frame_prop κλπ) ώστε ο ορισμός πλαισίων, όπως αυτών που παρουσιάστηκαν στην ενότητα 13.4 να γίνεται χωρίς τη χρήση παρενθέσεων.

13.8 Να υλοποιηθεί ένας μετα-διερμηνέας για ανάστροφη συλλογιστική κανόνων EAN-TOTE, όπως αυτοί που παρουσιάστηκαν στην ενότητα 13.6. Για παράδειγμα, αν τα δεδομένα είναι:

```
item(561,hairspray).
volume(561,10).
```

η εκτέλεση του μετα-διερμηνέα στην ερώτηση:

```
?-prove_bc(allowed(561,checked_baggage,yes)).
```

θα έχει το παρακάτω αποτέλεσμα:

```
The goal: allowed(561, checked_baggage, yes) matches the rule::
 IF item(561, personal_care)and volume(561, _G675)and
prolog(_G675<16) then allowed(561, checked_baggage, yes)

**Rule Start**
Trying to prove the conditions of the rule::
item(561, personal_care)and volume(561, _G675)and prolog(_G675<16)
Trying to first to prove:: item(561, personal_care)
 AND then volume(561, _G675)and prolog(_G675<16)
The goal: item(561, personal_care) matches the rule::
 IF item(561, cologne)or item(561, hairspray) then item(561,
personal_care)

**Rule Start**
Trying to prove the conditions of the rule::
item(561, cologne)or item(561, hairspray)
Trying to prove::
item(561, cologne) OR item(561, hairspray)
 Prooved:: item(561, hairspray)
Success of the rule:: IF item(561, cologne)or item(561, hairspray)
then item(561, personal_care)
**Rule End**

Trying to first to prove:: volume(561, _G675)
 AND then prolog(_G675<16)
 Prooved:: volume(561, 10)
 Prooved:: prolog(10<16)

Success of the rule:: IF item(561, personal_care)and volume(561,
10)and prolog(10<16) then allowed(561, checked_baggage, yes)
**Rule End**
```

Πίνακας Μεταφράσεων Όρων

Το κεφάλαιο περιέχει μεταφράσεις των όρων της Αγγλικής γλώσσας, όπως χρησιμοποιήθηκαν στο παρόν σύγγραμμα.

Αγγλικός Όρος	Ελληνικός Όρος
abort	εγκατάλειψη
abstract world	αφηρημένος κόσμος
and elimination	απαλοιφή σύζευξης
and introduction	εισαγωγή συζεύξεων
arc consistency	συνέπεια τόξου
argument	όρισμα, παράμετρος
argument	ισχυρισμός
ariable binding	δέσμευση μεταβλητής
arity	τάξη
assignment problems	προβλήματα ανάθεσης εργασιών
associativity	προσεταιριστικότητα
at run-time	την στιγμή της εκτέλεσης, δυναμικά
atom	άτομο
atomic type	ατομικός τύπος
backtracking	οπισθοδρόμηση
backtracking loop, failure-driven loop	βρόχος επανάληψης μέσω οπισθοδρόμησης
binary	δυαδικός
box model	μοντέλο κουτιού
branch and bound algorithm	αλγόριθμος επέκτασης και οριοθέτησης
break	διακοπή
built-in predicate	ενσωματωμένο κατηγόρημα
call	κλήση
canonical form	κανονική μορφή
chronological backtracking	χρονολογική οπισθοδρόμηση
clausal form	προτασιακή μορφή της λογικής
clause	φράση
combinatorial explosion	συνδυαστική έκρηξη
compilation	μετάφραση (κώδικα)
compiler	μεταφραστής (κώδικα)
complementary pairs	συμπληρωματικά ζεύγη

complete	πλήρης
computational logic	υπολογιστική λογική
conclusion	συμπέρασμα
conjunction	σύζευξη
conjunctive normal form	συζευκτική μορφή της λογικής
connective	συνδετικό
constraint	περιορισμός
constraint graph	γράφος περιορισμών
constraint satisfaction problem	πρόβλημα ικανοποίησης περιορισμών
context free languages	γλώσσες χωρίς συμφραζόμενα
context sensitive languages	γλώσσες με συμφραζόμενα
context-aware	προσαρμοστικός στις παραμέτρους
contradiction	αντίφαση
creep	ερπυσμός (βηματική παρακολούθηση)
cut	αποκοπή
data integrity constraint	περιορισμός ακεραιότητας δεδομένων
database schema	σχήμα βάσης δεδομένων
debugger	αποσφαλματωτής
decidable	καταληκτική
declarative	δηλωτικός
definite clause grammars	γραμματικές οριστικών προτάσεων
delayed goal	στόχος σε αναβολή
depth-first search	κατά βάθος αναζήτηση
destructive assignment	καταστροφική ανάθεση τιμής
destructive assignment	καταστροφική (εντολή) καταχώρισης
difference lists	λίστες διαφορών
disjunction	διάζευξη
disjunctive normal form)	διαζευκτική κανονική μορφή της λογικής
domain	πεδίο
domain dependent	εξαρτώμενο από την εφαρμογή
domain independent	ανεξάρτητο από την εφαρμογή
double negation elimination	απαλοιφή διπλής άρνησης
efficiency	αποδοτικότητα
entailment	συνεπαγώμενος
equivalence	ισοδυναμία
event	συμβάν (γεγονός)
exception handling	χειρισμός λαθών
existential introduction	εισαγωγή υπαρξιακού ποσοδείκτη

existential quantifier	υπαρξιακός ποσοδείκτης
exit	έξοδος
factorial	παραγοντικό
fail	αποτυχία
failure-driven loops	βρόχοι οδηγούμενοι από αποτυχία
finite domains	πεπερασμένα πεδία
first order predicate logic	κατηγορηματική λογική πρώτης τάξης
formula	τύπος
frames	πλαίσια
free variable	ελεύθερη μεταβλητή
functional term	συναρτησιακός όρος
functor	συναρτησιακό σύμβολο
global variables	καθολικές μεταβλητές
grounded	πλήρως ορισμένο
heuristic	ευρετικός
higher order logic	λογική ανώτερης τάξης
identifier	προσδιοριστικό
imperative	προστακτικός/διατακτικός
implication	συνεπαγωγή
infix operator	ενθεματικός τελεστής
interpretation	ερμηνεία
interpreter	διερμηνέας (κώδικα)
knowledge representation	αναπαράσταση της γνώσης
leap	άλμα
linear-bounded automata	μη-ντετερμινιστική μηχανή Turing με πεπερασμένο μήκος ταινίας
literal	λεκτικό, λεκτικό στοιχείο
logic proposition	λογική πρόταση
mathematical logic	μαθηματική λογική
memoing	απομνημόνευση
meta-interpreter	μετα-διερμηνέας
model	μοντέλο
modus ponens	τρόπος του θέτειν
most general unifier	γενικότερος ενοποιητής
node consistency	συνέπειας κόμβου
non-backtrackable	μη-αναστρέψιμο
non-terminal symbols	μη-τερματικά σύμβολα
occurs check	έλεγχος ύπαρξης

one way pattern matching	ταυτοποίηση
ontologies	οντολογίες
ontology	οντολογία
or introduction	εισαγωγή διαζεύξεων
pattern	πρότυπο
pattern matching	ταίριασμα μορφότυπων
postfix operator	επιθεματικός τελεστής
predicate	κατηγόρημα
predicate logic	κατηγορηματική λογική
prefix operator	προθεματικός τελεστής
premise	υπόθεση
prenex conjunctive normal form	προσημασμένη συζευκτική κανονική μορφή
procedural	διαδικαστικός
procedure box flow control model	μοντέλο κουτιού ελέγχου ροής εκτέλεσης διαδικασίας
production rules	κανόνες παραγωγής
prolog fact	γεγονός prolog
prolog rule	κανόνας prolog
proof	απόδειξη
proof procedure	διαδικασία απόδειξης
propositional logic	προτασιακή λογική
push-down automata	αυτόματα στοίβας
quantifier	ποσοδείκτης
query	ερώτηση
recursively enumerable languages	αναδρομικά απαριθμήσιμες γλώσσες
redo	επανάκληση
referential transparency	διαφάνεια αναφοράς
refutation, proof by contradiction	εις άτοπο απαγωγή
regular languages	κανονικές γλώσσες
resolution	αρχή της ανάλυσης
resolution principle	αρχή της ανάλυσης
resolvent	αναλυθέν
robustness	ευρωστία
rule of inference	κανόνας συμπερασμού
satisfiable	ικανοποιήσιμος
semantic web	σημασιολογικού δικτύου
semantic web	σημασιολογικός ιστός
semantics	σημασιολογία
shell	κέλυφος (γλώσσας προγραμματισμού)

singleton variable	μεμονωμένη μεταβλητή
skip	παράλειψη
skolemization	σκολεμοποίηση
sound	ορθή
specifications	προδιαγραφές
spy points	σημεία κατασκόπευσης
stack	στοίβα
state	κατάσταση
subgoal	υποστόχος
substitution	αντικατάσταση
symbolic logic	συμβολική λογική
syntax	σύνταξη
syntax tree	δένδρο σύνταξης
tail-recursion optimization	βελτιστοποίηση ουραίας κλήσης
tautology	ταυτολογία
term	όρος
term assignment	ανάθεση όρων
terminal symbols	τερματικά σύμβολα
theorem prover	λογισμικό απόδειξης θεωρημάτων, αποδείκτης θεωρημάτων
trace	παρακολούθηση (ίχνους)
truth table	πίνακα; αληθείας
tuple	πλειάδα
unary	μοναδιαίος
unification	ενοποίηση
unifier	ενοποιητής
universal elimination	απαλοιφή καθολικού ποσοδείκτη
universal quantifier	καθολικός ποσοδείκτης
unsatisfiable	μη-ικανοποιήσιμος
user interface	διασύνδεση χρήστη
valid argument	έγκυρος ισχυρισμός
verification	έλεγχος ορθότητας
well formed formula	ορθά δομημένος τύπος