ΠΑΡΑΡΤΗΜΑ 1

PROLOG και Τεχνητή Νοημοσύνη

Στο παράρτημα αυτό παρουσιάζεται η υλοποίηση διαφόρων θεμάτων ΤΝ που παρουσιάστηκαν θεωρητικά στο βιβλίο, όπως αλγορίθμων αναζήτησης, αναπαράστασης γνώσης με πλαίσια, ενός απλού συστήματος σχεδιασμού ενεργειών καθώς και μερικών εφαρμογών πρακτόρων. Για την υλοποίηση υιοθετήθηκε η γλώσσα λογικού προγραμματισμού PROLOG.

Για την καλύτερη κατανόηση των παραδειγμάτων, παρατίθεται στην αρχή ένα συνοπτικό εγχειρίδιο της γλώσσας. Το παράρτημα δεν έχει σκοπό την πλήρη διδασκαλία της PROLOG αλλά την παρουσίαση μόνο βασικών της. Η πλήρης μελέτη και κατανόηση όλων των παραδειγμάτων προϋποθέτει καλή γνώση της γλώσσας. Αν και θα μπορούσε να είχε χρησιμοποιηθεί άλλη γλώσσα προγραμματισμού, η συγκεκριμένη γλώσσα ενδείκνυται για εφαρμογές ΤΝ λόγω της εκφραστικότητας αλλά και απλότητας που παρέχει στην περιγράφη της γνώσης ενός προβλήματος, καθώς και της εξειδίκευσής της στο χειρισμό συμβόλων.

Σε πολλές περιπτώσεις παρουσιάζεται μόνο μέρος του κώδικα. Ο πλήρης κώδικας και όλα τα απαραίτητα αρχεία μαζί με οδηγίες για την εκτέλεση των προγραμμάτων είναι διαθέσιμα από την ιστοσελίδα του βιβλίου: https://aibook.gr. Στην ίδια ιστοσελίδα παρέχονται και σύνδεσμοι για διάφορες υλοποιήσεις της PROLOG, όπως SICSTUS PROLOG, LPA PROLOG, και SWI-PROLOG. Τα προγράμματα έχουν αναπτυχθεί χρησιμοποιώντας τα σταθερά χαρακτηριστικά της γλώσσας, γνωστά και ως *συντακτικό του Εδιμβούργου*, ώστε να είναι αποδεκτά από όλες τις εκδόσεις της γλώσσας που υποστηρίζουν αυτό το συντακτικό.

Π1.1 Εισαγωγή στη Γλώσσα PROLOG

Π1.1.1 Ιστορική Αναδρομή

Η PROLOG (PROgramming in LOGic) είναι μια συμβολική γλώσσα προγραμματισμού που βασίζεται στην κατηγορηματική λογική. Η ανάπτυξή της ξεκίνησε στη δεκαετία του '70 στην Ευρώπη, και βασίστηκε στις εργασίες των Colmerauer και Kowalski.

Η πρώτη υλοποίηση οφείλεται στη γαλλική ερευνητική ομάδα του Alain Colmerauer στο πανεπιστήμιο Luminy της Μασσαλίας, η οποία ανέπτυξε ένα πρόγραμμα απόδειξης θεωρημάτων (theorem prover), το οποίο θα χρησιμοποιούνταν για επεξεργασία φυσικής γλώσσας. Η υλοποίηση αυτή βασίστηκε στις εργασίες του Robert Kowalski, ο οποίος απέδειξε ότι οι *προτάσεις Horn*, ένα υποσύνολο της λογικής πρώτης τάξης, μπορούν να χρησιμοποιηθούν σαν βάση για την ανάπτυξη μιας νέας γενιάς γλωσσών προγραμματισμού. Και οι δύο προηγούμενες εργασίες βασίστηκαν στην *αρχή της ανάλυσης* (*Resolution Principle*), μια γενική αποδεικτική διαδικασία για τη λογική πρώτης τάξης, που είχε προταθεί από τον J.A. Robinson, στα μέσα της δεκαετίας του '60.

Σημαντικό ρόλο στη διάδοση της αποτέλεσε η υλοποίηση ενός μεταφραστή - διερμηνευτή της γλώσσας από τον D.H.D. Warren στο Πανεπιστήμιο του Εδιμβούργου (1977). Η υλοποίηση αυτή αποτέλεσε την πρώτη αποδοτική υλοποίηση της PROLOG, αποδεικνύοντας έτσι ότι είναι δυνατή η χρησιμοποίησή της σαν μιας γενικής γλώσσας προγραμματισμού, γεγονός το οποίο ήταν υπό αμφισβήτηση από πολλούς επιστήμονες την εποχή εκείνη ιδίως στην Αμερική. Η επιτυχία του μεταφραστή αυτού ήταν τόσο μεγάλη ώστε αποτέλεσε πρότυπο για τις επόμενες υλοποιήσεις, καθιερώνοντας παράλληλα τη σύνταξη της γλώσσας (Edinburgh Syntax).

Π1.1.2 Διαφορές Διαδικαστικών Γλωσσών και PROLOG

Σε ένα οποιοδήποτε πρόγραμμα, διακρίνουμε το τμήμα της λογικής και το τμήμα του ελέγχου. Σύμφωνα με την κλασική εξίσωση του Kowalski:

$$\textit{πρόγραμμα} = \textit{λογική} + \textit{έλεγχος}$$

Στις συμβατικές γλώσσες προγραμματισμού, όπως για παράδειγμα η C, η PASCAL, και η FORTRAN, το τμήμα της λογικής και το τμήμα του ελέγχου είναι αλληλένδετα και δεν διαχωρίζονται. Ο προγραμματιστής είναι επιφορτισμένος τις περισσότερες φορές με τον επακριβή καθορισμό της ροής ελέγχου του προγράμματος, ανάλογα με τη λογική και τα διαθέσιμα δεδομένα του προβλήματος. Αντίθετα, στην PROLOG γίνεται σαφής διαχωρισμός τους και απαιτείται να περιγραφεί μόνο η λογική του προς επίλυση προβλήματος ενώ ο έλεγχος αφήνεται στο σύστημα.

Π1.1.3 Προγραμματίζοντας στην PROLOG

Η λογική ενός προβλήματος, δηλαδή ενός προγράμματος, στην PROLOG είναι ένα σύνολο προτάσεων που περιγράφει τα δεδομένα του προβλήματος και τις σχέσεις που τα συνδέουν. Οι προτάσεις αυτές λέγονται προτάσεις Horn και αποτελούν υποσύνολο της κατηγορηματικής λογικής πρώτης τάξης.

Γεγονότα και Κανόνες

Υπάρχουν δύο είδη προτάσεων σε ένα πρόγραμμα PROLOG, τα γεγονότα και οι κανόνες. Τα γεγονότα εκφράζουν σχέσεις ανάμεσα στα αντικείμενα και αποτελούν κατά ένα τρόπο τα δεδομένα του προβλήματος. Για παράδειγμα, αν θέλουμε να δηλώσουμε ότι "*Ο Γιώργος είναι ο πατέρας της Μαρίας*", γράφουμε στην PROLOG ένα γεγονός της μορφής:

```
father(george,mary).
```

Οι κανόνες εκφράζουν γενικότερες σχέσεις ανάμεσα στα αντικείμενα οι οποίες ορίζονται με τη βοήθεια άλλων σχέσεων. Για παράδειγμα, η σχέση *γονιός*, ορίζεται με δύο κανόνες: "*Ο X είναι γονιός του Y εάν ο X είναι πατέρας του Y*" και "*Η X είναι γονιός του Y εάν η X είναι μητέρα του Y*", οι οποίοι σε PROLOG γράφονται ως εξής:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Τα **X**, **Y** που εμφανίζονται στους παραπάνω κανόνες είναι μεταβλητές. Το σύμβολο ":-" διαβάζεται σαν ΕΑΝ.

Αλληλεπίδραση με την PROLOG

Η PROLOG είναι κατεξοχήν μια *διερμηνευόμενη* γλώσσα (*interpreted*). Ο χρήστης αλληλεπιδρά με το σύστημα, δίνοντας σε αυτό *ερωτήσεις* (*queries*) ή *στόχους* (*goals*), μετά το προτρεπτικό σήμα (prompt) "**?-**", τις οποίες το σύστημα προσπαθεί να απαντήσει βάσει των γεγονότων και των κανόνων. Οι απαντήσεις που λαμβάνει είναι **yes** ή **no** (ναι ή όχι). Αν η ερώτηση περιέχει μεταβλητές, τότε η απάντηση περιλαμβάνει τις κατάλληλες τιμές για τις μεταβλητές αυτές, ώστε η ερώτηση να δέχεται καταφατική απάντηση. Σε περίπτωση που υπάρχουν περισσότερες από μία απαντήσεις στη συγκεκριμένη ερώτηση, ο χρήστης μπορεί να τις λάβει πατώντας διαδοχικά το πλήκτρο "**;**".

Παράδειγμα

Ένα απλό παράδειγμα προγράμματος σε PROLOG, είναι το ακόλουθο στο οποίο ορίζονται οι σχέσεις ανάμεσα στα μέλη μιας οικογένειας.

```
father(george,mary).
father(george,nick).
father(peter,marina).
mother(helen,mary).
mother(helen,nick).
mother(ann,marina).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Έστω ότι οι παραπάνω προτάσεις είναι αποθηκευμένες σε ένα απλό αρχείο κειμένου με όνομα "**family.pl**". Για να "φορτωθούν" σε κάποιον διερμηνευτή της PROLOG πρέπει να δοθεί η εντολή consult, συνοδευόμενη από το όνομα του αρχείου, ως εξής:

```
?- consult('family.pl').
```

Πιθανές ερωτήσεις που μπορεί να γίνουν με βάση το παραπάνω πρόγραμμα, καθώς και οι αντίστοιχες απαντήσεις, είναι οι εξής:

- Είναι ο **george** πατέρας της **mary**;

  ```
  ?- father(george,mary).
  yes
  ```

- Είναι ο **george** πατέρας της **marina**;

```
?- father(george,marina).
no
```

- Ποιος (**X**) έχει πατέρα τον **peter**;

```
?- father(peter,X).
X = marina
```

- Ποια (**X**) είναι η μητέρα της **marina**;

```
?- mother(X,marina).
X = ann
```

- Ποιου (**X**) είναι γονιός ο **george**;

```
?- parent(george,X).
X = mary;
X = nick;
no
```

Στην περίπτωση αυτή υπάρχουν δύο απαντήσεις. Το σύστημα επιστρέφει πρώτα την απάντηση **X=mary** και αν ο χρήστης πιέσει το πλήκτρο "**;**" επιστρέφεται η απάντηση **X=nick**. Αν ο χρήστης πιέσει ξανά το πλήκτρο "**;**" η PROLOG θα απαντήσει **no**, η οποία πρέπει να εκληφθεί ως "*δεν υπάρχουν άλλες εναλλακτικές απαντήσεις*".

- Ποιου (**X**) είναι γονιός ο **paul**;

```
?- parent(paul,X).
no
```

- Ποια είναι τα ζεύγη ατόμων **X**, **Y** , για τα οποία η **X** είναι μητέρα του **Y**;

```
?- mother(X,Y).
X = helen, Y = mary;
X = helen, Y = nick;
X = ann, Y = marina;
no
```

Εκτός από απλές ερωτήσεις, υπάρχουν και οι σύνθετες, οι οποίες χωρίζονται μεταξύ τους με κόμμα "," το οποίο αντιστοιχεί στο λογικό τελεστή AND. Η απάντηση σε μια σύνθετη ερώτηση είναι καταφατική μόνο αν αληθεύουν όλες οι επιμέρους κλήσεις που την αποτελούν.

- Οι **mary** και **nick** έχουν κοινό πατέρα; Αν ναι, ποιος (**X**) είναι αυτός;

```
?- father(X,mary),father(X,nick).
X = george
```

Η *κοινή μεταβλητή* (*shared variable*) **X** εκφράζει τη συνθήκη ότι οι **mary** και **nick** έχουν κοινό πατέρα. Γενικά, οι μεταβλητές στην PROLOG όταν εμφανίζονται παραπάνω από μία φορά σε μία ερώτηση, παίρνουν τιμή την πρώτη φορά που εμφανίζονται

και στη συνέχεια διατηρούν αυτήν την τιμή σε όλες τις επόμενες εμφανίσεις τους στην ίδια ερώτηση.

- Ποια είναι τα ζεύγη ατόμων **X**, **Y** , τα οποία έχουν για μητέρα τους την **helen**;

```
?- mother(helen,X),mother(helen,Y).
X = mary, Y = mary;
X = mary, Y = nick;
X = nick, Y = mary;
X = nick, Y = nick;
no
```

Από το παραπάνω παράδειγμα φαίνεται ότι η χρήση δύο διαφορετικών μεταβλητών δεν συνεπάγεται κατ' ανάγκη ότι αυτές πρέπει να πάρουν διαφορετικές τιμές.

Π1.1.4 Η Σύνταξη της PROLOG

Τα στοιχεία της γλώσσας είναι:

- οι όροι,
- τα γεγονότα,
- οι κανόνες,
- οι ερωτήσεις.

Ένα PROLOG πρόγραμμα είναι ένα σύνολο από προτάσεις της μορφής:

A.	*Γεγονός* (*fact*)
A :- B_1, B_2, ..., B_k.	*Κανόνας* (*rule*) (**k > 0**)
?- B_1, B_2, ..., B_k.	*Ερώτηση* (*question*) (**k > 0**)

Τα **A** και B_i ονομάζονται ατομικοί τύποι (όπως στην *Κατηγορηματική Λογική*) και είναι παραστάσεις της μορφής:

$p(t_1, t_2, \ldots, t_n)$

όπου το **p** ονομάζεται *κατηγόρημα* (*predicate*) και τα t_i ονομάζονται *ορίσματα* (*arguments*). Ο αριθμός των ορισμάτων (*n*) ονομάζεται *τάξη* (*arity*) του κατηγορήματος.

Τα ορίσματα ενός κατηγορήματος είναι *όροι* (*terms*) και μπορεί να είναι:

- *Σταθερές*, δηλαδή άτομο ή αριθμός
  - Οι αριθμοί έχουν τη συνήθη μορφή, κοινή και στις υπόλοιπες γλώσσες προγραμματισμού. Παραδείγματα αριθμών αποδεκτών από τη γλώσσα είναι: **2**, **3**, **6.7**, **-3**, **-10** κτλ.
  - Τα *άτομα* (*atoms*) είναι συμβολοσειρές που μπορεί να περιλαμβάνουν αριθμούς, οι οποίες όμως πρέπει απαραίτητα να ξεκινούν από πεζό γράμμα ή να περιλαμβάνονται σε μονά εισαγωγικά. Παραδείγματα αποδεκτών ατόμων είναι: **anna**, **x25**, **x_25**, **'Anna'**, κτλ.

- *Μεταβλητές*. Οι μεταβλητές στην PROLOG είναι συμβολοσειρές που μπορεί να περιέχουν ψηφία ή το χαρακτήρα "_". Χρησιμοποιούνται στη θέση άγνωστων ορισμάτων σε μια πρόταση και πρέπει πάντα να ξεκινούν με κεφαλαίο γράμμα ή με το χαρακτήρα "_". Παραδείγματα αποδεκτών μεταβλητών είναι: `X`, `Obj2`, `_X1`.
- *Σύνθετοι όροι*, δηλαδή δομή της μορφής $\mathtt{f(t_1, t_2, ..., t_\kappa)}$, όπου:
  - το `f` ονομάζεται *συναρτησιακό σύμβολο* (*functor*).
  - τα $\mathtt{t_i}$ ονομάζονται *ορίσματα* του συναρτησιακού συμβόλου και είναι όροι.
  - Ο αριθμός των ορισμάτων (κ) ονομάζεται *τάξη* (*arity*) του συναρτησιακού συμβόλου.

Σημειώνεται πώς στην PROLOG κάθε πρόταση πρέπει να τελειώνει με το σημείο στίξης της τελείας ".". Στους κανόνες ο ατομικός τύπος `A` ονομάζεται *κεφαλή* (*head*) του κανόνα, ενώ οι ατομικοί τύποι $\mathtt{B_i}$ αποτελούν το *σώμα* (*body*) του κανόνα. Η κεφαλή διαχωρίζεται από το σώμα με τους χαρακτήρες ":-", οι οποίοι μπορεί να διαβαστούν σαν το λογικό συνδετικό της συνεπαγωγής "ΕΑΝ". Η σημασία του σημείου στίξης του κόμματος "," μεταξύ των ατομικών τύπων στο σώμα ενός κανόνα, ή στην ερώτηση είναι αυτή της λογικής σύζευξης (AND).

Τα προγράμματα στην PROLOG μπορεί να γίνουν κατανοητά με δύο τρόπους: *δηλωτικά* (*declaratively*) και *διαδικαστικά* (*proceduraly*). Έτσι, αν θεωρήσουμε την πρόταση `P :- Q, R.`, αυτή μπορεί να ερμηνευτεί ως εξής:

- *Δηλωτικά*: Το `P` είναι αληθές εάν το `Q` και το `R` είναι αληθή.
- *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.

Π1.1.5 Εκτέλεση Προγραμμάτων Prolog

Η εκτέλεση ενός προγράμματος στην PROLOG ξεκινά με μια ερώτηση που υποβάλλει ο χρήστης και φτάνουμε σε λύση όταν έχουν εξαντληθεί όλες οι κλήσεις που αποτελούν την ερώτηση. Η απάντηση στην ερώτηση είναι το αποτέλεσμα του προγράμματος.

Επειδή τόσο οι κλήσεις μιας ερώτησης όσο και τα γεγονότα και οι κεφαλές των κανόνων του προγράμματος μπορεί να περιέχουν μεταβλητές στην προσπάθεια της η PROLOG να ελέγξει αν μία κλήση ικανοποιείται από μία πρόταση του προγράμματος χρησιμοποιεί το μηχανισμό *ενοποίησης* (*unification*), παρόμοια με την κατηγορηματική λογική. Αυτός προσπαθεί να καταστήσει ταυτόσημες μια κλήση μιας ερώτησης και την κεφαλή μιας πρότασης του προγράμματος εκτελώντας τις ελάχιστες απαραίτητες αναθέσεις τιμών σε μεταβλητές ή, όπως λέγεται στην ορολογία της PROLOG, υπολογίζοντας τον *πιο γενικό ενοποιητή* (*most general unifier*).

Όταν η κλήση που εξετάζεται ενοποιείται με ένα από τα γεγονότα του προγράμματος τότε αυτή ικανοποιείται και απομακρύνεται από την ερώτηση. Αν η τρέχουσα κλήση ενοποιείται με κάποιον κανόνα, τότε αυτή απομακρύνεται από την ερώτηση και τη θέση της παίρνει το σώμα του κανόνα αυτού, οπότε για την ικανοποίηση της αρχικής κλήσης είναι απαραίτητη η ικανοποίηση των κλήσεων του σώματος του κανόνα που την αντικατέστησε.

Αν στο πρόγραμμα υπάρχουν περισσότερες της μιας προτάσεις με τις οποίες μπορεί να ενοποιηθεί η κλήση της ερώτησης, τότε η κλήση ενοποιείται με την πρόταση που εμφανίζεται πρώτη κατά σειρά στο πρόγραμμα. Το σημείο αυτό του προγράμματος ονομάζεται *σημείο οπισθοδρόμησης* και αντιπροσωπεύει πιθανές εναλλακτικές "απαντήσεις" στη συγκεκριμένη κλήση. Σε περίπτωση αποτυχίας εύρεσης λύσης ή σε περίπτωση που ο χρήστης ζητά και άλλη λύση, ο *μηχανισμός οπισθοδρόμησης* επιστρέφει στο τελευταίο σημείο οπισθοδρόμησης ακυρώνοντας τα υπολογιστικά βήματα που έπονται του σημείου αυτού και αναζητά στις επόμενες προτάσεις του προγράμματος κάποια που να μπορεί να ενοποιηθεί με την κλήση.

Αν με την εξάντληση της διαδικασίας αυτής η PROLOG δεν καταλήξει σε ερώτηση χωρίς κλήσεις, τότε η απάντηση στην αρχική ερώτηση είναι αρνητική (no) ή αλλιώς αποτυχία. Σε αντίθετη περίπτωση η απάντηση στην ερώτηση είναι θετική (yes) ή αλλιώς επιτυχής και συνοδεύεται από τις αναθέσεις τιμών στις μεταβλητές της αρχικής ερώτησης που προέκυψαν από τις διαδοχικές ενοποιήσεις.

Παράδειγμα

Έστω το πρόγραμμα:

```
Πρόταση 1: greek(socrates).
Πρόταση 2: human(turing).
Πρόταση 3: human(socrates).
Πρόταση 4: fallible(X) :- human(X).
```

και έστω ότι τίθεται το ερώτημα:

```
?- fallible(Y), greek(Y).
```

Το σύστημα ξεκινά με την πρώτη από αριστερά κλήση της ερώτησης που είναι η:

```
?- fallible(Y).
```

Αυτή ενοποιείται με την πρόταση 4 που είναι ένας κανόνας, οπότε η κλήση `?- fallible(Y)` αντικαθίσταται από το σώμα του κανόνα με τον οποίο ενοποιείται αφού γίνουν πρώτα οι κατάλληλες αντικαταστάσεις μεταβλητών {`Y=X`}. Η ερώτηση τώρα έχει ως εξής:

```
?- human(Y), greek(Y).
```

Στη συνέχεια, εξετάζεται η πρώτη από αριστερά κλήση και αυτή ενοποιείται με την πρόταση 2, με την αντικατάσταση {`Y=turing`}. Το σημείο αυτό είναι ένα σημείο οπισθοδρόμησης καθώς η κλήση `human(Y)` μπορεί να ενοποιηθεί και με την πρόταση 3 του προγράμματος. Η ερώτηση τώρα γίνεται:

```
?- greek(turing).
```

Η κλήση όμως αυτή δεν μπορεί να ενοποιηθεί με καμία από τις προτάσεις του προγράμματος. Στο σημείο αυτό το σύστημα οπισθοδρομεί στο προηγούμενο σημείο οπισθοδρόμησης, ακυρώνοντας την ανάθεση τιμών που έχει γίνει μετά από αυτό {`Y=turing`}. Η ερώτηση επανέρχεται στη μορφή:

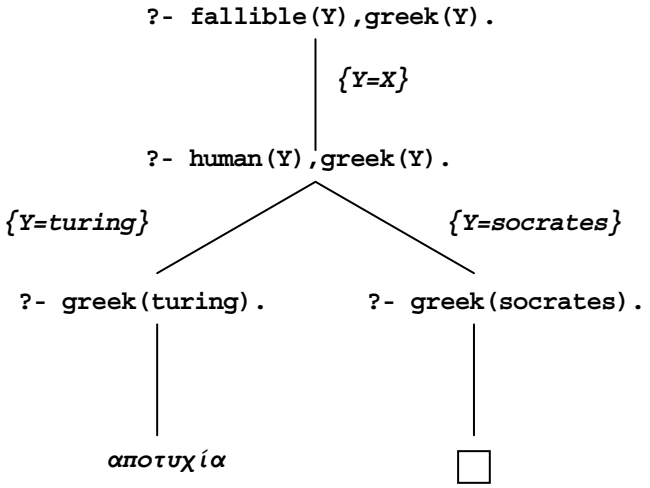
```
?- human(Y), greek(Y).
```

Η πρώτη από αριστερά κλήση ενοποιείται με την πρόταση 3. Η αντικατάσταση που προκύπτει από την ενοποίηση είναι {`Y=socrates`}, και η ερώτηση γίνεται:

```
?- greek(socrates).
```

Η κλήση αυτή ενοποιείται με την πρόταση 1 και επειδή δεν υπάρχουν προτάσεις που να μην έχουν εξετασθεί, δεν υπάρχουν εναλλακτικές λύσεις. Η μοναδική, λοιπόν, απάντηση στην αρχική ερώτηση είναι η `Y=socrates`.

Η διαδικασία που περιγράφηκε παραπάνω μπορεί να παρασταθεί με το δένδρο στο Σχήμα Π1.1 το οποίο ονομάζεται *δένδρο υπολογισμού* (*computation tree*).



Σχήμα Π1.1: Παράδειγμα δένδρου υπολογισμού.

Κάθε κόμβος του δένδρου αντιστοιχεί στην τρέχουσα ερώτηση. Οι συνδέσεις (κλαδιά) μεταξύ των κόμβων αντιπροσωπεύουν ένα υπολογιστικό βήμα και χαρακτηρίζονται από τις αντικαταστάσεις των μεταβλητών που γίνονται σε αυτό. Κάθε διαδρομή από τον αρχικό κόμβο μέχρι κάποιο φύλλο του δένδρου (τελικός κόμβος) καταλήγει είτε σε επιτυχία είτε σε αποτυχία. Οι διαδρομές που καταλήγουν σε επιτυχία αντιπροσωπεύουν τις εναλλακτικές απαντήσεις στην ερώτηση του χρήστη. Ο τελικός κόμβος στις διαδρομές αυτές είναι η κενή πρόταση, που συμβολίζεται με το □. Σε περίπτωση που μια κλήση ενοποιείται με περισσότερες από μια προτάσεις του προγράμματος, τότε από τον κόμβο αυτόν ξεκινούν περισσότερα του ενός κλαδιά. Το σημείο αυτό είναι ένα σημείο οπισθοδρόμησης.

Π1.1.6 Αναδρομή και Αναδρομικές Δομές

Η αναδρομή ως τεχνική προγραμματισμού

Η *αναδρομή* (*recursion*) αποτελεί βασικό στοιχείο στον προγραμματισμό με PROLOG. Με τον όρο αναδρομή εννοείται η δυνατότητα ένας κανόνας να περιέχει στο σώμα του μια κλήση προς τον εαυτό του. Οι κανόνες που χρησιμοποιούν αναδρομή χαρακτηρίζονται σαν αναδρομικοί κανόνες. Η χρήση της αναδρομής οδηγεί σε μικρότερα προγράμματα.

Παράδειγμα

Έστω ότι υπάρχει ένα σύνολο από γεγονότα της μορφής **parent(X,Z)**, τα οποία ερμηνεύονται ως ότι ο **X** είναι γονιός (πατέρας ή μητέρα) του **Z**.

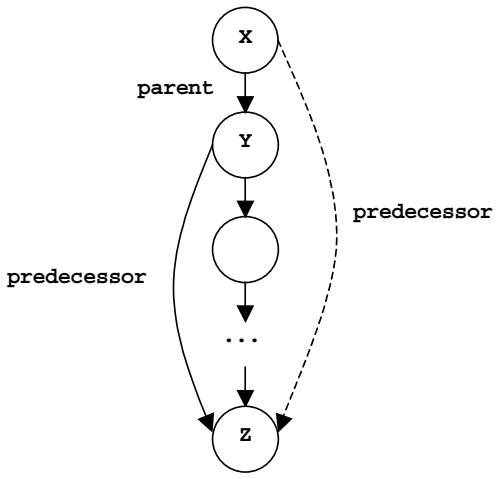
```
parent(john,george).
parent(john,nick).
parent(jim,bill).
parent(jim,jack).
parent(gregory,john).
parent(gregory,jim).
parent(bob,gregory).
parent(joseph,bob).
```

Ζητούμενο είναι να οριστεί έναν κατηγόρημα **predecessor(X,Z)**, το οποίο να αληθεύει εάν ο **X** είναι πρόγονος (πατέρας, παππούς, προ-παππούς κτλ) του **Z**. Αν δεν χρησιμοποιηθούν αναδρομικοί κανόνες, τότε θα πρέπει να γραφεί ένας κανόνας για κάθε διαφορετική γενεά, δηλαδή ένα κανόνας για τον πρόγονο-πατέρα, ένας κανόνας για τον πρόγονο-παππού, κ.ο.κ.

```
predecessor(X,Z) :- parent(X,Z).
predecessor(X,Z) :- parent(X,Y), parent(Y,Z).
predecessor(X,Z) :- parent(X,Y1), parent(Y1,Y2), parent(Y2,Z).
...
```

Προφανώς η παραπάνω λύση δεν είναι αποδεκτή, γιατί πρακτικά δεν μπορεί να γραφούν άπειροι κανόνες ώστε να καλύπτονται όλες οι γενεές. Η λύση είναι η χρήση ενός αναδρομικού κανόνα, ο οποίος θα διατρέχει τις γενεές ανεξάρτητα της απόστασής τους.

```
predecessor(X,Z) :- parent(X,Z).
predecessor(X,Z) :- parent(X,Y), predecessor(Y,Z).
```



Σχήμα Π1.2: Αναδρομική σχέση **predecessor**.

Η γενική ιδέα (Σχήμα Π1.2) είναι η εξής:

- Ο **X** είναι πρόγονος του **Z** όταν:
  - είτε ο **X** είναι πατέρας του **Z**,
  - είτε υπάρχει ένα **Y** τέτοιο ώστε ο **X** να είναι πατέρας του **Y** και ο **Y** να είναι πρόγονος του **Z**.

Εάν ζητήσουμε από το τελευταίο πρόγραμμα να μας εμφανίσει όλους τους πρόγονους του **george**, εισάγοντας την ερώτηση:

```
?- predecessor(X, george).
```

θα μας επιστρέψει τις ακόλουθες απαντήσεις:

```
X=john;
X=gregory;
X=bob;
X=joseph;
no
```

χωρίς να έχει πρόβλημα να εμφανίσει και άλλους προγόνους, εφόσον αυτοί είχαν δηλωθεί με γεγονότα **parent**.

Όταν ένα κατηγόρημα ορίζεται αναδρομικά, εμφανίζεται σχεδόν πάντα με τουλάχιστον δύο κανόνες. Από αυτούς, ο πρώτος συνήθως δεν περιέχει αναδρομική κλήση, λειτουργώντας έτσι σαν τερματική συνθήκη. Στο παράδειγμα που προηγήθηκε, η τερματική συνθήκη είναι ο πρώτος κανόνας του **predecessor**, ο οποίος απλά επιτυγχάνει εάν **X** είναι πατέρας του **Z**, χωρίς να έχει καμία αναδρομική κλήση του εαυτού του.

Αναδρομικές δομές δεδομένων (λίστες)

Οι σύνθετοι όροι χρησιμεύουν στην αναπαράσταση σύνθετων οντοτήτων ενός προβλήματος, όπως για παράδειγμα ο όρος:

```
triangle(point(0,0),point(1,4),point(5,12))
```

ο οποίος μπορεί να αναπαραστήσει ένα τρίγωνο. Από τη μορφή των σύνθετων όρων είναι φανερό ότι μοιάζουν με τις σύνθετες δομές δεδομένων των συμβατικών γλωσσών προγραμματισμού, όπως είναι οι *εγγραφές* (*records*) στην PASCAL ή οι *δομές* (*structs*) στην C.

Ιδιαίτερο ενδιαφέρον παρουσιάζουν οι αναδρομικές δομές δεδομένων, δηλαδή οι σύνθετοι όροι ο οποίοι ορίζονται αναδρομικά εγκλείοντας όρους της ίδιας μορφής με τον εαυτό τους. Μια ειδική κατηγορία αναδρομικών σύνθετων όρων είναι η *λίστα* (*list*). Πρόκειται για μια σύνθετη δομή που έχει συναρτησιακό σύμβολο την τελεία "`.`" ενώ τα ορίσματά της μπορεί να είναι οποιοιδήποτε όροι, ακόμα και άλλες λίστες. Μια λίστα μπορεί να μην έχει στοιχεία, οπότε ονομάζεται κενή λίστα και συμβολίζεται με `[]`. Ένα παράδειγμα λίστας με τρία στοιχεία είναι η `.(a,.(b,.(c,[])))`.

Οι λίστες είναι μια σχετικά απλή, αλλά ταυτόχρονα ιδιαίτερα σημαντική και ευρέως χρησιμοποιούμενη δομή δεδομένων της PROLOG, με την οποία μπορεί να αναπαρασταθούν συλλογές δεδομένων ως μία οντότητα. Μια λίστα στην PROLOG αναπαρίσταται με μια ακολουθία από οποιονδήποτε αριθμό στοιχείων. Τα στοιχεία της λίστας τοποθετούνται μέσα σε αγκύλες "`[`", "`]`" και χωρίζονται μεταξύ τους με κόμμα "`,`". Για παράδειγμα, η λίστα με τρία στοιχεία, το `a` το `b` και το `c` αναπαρίσταται ως `[a,b,c]`.

Μια λίστα μπορεί να είναι:

- κενή (δηλαδή μια δομή χωρίς όρους), η οποία συμβολίζεται με `[]`
- μια δομή με δύο όρους: την *κεφαλή* (*head*) που είναι το πρώτο στοιχείο της λίστας και την *ουρά* (*tail*) που είναι το υπόλοιπο τμήμα της λίστας.

Η λίστα που έχει κεφαλή Η και ουρά Τ παριστάνεται ως `[H|T]` και αντιστοιχεί στο σύνθετο όρο `.(H,T)`. Στη λίστα του παραδείγματος κεφαλή είναι το στοιχείο `a` και ουρά η λίστα `[b,c]`.

Η ουρά μιας λίστας είναι πάντα λίστα ενώ η κεφαλή μπορεί να είναι οποιοσδήποτε όρος, απλός ή σύνθετος, ακόμα και λίστα. Για παράδειγμα, η λίστα `[[a,b],c]` έχει κεφαλή τη λίστα `[a,b]` και ουρά τη λίστα `[c]`.

Ο χειρισμός των λιστών γίνεται συνήθως με αναδρομικούς κανόνες, οι οποίοι επιτελούν κάποια λειτουργία στην κεφαλή της λίστας και στη συνέχεια καλούν αναδρομικά τον εαυτό τους ώστε να επιτελέσουν την ίδια λειτουργία και στην ουρά της. Συνήθως υπάρχει και ένας τερματικός κανόνας ο οποίος ασχολείται με την περίπτωση της κενής λίστας.

Παράδειγμα - Διαπίστωση εάν ένας όρος είναι λίστα

Το κατηγόρημα `is_list` ελέγχει αν το όρισμά του είναι λίστα ή όχι. Αυτό επιτυγχάνεται ακολουθώντας τον ορισμό της λίστας που δόθηκε παραπάνω:

```
is_list([]).
is_list([Head|Tail]) :- is_list(Tail).
```

Η εξήγηση του παραπάνω ορισμού είναι η εξής: "*Ένας όρος είναι λίστα αν είναι η κενή λίστα, ή αν βρίσκεται στη μορφή [Κεφαλή|Ουρά], όπου η Ουρά πρέπει να είναι και αυτή με τη σειρά της λίστα*".

Παράδειγμα - Έλεγχος εάν ένα στοιχείο είναι μέλος μιας λίστας

Το κατηγόρημα `member` ελέγχει αν κάποιο στοιχείο υπάρχει σε μία λίστα. Αυτό μπορεί να επιτευχθεί με τη χρήση αναδρομικού κανόνα, ο οποίος βασίζεται στην παρατήρηση ότι "*ένα στοιχείο είναι μέλος μιας λίστας είτε αν είναι το πρώτο στοιχείο της λίστας* (*κεφαλή της λίστας*) *ή αν είναι μέλος της ουράς της*".

Το κατηγόρημα `member` δέχεται δύο ορίσματα. Το πρώτο όρισμα είναι είτε ένας όρος, είτε μια μεταβλητή. Το δεύτερο στοιχείο είναι η λίστα. Ανάλογα με τον τρόπο που θα κληθεί το κατηγόρημα, μπορεί να χρησιμοποιηθεί είτε για να ελέγξει αν ένα στοιχείο ανήκει σε μία λίστα ή για να επιστρέψει όλα τα στοιχεία της λίστας.

```
member(X,[X|Y]).
```

```
member(X,[Head|Tail]) :- member(X,Tail).
```

Ακολουθούν κάποια παραδείγματα ερωτήσεων:

```
?- member(a,[a,b,c]).
yes
?- member(b,[a,b,c]).
yes
?- member(d,[a,b,c]).
no
?- member(X,[a,b,c]).
X=a;
X=b;
X=c;
no
?-member(X,[]).
no
```

Π1.1.7 Ενσωματωμένα Κατηγορήματα

Η PROLOG, όπως παρουσιάζεται στις προηγούμενες ενότητες, είναι απόλυτα συμβατή με τον καθαρό λογικό προγραμματισμό και προκύπτει από την υλοποίηση σε υπολογιστή ενός διερμηνέα των προτάσεων Horn του υποσυνόλου δηλαδή της κατηγορηματικής λογικής πρώτης τάξης. Όμως η PROLOG εκτός από ένα εργαλείο απόδειξης θεωρημάτων χρησιμοποιείται και ως πρακτική γλώσσα προγραμματισμού γενικού σκοπού. Για το σκοπό αυτό είναι απαραίτητη η επέκτασή της με δυνατότητες που βρίσκονται πέρα (*extra-logical*) και πάνω (*meta-logical*) από τη μαθηματική λογική. Οι δυνατότητες αυτές παίρνουν τη μορφή των *ενσωματωμένων κατηγορημάτων* (*built-in predicates*) τα οποία παρέχονται από τον εκάστοτε κατασκευαστή της έκδοσης του διερμηνέα της PROLOG.

Τα ενσωματωμένα κατηγορήματα αφορούν συνήθως μαθηματικές πράξεις και συναρτήσεις, είσοδο/έξοδο σε αρχεία ή/και συσκευές, έλεγχο τύπου δεδομένων, διαχείριση λύσεων και προγράμματος, κτλ. Στην ενότητα αυτή παρουσιάζονται ενδεικτικά κάποια από τα πιο συνηθισμένα ενσωματωμένα κατηγορήματα. Περισσότερα μπορεί να αναζητηθούν στα εγχειρίδια χρήσης της εκάστοτε έκδοσης της PROLOG.

Ενοποίηση και σύγκριση όρων

Όπως αναφέρθηκε παραπάνω, η PROLOG κάνει ενοποίηση όρων κατά την προσπάθεια ικανοποίησης ενός στόχου. Υπάρχει όμως η δυνατότητα η ενοποίηση όρων να γίνει με σαφή κλήση του ενσωματωμένου κατηγορήματος "=" στο σώμα ενός κανόνα ή σε μια σύνθετη ερώτηση. Το κατηγόρημα αυτό αληθεύει αν οι δύο όροι μπορούν να ενοποιηθούν. Αν ναι, στη συνέχεια ενοποιούνται. Για παράδειγμα:

```
?- X = 5.
X = 5
?- X=Y.
```

```
X = Y = _
?- f(a,b) = f(a,b).
yes
?- f(X,b) = f(a,Y).
X=a, Y=y
?- f(a) = g(a).
no
```

Επίσης, υπάρχει και το αντίθετο κατηγόρημα "\=", το οποίο αληθεύει αν οι δύο όροι δεν μπορούν να ενοποιηθούν.

```
?- X \= 5.
no
?- X\=Y.
no
?- f(a,b) \= f(a,b).
no
?- f(X,b) = f(a,Y).
no
?- f(a) = g(a).
yes
```

Εκτός των κατηγορημάτων ενοποίησης υπάρχουν και τα κατηγορήματα σύγκρισης όρων "==" και "\==". Το κατηγόρημα "==" αληθεύει αν οι δύο όροι που συγκρίνει είναι ταυτόσημοι, δηλαδή έχουν την ίδια δομή και όλοι οι όροι που περιέχουν (αν είναι σύνθετοι) είναι ίδιοι. Μια μεταβλητή δεν είναι ταυτόσημη με μία σταθερά, αφού η πρώτη αντιπροσωπεύει μία θέση μνήμης ενώ η δεύτερη είναι ένα σύμβολο. Δύο διαφορετικές μεταβλητές δεν είναι ταυτόσημες, γιατί γενικά αντιπροσωπεύουν δύο ξεχωριστές "θέσεις μνήμης". Τέλος, το κατηγόρημα "\==" αληθεύει αν οι δύο όροι που συγκρίνει δεν είναι ταυτόσημοι.

Μαθηματικές διαδικασίες και συναρτήσεις

Στην PROLOG υπάρχει η δυνατότητα χειρισμού αριθμών και εκτέλεσης πράξεων με τη χρήση ενσωματωμένων κατηγορημάτων. Το πιο βασικό κατηγόρημα είναι το **is** με τη βοήθεια του οποίου αποτιμώνται αριθμητικές εκφράσεις και η τιμή τους ενοποιείται με κάποια μεταβλητή. Παραδείγματα χρήσης:

```
?- X is 3 + 4.
X=7
?- 9 is 3 * 3.
yes
?- 18 is 5 - 3.
no
```

Το κατηγόρημα **is** μπορεί να χρησιμοποιηθεί είτε για να αποδώσει τιμή σε κάποια μεταβλητή κάνοντας κάποιον μαθηματικό υπολογισμό ή για να ελέγξει αν η τιμή κάποιας μαθηματικής έκφρασης ισούται με κάποιον αριθμό. Στη μαθηματική έκφραση μπορούν να χρησιμοποιηθούν μεταβλητές μόνο αν έχουν πάρει προηγουμένως κάποια τιμή. Υπάρχουν και ενσωματωμένες συναρτήσεις (π.χ. ημίτονο, τετραγωνική ρίζα, κτλ) οι οποίες διαφέρουν από έκδοση σε έκδοση της γλώσσας.

Μία άλλη κατηγορία ενσωματωμένων κατηγορημάτων σχετικών με μαθηματικά είναι τα κατηγορήματα σύγκρισης **>**, **<**, κτλ. Η μοναδική ιδιομορφία της PROLOG σε αυτό το σημείο σε σχέση με τις άλλες γλώσσες είναι ότι ο τελεστής "*μικρότερο ή ίσο*" γράφεται ανάποδα "**=<**", για να μην θυμίζει "βέλος", το οποίο μπορεί να χρησιμοποιείται από κάποιες εκδόσεις ως τελεστής συνεπαγωγής.

Βασικές διαδικασίες εισόδου/εξόδου

Στην PROLOG, η επικοινωνία μεταξύ χρήστη και προγράμματος μπορεί να επιτευχθεί και με ενσωματωμένα κατηγορήματα τα οποία διαβάζουν όρους ή χαρακτήρες από το προκαθορισμένο κανάλι εισόδου (πληκτρολόγιο ή αρχείο) και επιστρέφουν το αποτέλεσμα στο προκαθορισμένο κανάλι εξόδου (οθόνη ή αρχείο).

Για την είσοδο όρων υπάρχει το κατηγόρημα **read(X)**, το οποίο διαβάζει τον επόμενο όρο που εισάγεται.

```
?- read(X).
|: prolog.
X = prolog
```

Η τελεία είναι απαραίτητη για να δηλώσει το τέλος του όρου. Η μεταβλητή **X** που υπάρχει στην αρχική κλήση ενοποιείται με τον όρο που πληκτρολογήθηκε.

Για την έξοδο όρων χρησιμοποιείται το κατηγόρημα **write(X)**, το οποίο τυπώνει τον όρο **X**. Για να αλλάξει η γραμμή εκτύπωσης χρησιμοποιείται το κατηγόρημα **nl**.

```
?- write(prolog), nl, write(lisp), nl.
prolog
lisp
yes
```

Για την είσοδο/έξοδο όρων σε αρχεία χρησιμοποιούνται τα ίδια κατηγορήματα αφού ανακατευθυνθούν πρώτα τα κανάλια εισόδου/εξόδου σε αρχεία ή/και συσκευές (για παράδειγμα εκτυπωτής). Τα κατηγορήματα που χρησιμοποιούνται για το σκοπό αυτό είναι τα **see**, **seen**, **tell**, **told** και πολλά άλλα που μπορεί να αναζητηθούν στα εγχειρίδια χρήσης της εκάστοτε έκδοσης της PROLOG.

Σύνθεση/διάσπαση σύνθετων όρων

Η PROLOG παρέχει τη δυνατότητα σύνθεσης και διάσπασης των σύνθετων όρων μέσω του κατηγορήματος "**=..**", το οποίο ονομάζεται *univ* (προφέρεται *γιουνίβ*). Το κατηγόρημα πετυχαίνει όταν το πρώτο όρισμά του είναι ένας σύνθετος όρος ενώ το δεύτερο μια λίστα, της οποίας το πρώτο στοιχείο είναι το συναρτησιακό σύμβολο του σύνθετου

όρου ενώ τα υπόλοιπα στοιχεία της είναι τα ορίσματά του. Η ακόλουθη κλήση της PROLOG κάνει αποσύνθεση ενός σύνθετου όρου σε λίστα:

```
?- parent(john,mary) =.. List.
List =.. [parent,john,mary]
```

Αν η μεταβλητή υπάρχει στο δεξί μέρος ενώ στο αριστερό υπάρχει λίστα, τότε το κατηγόρημα univ κάνει σύνθεση όρου:

```
?- P =.. [point,3,2].
P = point(3,2)
```

Έλεγχος τύπου δεδομένων

Οι μεταβλητές στην PROLOG δεν έχουν τύπο και άρα μπορούν να ενοποιηθούν με οποιονδήποτε όρο. Σε αρκετές όμως περιπτώσεις είναι απαραίτητο να καθορίζεται ο τύπος μιας μεταβλητής. Η PROLOG παρέχει ένα σύνολο κατηγορημάτων για το σκοπό αυτό όπως **var**, **nonvar**, **atom**, **integer** και πολλά άλλα που μπορεί να αναζητηθούν στα εγχειρίδια χρήσης της εκάστοτε έκδοσης της PROLOG.

Μεταβλητή κλήση

Η μεταβλητή κλήση μας δίνει τη δυνατότητα να τοποθετήσουμε μια μεταβλητή στη θέση μιας κλήσης στο σώμα ενός κανόνα ή ερώτησης. Η μεταβλητή αυτή παίρνει τιμή και εκτελείται κατά τη στιγμή της εκτέλεσης. Σε πολλές εκδόσεις της PROLOG υποστηρίζεται το κατηγόρημα **call(X)**. Συνήθως, η μεταβλητή κλήση συνδυάζεται με το κατηγόρημα univ για σύνθεση όρων. Έτσι δίνεται η δυνατότητα αρχικά να κατασκευαστεί δυναμικά μια κλήση με τη βοήθεια του univ και μετά να κληθεί με μεταβλητή κλήση.

```
?- X =.. [member,a,[a,b,c]], call(X).
X = member(a,[a,b,c])
```

Δυναμική τροποποίηση προγράμματος

Τα γεγονότα και οι κανόνες ενός προγράμματος PROLOG αποθηκεύονται στη μνήμη του συστήματος. Συνήθως αυτά τα γεγονότα και οι κανόνες "φορτώνονται" στη μνήμη της PROLOG από κάποιο αρχείο κειμένου. Υπάρχει όμως και η δυνατότητα τροποποίησης των περιεχομένων της μνήμης της PROLOG, με άλλα λόγια του προγράμματος, κατά τη διάρκεια εκτέλεσής του. Η προσθήκη/αφαίρεση στην περίπτωση αυτή γίνεται μέσω τριών κατηγορημάτων:

- **asserta(X)**: Προσθέτει στη μνήμη το **X** (γεγονός ή κανόνας) τοποθετώντας το πριν από τις ήδη υπάρχουσες προτάσεις με τον ίδιο κατηγόρημα και τον ίδιο αριθμό ορισμάτων.
- **assertz(X)**: Προσθέτει το **X** μετά από τις προτάσεις με το ίδιο κατηγόρημα και αριθμό ορισμάτων.
- **retract(X)**: Αφαιρεί από τη μνήμη το γεγονός ή τον κανόνα **X**.

Παράδειγμα

Έστω ότι στη μνήμη της PROLOG υπάρχει 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)).
yes
?- father(nick,X).
X = john;
X = mary
```

Αν προστεθεί ένα γεγονός κάτω από τα υπόλοιπα:

```
?- assertz(father(nick,anna)).
yes
?- father(nick,X).
X = john;
X = mary;
X = anna
```

Αν αφαιρεθεί ένα γεγονός:

```
?- retract(father(Y,mary)).
Y = nick
?- father(nick,X).
X = john;
X = anna
```

Μερικές υλοποιήσεις της PROLOG επιτρέπουν μόνο την εισαγωγή γεγονότων και όχι κανόνων, ενώ σε μερικές νεότερες εκδόσεις της γλώσσας, όπως για παράδειγμα στην SICStus PROLOG, την LPA PROLOG, και την SWI-PROLOG απαιτούνται δηλώσεις του τύπου:

```
:- dynamic functor/arity.
```

οι οποίες δηλώνουν ποιες διαδικασίες είναι δυνατό να τροποποιηθούν κατά τη διάρκεια εκτέλεσης του προγράμματος μέσω των κατηγορημάτων **assert** και **retract**.

Διαχείριση συνόλου λύσεων

Πολλές φορές είναι ιδιαίτερα χρήσιμο να επεξεργαστούν όλες οι εναλλακτικές λύσεις σε μια κλήση, όπως για παράδειγμα στις περιπτώσεις υλοποίησης αλγορίθμων αναζήτησης. Η PROLOG παρέχει τη δυνατότητα "συλλογής" των λύσεων σε λίστα μέσω τριών κατηγορημάτων: **bagof**, **setof** και **findall**.

Για παράδειγμα, το κατηγόρημα **findall(Var,Goal,List)** πετυχαίνει όταν στη λίστα **List** ως στοιχεία υπάρχουν όλες οι εναλλακτικές τιμές που μπορεί να πάρει η μεταβλητή **Var**, η οποία υπάρχει μέσα στο στόχο **Goal**, έτσι ώστε ο στόχος **Goal** να αληθεύει.

Αν, για παράδειγμα, υπάρχουν τα ακόλουθα δύο γεγονότα:

```
father(nick,anna).
father(nick,john).
```

και εκτελεστεί η ακόλουθη κλήση στην PROLOG:

```
?- findall(X,father(nick,X),List).
List = [anna,john]
```

η μεταβλητή **List** θα έχει σε λίστα όλα τα παιδιά του **nick**. Αν ο στόχος δεν ικανοποιείται τότε η **findall** επιστρέφει την κενή λίστα.

```
?- findall(X,father(john,X),List).
List = []
```

Αντί μεταβλητής, το πρώτο όρισμα της **findall** μπορεί να περιέχει οποιονδήποτε σύνθετο όρο, ο οποίος περιέχει μεταβλητές που υπάρχουν μέσα στην κλήση:

```
?- findall(child(X),father(nick,X),List).
List = [child(anna),child(john)]
```

Άρνηση ως αποτυχία

Μία από τις σημαντικότερες διαφορές της PROLOG από την κατηγορηματική λογική είναι η σημασία της *άρνησης* (*negation*). Στην κατηγορηματική λογική μπορεί να υπάρξουν αρνητικά γεγονότα, δηλαδή ατομικοί τύποι για τους οποίους δηλώνεται ρητά ότι είναι ψευδείς. Στην PROLOG δεν υπάρχει η δυνατότητα αναπαράστασης τέτοιας γνώσης, παρά μόνο θετικής γνώσης. Η ύπαρξη κάποιου γεγονότος δηλώνει την αλήθεια του, ενώ θεωρείται πως ό,τι δεν αναφέρεται ρητά στη μνήμη της PROLOG, τότε δεν ισχύει. Αυτό ονομάζεται "*υπόθεση κλειστού κόσμου*" (*closed-world assumption*) και δίνει τη δυνατότητα ύπαρξης ενός είδους περιορισμένης άρνησης, της "*άρνησης ως αποτυχίας*" (*negation-as-failure*).

Σύμφωνα με τα παραπάνω, οποιαδήποτε σχέση δεν μπορεί να αποδειχθεί από το σύστημα θεωρείται ως ψευδής. Η υλοποίηση αυτή της άρνησης διαφέρει σαφώς από την κλασική έννοια της άρνησης στη λογική, αλλά χρησιμοποιήθηκε καθώς απαλλάσσει τον προγραμματιστή από την υποχρέωση του ορισμού όλης την αρνητικής πληροφορίας για κάποια εφαρμογή.

Το κατηγόρημα **not** δέχεται ως όρισμα μια οποιαδήποτε κλήση της PROLOG. Αν η κλήση μπορεί να αποδειχθεί τότε το κατηγόρημα αποτυγχάνει. Αν όχι τότε πετυχαίνει.

```
?- not(member(a,[a,b,c])).
no
?- not(member(d,[a,b,c])).
yes
```

Πολλές εκδόσεις της PROLOG παρέχουν το κατηγόρημα \+ ως ισοδύναμο του **not** για να αποφευχθεί η σύγχυση της άρνησης της PROLOG με την άρνηση της λογικής.

Π1.2 Η PROLOG στην Τεχνητή Νοημοσύνη

Η PROLOG, λόγω της δηλωτικότητάς της και της ευκολίας χειρισμού συμβόλων που παρέχει, ενδείκνυται για την υλοποίηση εφαρμογών ΤΝ. Στη συνέχεια, παρουσιάζεται ενδεικτικά η υλοποίηση σε γλώσσα PROLOG κάποιων από τις τεχνικές και τους αλγορίθμους ΤΝ που αναπτύχθηκαν θεωρητικά στα προηγούμενα κεφάλαια του βιβλίου.

Π1.2.1 Αναπαράσταση Προβλημάτων

Στην ενότητα αυτή παρουσιάζεται η κωδικοποίηση διαφόρων προβλημάτων που αναφέρθηκαν στο ΜΕΡΟΣ Α του βιβλίου. Ουσιαστικά, πρόκειται για την κωδικοποίηση της αρχικής κατάστασης (**initial_state**), των τελικών καταστάσεων (**goal**) και των τελεστών μετάβασης (**operator**), έτσι ώστε να είναι δυνατή η επίλυσή του από τους αλγόριθμους αναζήτησης που θα περιγραφούν παρακάτω. Σε ορισμένα προβλήματα χρειάζεται και ο ορισμός της ευρετικής συνάρτησης (**heuristic**), όταν φυσικά για την επίλυσή τους εφαρμοστούν ευρετικοί αλγόριθμοι αναζήτησης.

Το πρόβλημα των ιεραποστόλων

Οι καταστάσεις αναπαρίστανται από ένα σύνθετο όρο με τρία ορίσματα: την κατάσταση της αριστερής και της δεξιάς όχθης και τη θέση της βάρκας. Η κατάσταση κάθε όχθης αναπαρίσταται από τον αριθμό των κανιβάλων και των ιεραποστόλων που βρίσκονται σε αυτήν.

```
initial_state(state(left(3,3),right(0,0), boat_left)). % Αρχική κατάσταση
goal(state(left(0,0),right(3,3), boat_right)). % Τελική κατάσταση

% Τελεστής μετακίνησης της βάρκας από την αριστερή (αφετηρία) στη
% δεξιά όχθη (προορισμός)
operator(state(left(ML,CL),right(MR,CR),boat_left),
                    state(left(NML,NCL),right(NMR,NCR),boat_right)) :-
   move(M, C),
   M =< ML, C =< CL,
   NML is ML-M, NCL is CL-C,
   NMR is MR+M, NCR is CR+C,
   valid(NML, NCL), valid(NMR, NCR).

% Τελεστής επιστροφής της βάρκας από τη δεξιά όχθη (προορισμός)
% στην αριστερή (αφετηρία).
operator(state(left(ML,CL),right(MR,CR),boat_right),
                    state(left(NML,NCL),right(NMR,NCR),boat_left)) :-
   move(M, C),
```

```
   M =< MR, C =< CR,
   NML is ML+M, NCL is CL+C,
   NMR is MR-M, NCR is CR-C,
   valid(NML, NCL), valid(NMR, NCR).

valid(M,C) :- M >= C, !.  % Πιστοποίηση αποδεκτής κατάστασης
valid(0,_).
```

Το πρόβλημα των ποτηριών

Οι καταστάσεις αναπαρίστανται με μία λίστα δύο ακεραίων που αντιπροσωπεύουν τον όγκο του υγρού που περιέχεται σε κάθε ποτήρι.

```
initial_state([0,0]).  % Αρχική κατάσταση
goal([_,40]).          % Τελικές καταστάσεις
goal([40,_]).

glass1(70).
glass2(50).

operator([V1,V2],[Vsum,0]) :-   % άδειασε όλο το περιεχόμενο του 2 στο 1
   V2 > 0,
   Vsum is V1 + V2,
   glass1(G1),
   Vsum =< G1.
operator([V1,V2],[0,Vsum]) :-   % άδειασε όλο το περιεχόμενο του 1 στο 2
   V1 > 0,
   Vsum is V1 + V2,
   glass2(G2),
   Vsum =< G2.
operator([V1,V2],[G1,Vdiff]) :- % άδειασε μέρος του 2 στο 1 για να γεμίσει
   V2 > 0,
   V1 >= 0,
   glass1(G1),
   Vdiff is V2 - ( G1 - V1 ),
   Vdiff > 0.
operator([V1,V2],[Vdiff,G2]) :- % άδειασε μέρος του 1 στο 2 για να γεμίσει
   V1 > 0,
   V2 >= 0,
   glass2(G2),
   Vdiff is V1 - ( G2 - V2 ),
   Vdiff > 0.
```

```
operator([V1,V2],[G1,V2]) :-    % γέμισε το 1 από τη βρύση
   glass1(G1), V1\=G1.
operator([V1,V2],[V1,G2]) :-    % γέμισε το 2 από τη βρύση
   glass2(G2), V2\=G2.
operator([V1,V2],[0,V2]).       % άδειασε το 1
operator([V1,V2],[V1,0]).    % άδειασε το 2
```

Το παζλ 3x3

Οι καταστάσεις αναπαρίστανται με λίστες 9 στοιχείων, όπως ακριβώς διαβάζονται τα πλακάκια κατά γραμμές από πάνω αριστερά μέχρι κάτω δεξιά. Το σύμβολο **e** υποδηλώνει το κενό.

```
initial_state([2,3,6,1,5,4,7,e,8]).   % Αρχική κατάσταση

goal([1,2,3,4,5,6,7,8,e]).         % Τελική κατάσταση

operator(P,NP):-                % Τελεστής μετάβασης
   find((X,Y,e), P),            % βρες συντεταγμένες του e
   move(X,Y,X1,Y1),             % ποια είναι μία πιθανή κίνηση του e
   find((X1,Y1,T),P),              % βρες ποιο βρίσκεται στη νέα θέση
   change(X,Y,X1,Y1,T,P,NP).    % άλλαξε θέση του e

move(X,Y,X,NY):-NY is Y-1,NY>0.    % Πιθανές κινήσεις του e
move(X,Y,X,NY):-NY is Y+1,NY<4.
move(X,Y,NX,Y):-NX is X-1,NX>0.
move(X,Y,NX,Y):-NX is X+1,NX<4.

heuristic(S,G,V):-              % Ευρετική συνάρτηση
   evaluate(S,G,V).             % S η τρέχουσα κατάσταση και G η τελική

evaluate([],[],0).
evaluate([(X,Y,T)|R],[(X,Y,T)|RF],V):-      % T στην τελική θέση του
   evaluate(R,RF,V).
evaluate([(X,Y,T)|R],[(X,Y,AT)|RF],V):-     % T σε κάποια άλλη θέση
   T \=AT,
   evaluate(R,RF,RV),
   V is RV+1.                          % αύξησε ευρετική τιμή κατά 1
```

Π1.2.2 Αλγόριθμοι Αναζήτησης

Στην ενότητα αυτή παρατίθεται η υλοποίηση των πιο αντιπροσωπευτικών αλγορίθμων αναζήτησης. Όλα τα προγράμματα υποθέτουν την ύπαρξη των **initial_state**, **goal** και

operator που εξαρτώνται από το πρόβλημα. Οι υλοποιήσεις που παρουσιάζονται δεν είναι οι πιο αποδοτικές, γιατί το κριτήριο για την υλοποίηση ήταν η ευκολία κατανόησης του κώδικα από τον αναγνώστη.

Αναζήτηση πρώτα σε βάθος (DFS)

Η αναζήτηση πρώτα σε βάθος (DFS) υλοποιείται εύκολα στην PROLOG, αφού μπορεί να εκμεταλλευτεί τον ενσωματωμένο μηχανισμό εκτέλεσής της, ο οποίος ακολουθεί αυτή τη στρατηγική. Η έναρξη εκτέλεσης του προγράμματος γίνεται με την ερώτηση **?-godfs(Solution)**, όπου στη μεταβλητή **Solution** επιστρέφεται η λύση του προβλήματος, σε μορφή λίστας. Στην υλοποίηση που ακολουθεί πραγματοποιείται έλεγχος για βρόχους μόνο στις καταστάσεις του τρέχοντος μονοπατιού αναζήτησης και όχι στο σύνολο των καταστάσεων που έχει επισκεφθεί ο αλγόριθμος. Επίσης, επειδή η λύση από τον κανόνα **dfs** επιστρέφεται με ανάστροφη σειρά, χρησιμοποιείται το ενσωματωμένο κατηγόρημα **reverse** της PROLOG, το οποίο αντιστρέφει τη σειρά των στοιχείων μιας λίστας.

```
godfs(Solution):-
   initial_state(IS),
   dfs(IS, [IS], Solution1),
   reverse(Solution1,Solution).

dfs(State, Solution, Solution):-
   goal(State).
dfs(State, PathSoFar, Solution):-
   operator(State,Child),
   not(member(Child, PathSoFar)),
   dfs(Child, [Child|PathSoFar], Solution).
```

Αναζήτηση πρώτα σε πλάτος (BFS)

Η έναρξη εκτέλεσης του προγράμματος γίνεται με την κλήση **?-gobfs(Solution)**, όπου στη μεταβλητή **Solution** επιστρέφεται η λύση του προβλήματος. Στην υλοποίηση που ακολουθεί γίνεται έλεγχος για βρόχους στις καταστάσεις που επισκέπτεται ο αλγόριθμος.

```
gobfs(Solution):-
   initial_state(IS),
   bfs([[IS]],RSolution),
   reverse(RSolution,Solution).

bfs([[State|Path]|_], [State|Path]):-
   goal(State).
bfs([[State|Path]|RestFrontierSet], Solution):-
   expand(State,Path,ChildrenStates),
```

```
    append(RestFrontierSet,ChildrenStates,NewFrontierSet),
    bfs(NewFrontierSet, Solution).

expand(State,Path,Children):-
    findall([Child,State|Path],
          (operator(State,Child),not(member(Child,Path))), Children).
```

Να σημειωθεί ότι το παραπάνω πρόγραμμα μπορεί πολύ εύκολα να τροποποιηθεί ώστε να υλοποιεί τον αλγόριθμο αναζήτησης πρώτα σε βάθος, αλλάζοντας την κλήση στην **append/3** στο δεύτερο κανόνα **bfs/2** σε:

```
    append(ChildrenStates, RestFrontierSet, NewFrontierSet)
```

Επαναληπτική εκβάθυνση (ID)

Η έναρξη εκτέλεσης του προγράμματος γίνεται με την κλήση **?-goid(Solution)**, όπου στη μεταβλητή **Solution** επιστρέφεται η λύση του προβλήματος. Ο αλγόριθμος υλοποιήθηκε με επανάληψη (κατηγόρημα **repeat**) και όχι με αναδρομή, επειδή η επανάληψη είναι πιο αποδοτική, αφού σε κάθε επανάληψη ανακτάται η μνήμη που χρησιμοποιήθηκε.

```
goid(Solution):-
    initial_state(IS),
    id(IS,Solution).

id(IS,Solution):-
    abolish(depth/1),
    assert(depth(0)),
    repeat,                          % Επανάληψη των παρακάτω
    retract(depth(PreviousD)),       % Ποιο ήταν το προηγούμενο βάθος
    NextD is PreviousD + 1,          % Νέο βάθος NextD (συνήθως +1)
    assert(depth(NextD)),               % Κράτησε το νέο βάθος
    bdfs(NextD,IS,[IS],Solution).       % Κάνε DFS μέχρι βάθος NextD
```

Στο παραπάνω πρόγραμμα το τρέχον βάθος δίνεται από το δυναμικό γεγονός **depth/1**, ενώ οι κανόνες **bdfs/4** υλοποιούν την περιορισμένη αναζήτηση πρώτα σε βάθος (bounded depth first search).

Επέκταση και οριοθέτηση (B&B)

Η έναρξη εκτέλεσης του προγράμματος γίνεται με την κλήση **?-go_bb**. Ο αλγόριθμος δεν τερματίζει μόλις βρει την πρώτη λύση, αλλά συνεχίζει για να βρει και άλλες καλύτερες. Οι λύσεις τυπώνονται στην οθόνη. Κάθε λύση που βρίσκεται είναι καλύτερη από όλες τις προηγούμενες που έχουν βρεθεί.

```
go_bb:-
```

```
    abolish(bestsofar/2),
    assert(bestsofar(_,9999)),   % αρχικοποίηση της καλύτερης λύσης
    initial_state(InitialState),
    bandb(InitialState,[],Solution,0,TotalCost),
    update(Solution,TotalCost),     % ανανέωση της καλύτερης λύσης
    write(TotalCost),nl,
    fail.                     % εξαναγκασμένη αποτυχία για την εύρεση
                              % όλων των λύσεων
go_bb:-                          % Τέλος αναζήτησης
    bestsofar(Solution,Cost),
    write(Solution),nl,
    write(Cost),nl.

bandb(State,_,[],Cost,Cost):-
    goal(State).
bandb(State,PathSoFar,[State|RestSolution],CostSoFar,TotalCost):-
    operator(State,Next),
    not(member(Next,PathSoFar)),
    evaluate_cost(State,Next,C),    % υπολογισμός επιμέρους κόστους
    NewCostSoFar is CostSoFar+C,    % υπολογισμός συνολικού κόστους
    bestsofar(_,BestCost),
    NewCostSoFar < BestCost,        % κριτήριο κλαδέματος ή συνέχειας
    bandb(Next,[Next|PathSoFar],RestSolution,NewCostSoFar,TotalCost).
update(Solution,TotalCost):-       % ανανέωση της καλύτερης λύσης
    retract(bestsofar(_,_)),
    assert(bestsofar(Solution,TotalCost)),!.
```

Να σημειωθεί τέλος ότι η υλοποίηση του κατηγορήματος **evaluate_cost/3**, το οποίο επιστρέφει το κόστος μετάβασης από μια κατάσταση σε μια γειτονική της, μπορεί να διαφέρει ανάλογα με το πρόβλημα.

Ο Αλγόριθμος αναρρίχησης λόφου (HC)

Ο αλγόριθμος καλείται με την κλήση **?-gohc(Solution)**, όπου στη μεταβλητή **Solution** επιστρέφεται η λύση του προβλήματος.

```
gohc(Solution):-
    initial_state(IS), goal(FS),
    heuristic(IS, FS, V),
    hc(IS, [V-IS], RSolution, FS),
    reverse(RSolution, Solution).
```

```
hc(FS, Solution, Solution, FS):-!.
hs(State, PathSoFar, Solution, FS):-
   next_states(State, Children, FS),  % βρες όλες τις καταστάσεις-παιδιά
   keysort(Children, [BestChild|_]),  % ταξινόμηση των παιδιών
   not(member(Children, PathSoFar)),
   hc(BestChild, [BestChild|PathSoFar], Solution, FS).

next_states(V-State, Children, FS):-
   findall(HV-Child,                  % Φτιάξε ζευγάρια αυτής της μορφής
      (operator(State, Child) ,       % Βρες επόμενη κατάσταση
       heuristic(Child, FS, HV) ),    % με την ευρετική της τιμή
      Children).                   % βάλε τα σε μία λίστα
```

*Ο αλγόριθμος A**

Η έναρξη εκτέλεσης του προγράμματος γίνεται με την κλήση **?-goastar(Solution)**, όπου στη μεταβλητή **Solution** επιστρέφεται η λύση του προβλήματος. Αφήνεται σαν άσκηση στον αναγνώστη να τροποποιηθεί ο αλγόριθμος, ώστε αυτός να λειτουργεί σαν αναζήτηση πρώτα στο καλύτερο (best-first).

```
goastar(Solution):-
   initial_state(IS),heuristic(IS,V),
   astar([V-[IS]],Solution).

astar([_-[State|Path]|_],[State|Path]):-
   goal(State).
astar([V-[State|Path]|RestPaths],Solution):-
   expand(V-[State|Path],NewPaths),
   append(NewPaths,RestPaths,Frontier),
   keysort(Frontier,OrderedFrontier), % ταξινόμηση του συνόρου αναζήτησης
   astar(OrderedFrontier,Solution).

expand(_-[State|Path],NewPaths):-
   findall( Value-[NewState,State|Path],
         (operator(State,NewState),
         not(member(NewState,Path)),  % έλεγχος βρόχων
         heuristic(NewState,HV),   % ευρετική τιμή
         length(Path,L),           % μήκος μερικής λύσης (απόσταση)
         Value is L+1+HV),         % ευρετική τιμή + απόσταση
      NewPaths).
```

Ένα παράδειγμα του κατηγορήματος **heuristic** είναι αυτό που επιστρέφει την απόσταση Manhattan στο πρόβλημα εύρεσης διαδρομής σε λαβύρινθο.

```
heuristic((X,Y), Value):-
   goal((GX,GY)),
   DX is abs(GX-X),   DY is abs(GY-Y),   Value is DX+DY.
```

Ο αλγόριθμος Minimax

Η έναρξη εκτέλεσης του προγράμματος γίνεται με την κλήση **?-minimax(<WHO>, <INITIAL STATE>, BestPosition, BestValue)**, όπου **<WHO>** είναι ένα από τα **max** ή **min** και **<INITIAL STATE>** η τρέχουσα κατάσταση του παιχνιδιού. Στη μεταβλητή **BestPosition** επιστρέφεται η καλύτερη επόμενη κατάσταση (που υποδεικνύει την καλύτερη κίνηση) ενώ στη μεταβλητή **BestValue** η τιμή αξιολόγησης που προκύπτει από την καλύτερη κίνηση.

```
minimax(Who, Pos, BestSucc, Val):-
   moves(Pos, PosList), !,      % επιτρεπτές κινήσεις
   switch(Who,TheOther),
   best(TheOther, PosList, BestSucc, Val).
minimax(Who, Pos, _, Val)  :-
   staticval(Who, Pos, Val).       % αξιολόγηση τερματικής κατάστασης

best(Who, [Pos], Pos, Val)  :-
   minimax(Who, Pos, _, Val), !.
best(Who, [Pos1 | PosList], BestPos, BestVal)  :-
   minimax(Who, Pos1, _, Val1),
   best(Who, PosList, Pos2, Val2),
   betterof(Who, Pos1, Val1, Pos2, Val2, BestPos, BestVal).

switch(max,min).
switch(min,max).
```

Να σημειωθεί τέλος ότι η υλοποίηση των κατηγορημάτων **staticval/3**, το οποίο επιστρέφει την τιμή αξιολόγησης μίας τερματικής κατάστασης και **moves/2** διαφέρει ανάλογα με το πρόβλημα. Για παράδειγμα, στο παιχνίδι NIM με 7 πούλια τα κατηγορήματα ορίζονται ως εξής:

```
   moves( 7, [6-1, 5-2, 4-3]).
   moves( 6-1, [5-1-1, 4-2-1]).
   moves( 5-2, [4-2-1, 3-2-2]).
```

```
moves( 4-3, [4-2-1, 3-3-1]).
moves( 5-1-1, [4-1-1-1, 3-2-1-1]).
moves( 4-2-1, [3-2-1-1]).
...
staticval(min, 2-1-1-1-1-1, 1).
...
```

Π1.2.3 Αναπαράσταση Γνώσης

Δύο από τις πλέον διαδεδομένες μορφές αναπαράστασης γνώσης στην ΤΝ είναι τα πλαίσια και οι κανόνες if-then. Στην ενότητα αυτή παρουσιάζεται η χρήση των δομών της PROLOG για την αναπαράσταση γνώσης με τις δύο αυτές μεθόδους, καθώς και η υλοποίηση της συλλογιστικής τους.

Πλαίσια

Στη συνέχεια παρουσιάζεται η αναπαράσταση της γνώσης χρησιμοποιώντας πλαίσια. Παράλληλα επιδεικνύεται η δυνατότητα αναζήτησης τιμών μέσα σε μια ιεραρχία πλαισίων.

Γνώση για χαρακτηριστικά αυτοκινήτων με πλαίσια

Αρχικά ορίζονται ένα γενικό πλαίσιο "αυτοκίνητο" και ειδικεύσεις του που αφορούν συγκεκριμένα αυτοκίνητα. Στο γενικό πλαίσιο ορίζονται γενικές ιδιότητες και δίνονται προκαθορισμένες τιμές σε αυτές, ενώ στα ειδικότερα πλαίσια είτε ορίζονται νέες ιδιότητες ή δίνονται διαφορετικές τιμές για τις ιδιότητες του γενικού πλαισίου.

```
frame(car).
   slot(car,ako,thing).
   slot(car,country_of_manufacture,range([britain,france,germany,italy])).
   slot(car,miles_per_gallon,range([1,100])).
   slot(car,reliability,range([low,medium,high])).
   slot(car,fuel_consumption,compute(fuel_consumption)).
frame(bmw).
   slot(bmw,isa,car).
   slot(bmw,country_of_manufacture,germany).
   slot(bmw,reliability,high).
frame(italian_car).
   slot(italian_car,isa,car).
   slot(italian_car,country_of_manufacture,italy).
   slot(italian_car,reliability,low).
frame(fiat).
   slot(fiat,isa,italian_car).
frame(ferrari).
   slot(ferrari,isa,italian_car).
```

```
   slot(ferrari,reliability,high).
frame(bmw_320).
   slot(bmw_320,isa,bmw).
   slot(bmw_320,miles_per_gallon,28).
frame(my_bmw).
   slot(my_bmw,instance_of,bmw_320).
   slot(my_bmw,miles_per_gallon,20).
   slot(my_bmw,reliablity,medium).
```

Συλλογιστική πλαισίων

Το πρόγραμμα που ακολουθεί μπορεί να απαντά σε ερωτήσεις του τύπου: "*Ποια η τιμή της ιδιότητας X του αντικειμένου Y;*". Το πρόγραμμα ψάχνει πρώτα να βρει εάν το αντικείμενο *Y* έχει την ιδιότητα *X*. Εάν ναι, επιστρέφει την τιμή της ιδιότητας. Εάν όχι, ψάχνει να βρει εάν η ιδιότητα *X* εμφανίζεται σε κάποιο από τα γονικά αντικείμενα του *Y*, αρχίζοντας από το πλησιέστερο. Μόλις βρει την ιδιότητα *X*, επιστρέφει την τιμή της. Εάν τελικά η ιδιότητα *X* δεν βρεθεί σε κανένα από τα γονικά αντικείμενα, εμφανίζονται στην οθόνη σχετικά μηνύματα λάθους.

```
getvalue(Frame,Prop,V):-
   getprop(Frame,Prop,Property),
   match(Frame,Property,V).
getprop(Frame,Prop,range(Constraints)):-      % Εύρεση πεδίου τιμών
      slot(Frame,Prop,range(Constraints)),!.
getprop(Frame,Prop,compute(Prop)):-           % Εύρεση τιμής μέσω δαίμονα
      slot(Frame,Prop,compute(Prop)),!.
getprop(Frame,Prop,V):-                    % Απευθείας εύρεση τιμής
      slot(Frame,Prop,V),!.
getprop(Frame,Prop,V):-                    % Δεσμός ISA
      slot(Frame,isa,Class),
      getprop(Class,Prop,V),!.
getprop(Frame,Prop,V):-                    % Δεσμός AKO
      slot(Frame,ako,Class),
      getprop(Class,Prop,V),!.
getprop(Frame,Prop,V):-                    % Δεσμός INSTANCE_OF
      slot(Frame,instance_of,Class),
      getprop(Class,Prop,V),!.
getprop(Frame,Prop,V):-                % Άγνωστη Ιδιότητα
      write('i do not know value of property '),
      write(Prop), write(' of '), write(Frame), nl,
      !,fail.                          % Αποτυχία εύρεσης τιμής
match(Frame,compute(Prop),V):-
```

```
   compute(Frame,Prop,V),!.
match(Frame,range(Constraints),V):-
   nonvar(V),
   test(Constraints,V).
match(_,V,V).

test([N1,N2],V):- integer(N1),integer(N2),V>=N1,V=<N2,!.
test(C,V):- member(V,C),!.
test(_,V):-write('This is not a valid value property'),nl.
```

Για να πάρουμε την τιμή μιας ιδιότητας ενός πλαισίου πρέπει να γίνει η κλήση:

```
?-getv(Obj,Prop,V)
```

όπου **Obj** και **Prop** το αντικείμενο και η ιδιότητα που μας ενδιαφέρουν (μεταβλητές εισόδου), και **V** η ζητούμενη τιμή. Για παράδειγμα:

```
?-getv(my_bmw,fuel_consumpion,V).
```

Κανόνες if-then

Στη συνέχεια παρουσιάζεται η αναπαράσταση της γνώση με κανόνες if-then. Παράλληλα επιδεικνύεται η δυνατότητα ανάστροφης συλλογιστικής.

Η PROLOG δίνει τη δυνατότητα να οριστούν κανόνες με τη μορφή:

```
<N>: if <CONDITION> then <CONCLUSION>.
```

όπου **N** είναι ο αριθμός του κανόνα, **<CONDITION>** η συνθήκη 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**) πρέπει να οριστούν ως τελεστές:

```
:-op(900,xfx,:).
:-op(870,fx,if).
:-op(880,xfx,then).
:-op(550,xfy,or).
:-op(540,xfy,and).
```

Αναπαράσταση γνώσης με κανόνες για το ζωικό βασίλειο

Με βάση τα παραπάνω κωδικοποιείται η γνώση για το ζωικό βασίλειο που παρουσιάστηκε σε σχετικό κεφάλαιο ως εξής:

```
1: if   has(hair) or  gives(milk)
   then isa(mammal).
2: if   has(feathers) or (flies and lays(eggs))
```

```
   then  isa(bird).
3: if    isa(mammal) and (eats(meat) or (has(pointed_teeth) and
         has(claws) and has(forward_pointing_eyes)))
   then  isa(carnivore).
4: if    isa(carnivore) and  has(tawny_colour) and has(dark_spots)
   then  isa(cheetah).
5: if    isa(carnivore) and has(tawny_colour) and has(black_stripes)
   then  isa(tiger).
6: if    isa(bird) and not(flies) and swims
   then  isa(penguin).
7: if    isa(bird) and isa(good_flyer)
   then  isa(albatros).
```

Ανάστροφη ακολουθία εκτέλεσης κανόνων

Το πρόγραμμα που ακολουθεί υλοποιεί την ανάστροφη εκτέλεση κανόνων για κανόνες if-then. Για να αποδειχτεί κάτι που ζητάμε πρέπει να γίνει η ερώτηση `?-backchain(<HYPOTHESIS>)`, όπου `<HYPOTHESIS>` είναι αυτό που πρέπει να αποδειχτεί. Για παράδειγμα: `?-backchain(isa(albatross)).`

Το πιο ενδιαφέρον στοιχείο αυτού του προγράμματος είναι ότι δημιουργήθηκε με τις αρχές ενός μετα-διερμηνέα (meta-interpreter). Η PROLOG είναι μία γλώσσα στην οποία η υλοποίηση ενός μετα-διερμηνέα είναι σχετικά εύκολη λόγω της δυνατότητας της να χειρίζεται σύνθετους όρους και κατηγορήματα σαν ορίσματα άλλων κατηγορημάτων καθώς επίσης και της ύπαρξης έξτρα-λογικών κατηγορημάτων.

```
backchain(X):-
   abolish(told_yes/1),abolish(told_no/1),
   assert(told_yes(nothing)),assert(told_no(nothing)),
   backsolve(X).
backsolve(G):-
   told_yes(G),!,          % αν έχει ειπωθεί τότε είναι αληθές
   write('I know that '),write(G),write(' is true'),nl.
backsolve(G):-
   told_no(G),          % αν έχει ειπωθεί το αντίθετο τότε είναι ψευδές
   write('I know that '),write(G),write(' is NOT true'),nl,
   !,fail.
backsolve(not(G)):-        % επίλυση της άρνησης
   not(backsolve(G)).
backsolve(G1 and G2):-     % επίλυση σύζευξης AND
   backsolve(G1),
   backsolve(G2).
```

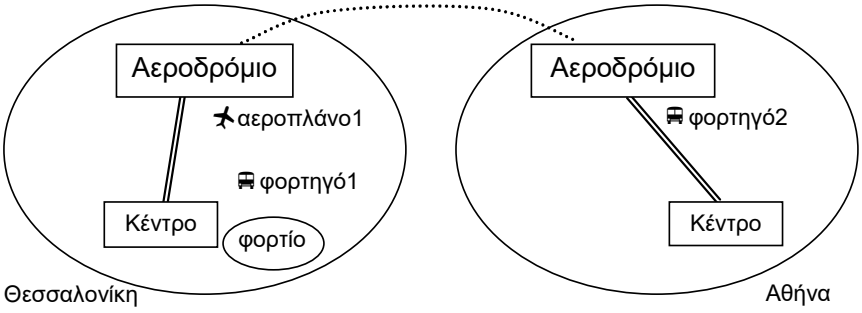
```
backsolve(G1 or _):-          % επίλυση διάζευξης OR
   backsolve(G1).             % είτε
backsolve(_ or G2):-
   backsolve(G2).             % ή
backsolve(G):-
   not(told_yes(G)),
   not(told_no(G)),
   G \= _ and _,
   G \= _ or _,
   G \= not(_),
   N:if X then G,             % αν υπάρχει κανόνας με τέτοιο συμπέρασμα
   write('I found a rule '),write(N),write(' that says: '),
   write(' if '),write(X),write(' then '), write(G),nl,
   (backsolve(X),             % είτε επίλυση της συνθήκης
   assert(told_yes(G)),
   tab(10),write('Therefore, I proved that '),
   write(G),write(' is true'),nl
        ;
   assert(told_no(G)),        % ή το αντίθετο έχει αποδειχθεί
   tab(10),write('Therefore, I proved that '),
   write(G),write(' is NOT true'),nl,
   fail).
backsolve(G):-                % αν τίποτα από όλα τα προηγούμενα δεν συμβαίνει
   not(told_yes(G)),
   not(told_no(G)),
   G \= _ and _,
   G \= _ or _,
   G \= not(_),
   not(_:if _ then G),
   ask(G).                    % ρώτα το χρήστη μήπως γνωρίζει
```

Το παραπάνω πρόγραμμα επιδεικνύει διαδραστικότητα με το χρήστη, υπό την έννοια ότι του εξηγεί τη συλλογιστική και τον ρωτά για το αν ισχύουν κάποια γεγονότα για τα οποία δεν μπορεί να συμπεράνει κάτι. Με τον ίδιο περίπου τρόπο μπορεί κάποιος να υλοποιήσει έμπειρα συστήματα βασισμένα σε κανόνες.

Π1.2.4 Σχεδιασμός Ενεργειών

Έστω το πρόβλημα μεταφοράς φορτίων (transportation logistics) που φαίνεται στο Σχήμα Π1.3. Συγκεκριμένα, έστω δύο πόλεις, η Αθήνα και η Θεσσαλονίκη. Κάθε πόλη έχει δύο τοποθεσίες, το αεροδρόμιο και το κέντρο της. Κάθε πόλη διαθέτει ένα φορτηγό, το οποίο μπορεί να μετακινείται μεταξύ των δύο τοποθεσιών της πόλης, αλλά όχι

από τη μία πόλη στην άλλη. Επιπλέον, υπάρχει ένα αεροπλάνο που μπορεί να μετακινείται μεταξύ των δύο αεροδρομίων. Τέλος, υπάρχει ένα φορτίο, το οποίο αρχικά βρίσκεται στο κέντρο της Θεσσαλονίκης. Το φορτίο μπορεί να φορτωθεί/ξεφορτωθεί στα φορτηγά των δύο πόλεων καθώς και στο αεροπλάνο.



Σχήμα Π1.3: Ένα πρόβλημα μεταφοράς φορτίων.

Αρχικά το φορτίο δεν είναι φορτωμένο πουθενά και βρίσκεται στο κέντρο της Θεσσαλονίκης. Έστω επίσης ότι αρχικά το φορτηγό της Θεσσαλονίκης βρίσκεται στο κέντρο της Θεσσαλονίκης, το αεροπλάνο βρίσκεται στο αεροδρόμιο της Θεσσαλονίκης και το φορτηγό της Αθήνας βρίσκεται στο αεροδρόμιο της Αθήνας, όπως φαίνεται στο Σχήμα Π1.3. Το ζητούμενο είναι να μεταφερθεί το φορτίο στο κέντρο της Αθήνας.

Το σύστημα σχεδιασμού ενεργειών που ακολουθεί δέχεται περιγραφές προβλημάτων κατά STRIPS και χρησιμοποιεί μια απλή αναζήτηση κατά πλάτος για την επίλυσή τους. Για τη λειτουργία του απαιτεί τον ορισμό των στατικών γεγονότων του προβλήματος σαν απλά γεγονότα PROLOG, ενώ τα δυναμικά γεγονότα της αρχικής κατάστασης πρέπει να περιλαμβάνονται σε ένα γεγονός της μορφής:

```
initial( [fact1, fact2, …]).
```

όπου οι όροι με πλάγια γράμματα πρέπει να αντικατασταθούν από τα δυναμικά γεγονότα που αληθεύουν στην αρχική κατάσταση. Παρόμοια, οι στόχοι του προβλήματος πρέπει να περιλαμβάνονται σε ένα γεγονός της μορφής:

```
goal([fact1, fact2, …]).
```

Τέλος τα σχήματα των ενεργειών πρέπει να οριστούν σαν κανόνες της μορφής:

```
operator(name(V1, V2, …)        % το όνομα και οι παράμετροι της ενέργειας
   [prec1, prec2, …],           % η λίστα προϋποθέσεων
   [del1, del2, …],           % η λίστα διαγραφής
   [add1, add2, …]):-           % η λίστα προσθήκης
   fact1, fact2, …. .         % Στατικές προϋποθέσεις εφαρμογής.
```

Ο διαχωρισμός των στατικών από τις δυναμικές προϋποθέσεις εφαρμογής μιας ενέργειας γίνεται για λόγους απόδοσης.

Κωδικοποίηση του προβλήματος σχεδιασμού

Τα στατικά γεγονότα του προβλήματος είναι αυτά που δηλώνουν την ύπαρξη των αντικειμένων του προβλήματος καθώς και τις μεταξύ τους σταθερές σχέσεις. Έτσι έχουμε:

```
city(thessaloniki).          % Ορισμός των πόλεων
...
location(thessaloniki_center).  % Ορισμός των τοποθεσιών
...
at_city(thessaloniki_center, thessaloniki). % Καθορισμός τοποθεσίας
...
airport(thessaloniki_airport).  % Χαρακτηρισμός τοποθεσιών ως αεροδρόμια
...
truck(truck1).               % Ορισμός των μέσων μεταφοράς
...
airplane(plane1).
package(package1).           % Ορισμός του φορτίου
```

Στη συνέχεια, ορίζεται η αρχική κατάσταση από τα δυναμικά γεγονότα που ισχύουν αρχικά. Έτσι, δηλώνονται οι αρχικές θέσεις των μέσων μεταφοράς και του φορτίου.

```
initial([at(truck1,    thessaloniki_center),    at(truck2,    athens_airport),
   at(plane1, thessaloniki_airport), at(package1, thessaloniki_center) ]).
```

Στην τελική κατάσταση δηλώνεται μόνο η τελική θέση του φορτίου:

```
goal( [ at(package1, athens_center) ]).
```

ενώ επιπλέον ορίζονται έξι σχήματα ενεργειών, τα οποία αφορούν:

- φόρτωση ενός φορτίου σε ένα φορτηγό
- εκφόρτωση ενός φορτίου από ένα φορτηγό
- φόρτωση ενός φορτίου σε ένα αεροπλάνο
- εκφόρτωση ενός φορτίου από ένα αεροπλάνο
- μετακίνηση ενός φορτηγού
- μετακίνηση ενός αεροπλάνου

Παραθέτουμε μόνο παραδείγματα αυτών των ενεργειών.

```
% εκφόρτωση ενός φορτίου από ένα φορτηγό
operator(unload_truck(Package, Truck, Location),
           [ at(Truck, Location), in(Package, Truck)],
           [ in(Package, Truck)],
           [ at(Package, Location)]):-
```

```
   package(Package), truck(Truck), location(Location).

% μετακίνηση ενός φορτηγού
operator(move_truck(Truck, Location1, Location2),
            [at(Truck, Location1)],
            [at(Truck, Location1)],
            [at(Truck, Location2)]):-
   truck(Truck),
   location(Location1),
   location(Location2),
   at_city(Location1, City),
   at_city(Location2, City),
   Location1\=Location2.
```

Ακολουθεί το πρόγραμμα του σχεδιαστή ενεργειών. Θα πρέπει να σημειωθεί ότι η υλοποίηση αυτή δεν είναι η πιο αποδοτική, αφού ο αλγόριθμος αναζήτησης κατά πλάτος που υιοθετείται είναι τυφλός και έχει μεγάλες απαιτήσεις σε μνήμη και σε χρόνο. Για το συγκεκριμένο πρόβλημα η εύρεση λύσης σε έναν προσωπικό υπολογιστή μεσαίων δυνατοτήτων μπορεί να απαιτήσει λίγα λεπτά, ενώ το παραγόμενο πλάνο αποτελείται από 9 ενέργειες. Ωστόσο υπάρχουν αποδοτικότερες υλοποιήσεις συστημάτων σχεδιασμού (για παράδειγμα συστήματα σχεδιασμού βασισμένα σε γράφους, σχεδιαστές ικανοποίησης προτάσεων, ευρετικοί σχεδιαστές), που επιλύουν προβλήματα σαν το παραπάνω σε ελάχιστο χρόνο. Δεν επιλέχθηκε ωστόσο η παρουσίαση ενός τέτοιου συστήματος λόγω της μεγαλύτερης πολυπλοκότητάς τους η οποία θα κάνει δύσκολη την κατανόησή του.

```
/* Εκκίνηση του συστήματος σχεδιασμού */
go:-garbage_collect,
   retract_all(n(_,_,_)),
   retract_all( agenda(_) ),
   retract_all( last_id(_) ),!,
   assert(last_id(0)),
   initial(Initial),
   % Εύρεση των ενεργειών που είναι εφαρμόσιμες στην αρχική κατάσταση
   findall(OP, applicable_operator(OP, Initial), OPs),
   % Εισαγωγή των νέων καταστάσεων στο μέτωπο αναζήτησης (agenda)
   add_nodes( 0, OPs, New_agenda),
   assert(n(0, nil, nil)),
   assert( agenda( New_agenda ) ),
   repeat,
   expand_states,
```

```
   agenda([]).

% Εξαγωγή της πρώτης κατάστασης από το μέτωπο αναζήτησης
expand_states:-retract( agenda( [ n( ID) | Agenda ])),
   assert(agenda(Agenda)),
   expand2_state(ID ), !.
% Εάν το μέτωπο αναζήτησης είναι κενό, η εκτέλεση τερματίζει
expand_states:- agenda([]).

% Επέκταση της πρώτης κατάστασης του μετώπου αναζήτησης
expand2_state(ID ):-
   n(ID, Operator, Previous_ID),!,
   find_plan(Previous_ID, Plan),
   append(Plan, [Operator], New_plan),
   initial(Initial),
   find_state(Initial, New_plan, New_state),
   not_repeated_state(Initial, Plan, New_state),  % Έλεγχος για βρόχο
   % Έλεγχος για τελική κατάσταση
   is_goal_state(New_state, New_plan ),
   % Εύρεση των εφαρμόσιμων ενεργειών
   findall(OP1, applicable_operator(OP1, New_state), OPs),
   add_nodes(ID, OPs, Apnd_agenda),
   retract(agenda(Old_agenda)),
   append(Old_agenda,Apnd_agenda, New_agenda),
   assert(agenda( New_agenda )),!.

% Προσθήκη νέων καταστάσεων στο μέτωπο αναζήτησης
add_nodes( _, [], [] ) :- ! .
add_nodes( Previous_ID, [OP|OPs ], [n(New_ID)|Append_Ordered ]):-
   retract( last_id( Last_ID )),!,
   New_ID is Last_ID + 1 ,
   assert( last_id( New_ID )),
   assert( n( New_ID, OP, Previous_ID ) ),!,
   add_nodes( Previous_ID, OPs, Append_Ordered ).

% Εύρεση του πλάνου μέχρι την τρέχουσα κατάσταση
find_plan(0, []):-!.
find_plan(ID, Plan):-
   n(ID, OP, Previous_ID),!,
   find_plan(Previous_ID, H_Plan),
   append(H_Plan, [OP], Plan).
```

```
find_state(New_state, [], New_state):- ! .
find_state(State, [OP|New_plan], New_state):-
   newstate( State, OP, State1),!,
   find_state( State1, New_plan, New_state ).

% Δημιουργία νέας κατάστασης μετά την εφαρμογή ενέργειας
newstate(State,Operator,NewState):-
   operator(Operator, _, Delete_List, Add_List),
   delete_all(Delete_List,State,State1),
   add_all(Add_List,State1,NewState).

% Εύρεση τελεστή εφαρμόσιμου σε κατάσταση
applicable_operator(OP1,State):-
   operator( OP1, Prec, _Del, _Add),
   compatible(Prec,State).
```

Π1.2.5 Συστήματα Πρακτόρων

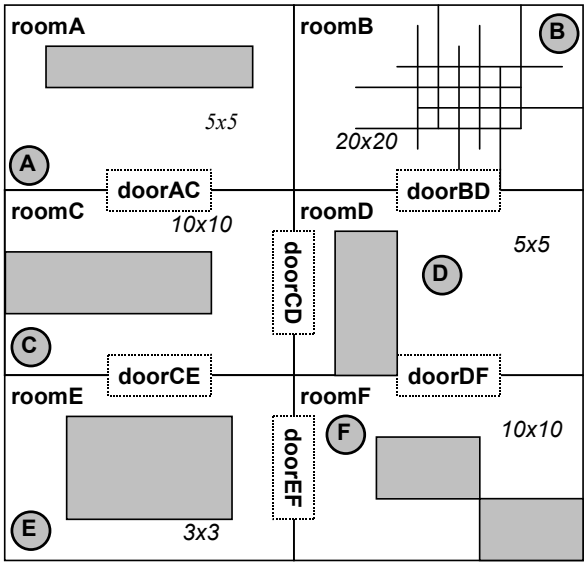
Στην ενότητα αυτή παρουσιάζονται εφαρμογές της ΤΝ οι οποίες αφορούν πολυπρακτορικά συστήματα (multi-agent systems). Ένα πρόβλημα που μπορεί να αντιμετωπισθεί από πολλούς πράκτορες είναι η μετακίνηση φορτίων σε διάφορα δωμάτια, όπου σε κάθε δωμάτιο είναι υπεύθυνος ένας πράκτορας. Συγκεκριμένα, έστω ο κόσμος που απεικονίζεται στο Σχήμα Π1.4, όπου:

- υπάρχουν έξι δωμάτια,
- τα δωμάτια συνδέονται με πόρτες,
- υπάρχουν διάφορα εμπόδια μέσα στα δωμάτια,
- υπάρχει ένας πράκτορας μέσα σε κάθε δωμάτιο,
- ο πράκτορας κάθε δωματίου μπορεί να μετακινείται μόνο μέσα στο δωμάτιό του,
- κάθε πράκτορας χρησιμοποιεί διαφορετικό αλγόριθμο αναζήτησης για την επίλυση των υπο-προβλημάτων που τον αφορούν.

Το ζητούμενο είναι να μεταφερθεί ένα αντικείμενο από μία συγκεκριμένη αρχική θέση ενός δωματίου σε μία συγκεκριμένη τελική θέση ενός άλλου δωματίου. Για παράδειγμα, εάν το αντικείμενο είναι στο δωμάτιο F και πρέπει να μεταφερθεί στο δωμάτιο A, πρέπει να γίνουν οι παρακάτω ενέργειες:

- Ο πράκτορας F πρέπει να πάει δίπλα στο αντικείμενο.
- Ο πράκτορας F πρέπει να μετακινήσει το αντικείμενο δίπλα στην πόρτα που συνδέει το δωμάτιο F με το δωμάτιο E.
- Ο πράκτορας E πρέπει να πάει δίπλα στην πόρτα που συνδέει το δωμάτιο E με το δωμάτιο F.

- Ο πράκτορας E πρέπει να μετακινήσει το αντικείμενο δίπλα στην πόρτα που συνδέει το δωμάτιο E με το δωμάτιο C, κτλ.



Σχήμα Π1.4: Ο κόσμος του προβλήματος.

Η διαδικασία επίλυσης του προβλήματος είναι η εξής: Με δεδομένη μια αρχική και μια τελική θέση ενός αντικειμένου, ένας εξωτερικός πράκτορας, ο επιβλέπων (supervisor), αναλύει το αρχικό πρόβλημα σε μια σειρά από υπο-προβλήματα, τα οποία πρέπει να λυθούν από τους πράκτορες των δωματίων ανεξάρτητα. Για το σκοπό αυτό ο επιβλέπων πράκτορας εφαρμόζει κάποιον αλγόριθμο αναζήτησης στην τοπολογία των δωματίων. Στη συνέχεια, οι πράκτορες των δωματίων προσπαθούν να βρουν τη διαδρομή που πρέπει να ακολουθήσουν μέσα στο δωμάτιό τους, καθώς και τις υπόλοιπες ενέργειες που πρέπει να εκτελέσουν (μεταφορά του αντικειμένου), ώστε να επιτύχουν το στόχο που τους έχει ανατεθεί. Αυτό γίνεται εφαρμόζοντας αλγορίθμους αναζήτησης στην τοπολογία του δωματίου τους. Για το σκοπό αυτό το δωμάτιο αναλύεται σε ένα πλέγμα θέσεων (grid), κάποιες από τις οποίες είναι ελεύθερες και κάποιες άλλες έχουν εμπόδια. Το πλέγμα θέσεων μπορεί να διαφέρει στα διάφορα δωμάτια ως προς τη λεπτομέρειά του, κάτι που φυσικά εξαρτάται από το δωμάτιο.

Η εκκίνηση του προγράμματος γίνεται με μία κλήση της μορφής:

```
?- solve(( Room1, X1, Y1), (Room2, X2, Y2)).
```

όπου **Room1** το δωμάτιο στο οποίο βρίσκεται αρχικά το αντικείμενο και **X1**, **Y1** οι συντεταγμένες της θέσης του μέσα στο δωμάτιο αυτό, ενώ **Room2**, **X2** και **Y2** οι αντίστοιχες πληροφορίες για την τελική του θέση. Για παράδειγμα:

```
?- solve( (roomA, 5, 5), (roomF, 10,5)).
```

Περιγραφή του κόσμου του προβλήματος σε σύστημα πρακτόρων

Στη συνέχεια, παρουσιάζεται η περιγραφή του κόσμου του προβλήματος. Για την επίλυση του προβλήματος μπορεί να χρησιμοποιηθεί οποιοσδήποτε από τους αλγορίθμους αναζήτησης που παρατέθηκαν στα προηγούμενα. Ωστόσο, κατά την κλήση του,

ο αλγόριθμος θα πρέπει να δέχεται τέσσερα ορίσματα, τα οποία κατά σειρά πρέπει να είναι το όνομα του πράκτορα, η αρχική κατάσταση, η τελική κατάσταση και η λύση, η οποία και θα επιστρέφεται.

```
% Σύνδεση (πόρτες) μεταξύ δωματίων
connects(roomA,roomC,doorAC).
connects(roomC,roomE,doorCE).
...
% Υλοποίηση της αμφίδρομης σύνδεσης μεταξύ δύο δωματίων
next(R1,R2,D):-connects(R1,R2,D).
next(R1,R2,D):-connects(R2,R1,D).

% Κάθε πόρτα βρίσκεται σε συγκεκριμένες συντεταγμένες μέσα σε ένα δωμάτιο
door_at(roomA,doorAC,(3,1)).
door_at(roomC,doorAC,(5,10)).
door_at(roomC,doorCE,(5,1)).
...
% Κάθε δωμάτιο χωρίζεται σε ένα πλέγμα θέσεων ίσου μεγέθους
grid(roomA,5,5).
grid(roomB,20,20).
...
% Κάθε πράκτορας βρίσκεται αρχικά σε συγκεκριμένη θέση μέσα σε ένα δωμάτιο
agent_at(agentA,(roomA,5,3)).
agent_at(agentB,(roomB,20,20)).
...
% Κάθε πράκρορας μπορεί να μετακινείται είτε
% οριζόντια/κατακόρυφα ή διαγώνια
move_ability(agentA,diag).
move_ability(agentB,hv).
...
% Κάθε πράκτορας γνωρίζει έναν διαφορετικό αλγόριθμο αναζήτησης
search_ability(agentA,dfs).
search_ability(agentB,id).
search_ability(agentC,astar).
search_ability(agentD,bfs).
search_ability(agentE,bestf).
search_ability(agentF,id).

% Τα εμπόδια σε ένα δωμάτιο
obstacle(roomA,(X,Y)):- X>=2, X=<4, Y>=3, Y=<4.
obstacle(roomC,(X,Y)):- X>=1, X=<7, Y>=3, Y=<6.
```

```
...
% Ικανότητα του κάθε πράκτορα
operator(supervisor,Room1,Room2):-!,
   next(Room1,Room2,_).
operator(Agent,(Room,X,Y),(Room,X1,Y1)):-
   move_ability(Agent,hv),
   movehv(Agent,(X,Y),(X1,Y1)),
   grid(Room,MaxX,MaxY),
   X1 > 0, X1 =< MaxX,
   Y1 > 0, Y1 =< MaxY,
   not(obstacle(Room,(X1,Y1))).
operator(Agent,(Room,X,Y),(Room,X1,Y1)):-
   move_ability(Agent,diag),
   movediag(Agent,(X,Y),(X1,Y1)),
   grid(Room,MaxX,MaxY),
   X1 > 0, X1 =< MaxX,
   Y1 > 0, Y1 =< MaxY,
   not(obstacle(Room,(X1,Y1))).

% Απόσταση Manhatan μεταξύ δύο θέσεων
heuristic((_,X,Y),(_,FX,FY),M):-
   M is abs(FX-X)+abs(FY-Y).

% Η βασική κλήση
solve((R1,X1,Y1),(R2,X2,Y2)):-
   abstract_path((R1,X1,Y1),(R2,X2,Y2),AllGoals),
   find_solution(AllGoals).

% Εύρεση όλων των στόχων που θα ανατεθούν στους πράκτορες
abstract_path((R1,X1,Y1),(R2,X2,Y2),AllGoals):-
   gobfs(supervisor,R1,R2,Path),
   analyse(Path,NewPath),
   append([(R1,X1,Y1)|NewPath],[(R2,X2,Y2)],AllGoals),!.

% Μετατροπή ενός μονοπατιού δωματίων σε ένα μονοπάτι από πόρτες
analyse([_],[]).
analyse([R1,R2|T],[(R1,X1,Y1),(R2,X2,Y2)|R]):-
   next(R1,R2,Door),
   door_at(R1,Door,(X1,Y1)),
   door_at(R2,Door,(X2,Y2)),
   analyse([R2|T],R).
```

```
% Δύο στόχοι που πρέπει να επιτευχθούν από έναν πράκτορα σε ένα δωμάτιο
do((R,X1,Y1),(R,X2,Y2)):-
   agent_at(Agent,(R,X,Y)),
   search_ability(Agent,Search),
   write(Agent), write(' Applying '), write(Search), nl,
   name(go,L1), name(Search,L2), append(L1,L2,L), name(PredN,L),
   Call1=..[PredN,Agent,(R,X,Y),(R,X1,Y1),S1],
   Call1,
   write(Agent),write(':'),write(S1),nl,
   Call2=..[PredN,Agent,(R,X1,Y1),(R,X2,Y2),S2],
   Call2,
   write(Agent),write(':'),write(S2),nl, !.
```

Αντιδραστικοί πράκτορες

Ένα διαστημόπλοιο προσγειώνεται σε έναν πλανήτη του ηλιακού μας συστήματος (για παράδειγμα στον Άρη). Το σκάφος διαθέτει τέσσερα ρομπότ-πράκτορες, που σκοπό έχουν να μαζέψουν δείγματα εδάφους και να τα μεταφέρουν στο διαστημόπλοιο. Το έδαφος είναι γεμάτο εμπόδια, τα οποία πρέπει να αποφεύγουν τα ρομπότ. Τα ρομπότ χρησιμοποιούν ένα σύνολο από απλούς αντιδραστικούς κανόνες για να αποφεύγουν εμπόδια και για να συλλέγουν δείγματα. Επίσης, δε γνωρίζουν τίποτα για το έδαφος, ενώ μπορούν να αναγνωρίζουν εμπόδια και δείγματα μόνο όταν τα συναντήσουν κατά την τυχαία κίνησή τους. Για την επιστροφή των ρομπότ στο διαστημόπλοιο, το τελευταίο εκπέμπει συνεχώς ένα σήμα που ανιχνεύεται από τα ρομπότ. Αναλυτική περιγραφή των κανόνων (συμπεριφορών) των πρακτόρων υπάρχει σε σχετικό κεφάλαιο.

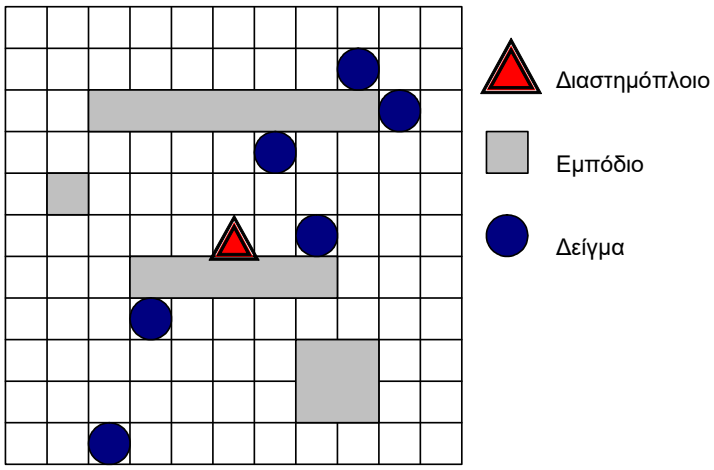
Για την αντιμετώπιση του παραπάνω προβλήματος σε PROLOG γίνονται οι εξής παραδοχές:

- Το έδαφος διαχωρίζεται σε θέσεις, έτσι ώστε να δημιουργηθεί ένα πλέγμα θέσεων. Η μορφή του εδάφους φαίνεται στο Σχήμα Π1.5.
- Η θέση του διαστημοπλοίου έχει συντεταγμένες (0, 0).
- Όλα τα ρομπότ βρίσκονται αρχικά επίσης στη θέση (0, 0).
- Τα ρομπότ μπορούν να μετακινούνται μόνο σε τέσσερις διευθύνσεις: δυτικά, ανατολικά, βόρεια και νότια.

Οι κανόνες που ακολουθούν τα ρομπότ, σε φθίνουσα σειρά προτεραιότητας, είναι οι εξής:

- Εάν συναντήσεις εμπόδιο, άλλαξε κατεύθυνση.
- Εάν έχεις συγκεντρώσει δείγματα και βρίσκεσαι στη βάση, άφησε τα δείγματα.
- Εάν έχεις συγκεντρώσει δείγματα και δεν βρίσκεσαι στη βάση, τότε προχώρα προς τη βάση.
- Εάν συναντήσεις κάποιο δείγμα και δεν βρίσκεσαι στη βάση, μάζεψε το δείγμα.

- Προχώρα στην τύχη.



Σχήμα Π1.5: Η μορφολογία της επιφάνειας του πλανήτη.

Η εκκίνηση της εκτέλεσης του προγράμματος γίνεται με την κλήση:

```
?- run.
```

Περιγραφή του κόσμου του προβλήματος σε αντιδραστικούς πράκτορες

Οι κανόνες μπορεί να γραφούν στη μορφή if-then ορίζοντας τους κατάλληλους τελεστές, όπως έγινε και στην αναπαράσταση γνώσης.

```
% Κανόνες κατά φθίνουσα σειρά προτεραιότητας
% A είναι το όνομα του πράκτορα
1-A : if detect_obstacle(A)
      then change_direction(A).
2-A : if carrying_samples(A) and at_the_base(A)
      then drop_samples(A).
3-A : if carrying_samples(A) and not(at_the_base(A))
      then travel_up_gradient(A).
4-A : if detect_samples(A) and not(at_the_base(A))
      then pick_up_samples(A).
5-A : if true
      then move_randomly(A).

% Ο βασικός βρόχος ελέγχου του πράκτορα. Εκτελείται συνέχεια!!
% Στη λίστα της member φαίνονται τα ονόματα των πρακτόρων.
run:-
   repeat,
   member(A,[mike,jeff,ralf,jack]),
   one_cycle(A),
```

```
   fail.

% Βρίσκει τους κανόνες που πυροδοτούνται για έναν πράκτορα
% και επιλέγει τον πιο σημαντικό.
one_cycle(A):-
   findall(Action,
         (_-A:if Condition then Action,
         solvec(Condition)),
      [FirstAction|_]),
   FirstAction,!.

% Μετα-διερμηνευτής που ελέγχει εάν ικανοποιείται η συνθήκη ενός κανόνα.
solvec(P1 and P2):-!,
   solvec(P1),
   solvec(P2).
solvec(not(P)):-!,
   not(P).
solvec(P):-
   P.
```

Υλοποίηση του πρωτοκόλλου KQML

Για να μπορεί ένας πράκτορας να επικοινωνεί με μηνύματα KQML, χρειάζεται:

- Να ελέγχει εάν τα εισερχόμενα μηνύματα είναι συντακτικώς σωστά.
- Να καταλαβαίνει το περιεχόμενο των εισερχόμενων μηνυμάτων και να προβαίνει στις απαραίτητες ενέργειες.
- Να συντάσσει και να αποστέλλει νέα μηνύματα, εάν κάτι τέτοιο απαιτείται.

Έστω μια περιορισμένη έκδοση του πρωτοκόλλου KQML, όπου τα μηνύματα έχουν την ακόλουθη μορφή:

```
(<performative>
   :sender <word>
   :receiver <word>
   :in-reply-to <word>
   :reply-with <word>
   :language <word>
   :content <expression>)
```

Ενέργεια (performative)	S: Αποστολέας (Sender), R: Παραλήπτης (Receiver)	Απαιτείται απάντηση
ask-if	Ο S θέλει να ξέρει εάν το :content υπάρχει στη βάση γνώσης του R	Ναι
ask-one	Ο S θέλει να ξέρει μια περίπτωση του :content που υπάρχει στη βάση γνώσης του R	Ναι
ask-all	Ο S θέλει να ξέρει όλες τις περίπτωσεις του :content που υπάρχουν στη βάση γνώσης του R	Ναι
tell	Η πρόταση στο :content υπάρχει στη βάση γνώσης του S	Όχι
advertise	Ο S θέλει να ξέρει ότι ο R μπορεί και θέλει να αναλάβει την εργασία που περιγράφεται στο μήνυμα :content	Ναι
sorry	Ο S καταλαβε το μήνυμα του R αλλά δεν μπορεί να απαντήσει	Όχι

Μερικές παρατηρήσεις για τα μηνύματα:

- Η ενέργεια **sorry** δεν έχει τα πεδία **:language** και **:content**.
- Η σειρά των πεδίων στα μηνύματα είναι αυστηρή.
- **<word>** είναι οποιαδήποτε ακολουθία χαρακτήρων, εκτός κενού.
- **<expression>** είναι οποιοδήποτε αλφαριθμητικό περικλείεται σε διπλά εισαγωγικά ή ένα άλλο KQML μήνυμα.

Ο πράκτορας που περιγράφεται παρακάτω ονομάζεται **petros**, γνωρίζει μόνο τις γλώσσες KQML και PROLOG, διαβάζει μηνύματα από αρχεία και αποκρίνεται εμφανίζοντας μηνύματα στην οθόνη. Ωστόσο, θα μπορούσε εύκολα να επεκταθεί ώστε η είσοδος και η έξοδος των μηνυμάτων να είναι διαφορετική (για παράδειγμα από και προς το διαδίκτυο).

Η εκτέλεση του προγράμματος γίνεται με την κλήση:

```
?- incoming(Filename).
```

όπου **Filename** το όνομα ενός αρχείου κειμένου τύπου ASCII, όπου περιέχεται ένα μήνυμα όπως για παράδειγμα το ακόλουθο:

```
(ask-if
   :sender john
   :receiver petros
   :in-reply-to id1
   :reply-with id2
   :language prolog
   :content "length([a,b,c],L)" ).
```

Ο κώδικας του πράκτορα που αντιλαμβάνεται KQML

```
% Εισερχόμενο μήνυμα
```

```
incoming(FileName):-
   see(FileName),      % διαβάζει ένα μήνυμα από ένα αρχείο
   reads(L),
   seen,!,
   parse(Message,L),!, % ελέγχει εάν είναι συντακτικά σωστό
   respond(Message).      % παράγει την απάντηση

% Παράδειγμα πράκτορα
message_id(petros-1).

get_message_id(NID):-
   retract(message_id(_-N)),
   N1 is N+1,
   get_agent_id(A),
   NID = A-N1,
   assert(message_id(A-N1)),!.

get_agent_id(petros).

speak(prolog).
speak(kqml).
```

Ο κώδικας του συντακτικού αναλυτή της KQML

Παρατίθεται ένας περιορισμένος συντακτικός αναλυτής (parser) KQML, βασισμένος σε DCG (Definite Clause Grammars). Οι ενέργειες που αναγνωρίζονται είναι οι: **ask-if**, **ask-one**, **ask-all**, **tell**, **advertise**, **sorry**. Εάν η έκφραση KQML είναι συντακτικώς σωστή, τότε ο αναλυτής παράγει έναν όρο της PROLOG στην ακόλουθη μορφή:

```
message(performative(P), sender(S), receiver(R), replyto(RT),
      replywith(RW), language(L), content(C)  )
```

Ο συντακτικός αναλυτής KQML είναι ο ακόλουθος:

```
parse(Term,L):-
   kqml_message(Term,L,[]),
   nl,write('The Message is parsed:'),nl,!,
   write(Term),nl.
parse(error,_):-
   write('Message has a syntax error'),nl,!.
kqml_message(Term) -->
   ['('],
   performative_with_content(P),
```

```
   parameters_with_content(L),
   [')'],              {Term=..[message,P|L]}.
kqml_message(Term) -->
   ['('],
   performative_without_content(P),
   parameters_without_content(L),
   [')'],              {Term=..[message,P|L]}.
performative_with_content(performative(P))-->
   discourse_performative(P).
performative_without_content(performative(P))-->
   intervention_performative(P).
% Αυτές είναι οι ενέργειες που υποστηρίζονται από τον αναλυτή του πράκτορα
discourse_performative('ask-if') --> ['ask-if'].
discourse_performative('ask-all') --> ['ask-all'].
discourse_performative('ask-one') --> ['ask-one'].
discourse_performative('tell') --> ['tell'].
discourse_performative('advertise') --> ['advertise'].
intervention_performative('sorry') --> ['sorry'].
% Οι παράμετροι που αναγνωρίζονται από τον αναλυτή του πράκτορα
parameters_with_content([S,R,RT,RW,L,C]) -->
   [':sender'], word(WS),    {S=sender(WS)},
   [':receiver'], word(WR),     {R=receiver(WR)},
   [':in-reply-to'], word(WRT), {RT=replyto(WRT)},
   [':reply-with'], word(WRW),  {RW=replywith(WRW)},
   [':language'], word(WL),     {L=language(WL)},
   [':content'], content(WC),{C=content(WC)}.
parameters_without_content([S,R,RT,RW,language(empty),content(empty)]) -->
   [':sender'], word(WS),    {S=sender(WS)},
   [':receiver'], word(WR),     {R=receiver(WR)},
   [':in-reply-to'], word(WRT), {RT=replyto(WRT)},
   [':reply-with'], word(WRW),  {RW=replywith(WRW)}.
% Το πεδίο content μπορεί να είναι οποιοδήποτε αλφαριθμητικό
% ή ένα άλλο μήνυμα KQML
content(S) --> string(S).
content(S) --> kqml_message(S).
```

Ο κώδικας που παράγει απαντήσεις KQML

Στη συνέχεια, πρέπει να βρεθεί η ενέργεια (performative) και αν είναι κάποια που απαιτεί μια ενέργεια, πρέπει να παραχθεί το μήνυμα της απάντησης. Παρατίθεται μόνο μέρος του κώδικα:

```
respond(error):-!.
respond(Message):-
   find_performative(Message,P),
   action_required(P),!,
   do_requested_action(Message,ReplyMessage),
   writemessage(ReplyMessage).

do_requested_action(Message,ReplyMessage):-
   find_language(Message,L),
   not(speak(L)),!,
   compose(sorry,Message,ReplyMessage).
do_requested_action(Message,ReplyMessage):-
   find_performative(Message,P),!,
   compose(P,Message,ReplyMessage).

% Μηνύματα όπου η γλώσσα δεν αναγνωρίζεται απαντώνται με "sorry"
compose(sorry,Message,NewMessage):-!,
   find_sender(Message,S),
   find_replywith(Message,RW),
   get_agent_id(A),
   get_message_id(ID),
   NewMessage=message( performative(sorry),
                       sender(A),
                       receiver(S),
                       replyto(RW),
                       replywith(ID),
                       language(empty),
                       content(empty)    ).

% Μηνύματα 'ask-if' πρέπει να ελέγχουν την ύπαρξη του
% περιεχομένου (content) στη βάση γνώσης και στη συνέχεια να απαντιόνται
% με "true" ή "fail".
compose('ask-if',Message,NewMessage):-
   find_sender(Message,S),
   find_replywith(Message,RW),
   find_content(Message,C),
   find_language(Message,L),
   get_agent_id(A),
   get_message_id(ID),
   string_to_term(C,Call),
```

```
    verify(Call,Value),
    NewMessage=message( performative(tell),
                        sender(A),
                        receiver(S),
                        replyto(RW),
                        replywith(ID),
                        language(L),
                        content(Value)    ).

% Μηνύματα για τα οποία απαιτείται απόκριση
action_required('ask-if').
action_required('ask-all').
action_required('ask-one').
action_required('advertise').

% Μηνύματα για τα οποία δεν απαιτείται απόκριση
response_only('tell').
no_response('sorry').
```

Για να λειτουργήσει ο παραπάνω κώδικας χρειάζεται να εφοδιαστεί με μηνύματα. Για παράδειγμα, τα παρακάτω μηνύματα μπορούν να γραφούν σε ξεχωριστά αρχεία ώστε ο πράκτορας να τα απαντήσει ανάλογα με την ενέργεια που απαιτείται και τις γνώσεις που έχει.

```
(ask-one                          % μήνυμα-1
   :sender maria
   :receiver petros
   :in-reply-to id1
   :reply-with id2
   :language lisp
   :content "(LENGTH (a b c))").

(ask-if                         % μήνυμα-2
   :sender john
   :receiver petros
   :in-reply-to id1
   :reply-with id2
   :language prolog
   :content "length([a,b,c],L)" ).

(ask-one                          % μήνυμα-3
   :sender robert
```

```
   :receiver petros
   :in-reply-to id1
   :reply-with id2
   :language prolog
   :content "length([a,b,c],L)" ).

(ask-all                        % μήνυμα-4
   :sender christine
   :receiver petros
   :in-reply-to id1
   :reply-with id2
   :language prolog
   :content "member(X,[a,b,c])" ).

(advertise                    % μήνυμα-5
   :sender steffen
   :receiver petros
   :in-reply-to id1
   :reply-with id2
   :language kqml
   :content (  ask-if
               :sender petros
               :receiver steffen
               :in-reply-to id2
               :reply-with id3
               :language prolog
               :content "a(X)"   )
).
```

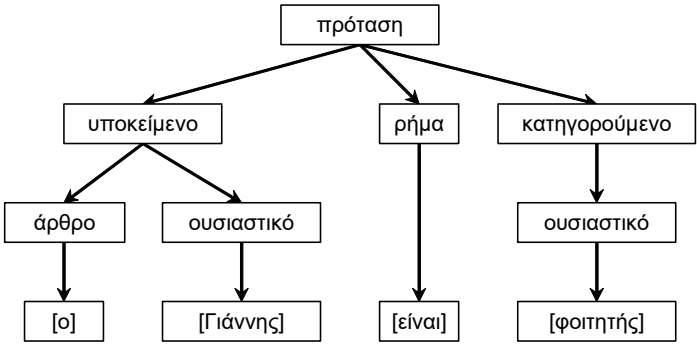
Το πρόγραμμα μπορεί εύκολα να τροποποιηθεί έτσι ώστε ο πράκτορας να καταλαβαίνει δύο επιπλέον ενέργειες:

Ενέργεια (performative)	S: Αποστολέας (Sender), R: Παραλήπτης (Receiver)	Απαιτείται απάντηση
`stream-all`	Σαν το `ask-all`, μόνο που οι πολλαπλές απαντήσεις στέλνονται με πολλά μηνύματα.	Ναι
eos	Τέλος μετάδοσης πολλαπλής απάντησης (end-of-stream).	Όχι

Π1.2.6 Επεξεργασία Φυσικής Γλώσσας

Η γλώσσα λογικού προγραμματισμού PROLOG είναι η κατεξοχήν γλώσσα για ανάπτυξη εφαρμογών επεξεργασίας φυσικής γλώσσας γιατί υποστηρίζει την άμεση αναπαράσταση γραμματικών οριστικών προτάσεων καθώς και πλούσιων και ευέλικτων

δομών για την αναπαράσταση των δένδρων συντακτικής ανάλυσης (Σχήμα Π1.6). Επίσης, λόγω του δηλωτικού της χαρακτήρα, η γλώσσα PROLOG βοηθάει στην αναπαράσταση της γνώσης που αποκομίζεται από τη σημασιολογική ανάλυση, καθώς και στην εξαγωγή συμπερασμάτων που είναι αναγκαία στο στάδιο της πραγματολογικής ανάλυσης.



Σχήμα Π1.6: Συντακτική ανάλυση της πρότασης "*Ο Γιάννης είναι φοιτητής*".

Μία γραμματική οριστικών προτάσεων έχει στην PROLOG την ακόλουθη μορφή:

```
protasi --> ypokeimeno, rhma, antikeimeno.
protasi --> ypokeimeno, rhma, kathgoroymeno.
ypokeimeno --> arthro, oysiastiko.
rhma --> [einai].
rhma --> [exei].
kathgoroymeno --> epitheto.
kathgoroymeno --> oysiastiko.
antikeimeno --> oysiastiko.
epitheto --> [nea].
arthro --> [o].
arthro --> [h].
oysiastiko --> [foithths].
oysiastiko --> ['Giannhs'].
oysiastiko --> ['Maria'].
```

Η παραπάνω γραμματική μπορεί να αναγνωρίσει προτάσεις, όπως αυτές στο Σχήμα Π1.6, δίνοντας σε έναν διερμηνέα της γλώσσας PROLOG την παρακάτω κλήση:

```
?- protasi([o,'Giannhs',einai,foithths],[]).
```

Η παραπάνω κλήση τροφοδοτεί τη γραμματική με μία πρόταση (πρώτο όρισμα) που αναπαρίσταται με λίστα (δηλαδή μια ακολουθία λέξεων), απαιτώντας από τη γραμματική να τις "καταναλώσει" όλες, δηλαδή η λίστα λέξεων που περισσεύει να είναι κενή (δεύτερο όρισμα).

Η ίδια γραμματική μπορεί να χρησιμοποιηθεί και για την παραγωγή φυσικής γλώσσας. Η κλήση:

```
?- protasi(L,[]).
```

επιστρέφει όλες τις δυνατές προτάσεις που μπορεί να αναγνωρίσει η γραμματική, όπως για παράδειγμα:

```
L = [o,'Giannhs',einai,foiththsh]
L = [h,'Maria',einai,nea]
...
```

Βέβαια, λόγω έλλειψης περιορισμών, στο συγκεκριμένο παράδειγμα παράγονται και ανούσιες προτάσεις, όπως για παράδειγμα `[o,foiththsh,einai,foiththsh]` ή και λανθασμένες, όπως `[o,'Giannhs',einai,'Maria']`, `[o,'Maria',exei,foiththsh]`, κτλ.

Μία ρεαλιστική γραμματική για ένα μεγάλο υποσύνολο της ελληνικής γλώσσας (αλλά και οποιασδήποτε άλλης γλώσσας) πρέπει να περιέχει αναλυτικά όλες τις κλίσεις των άρθρων, επιθέτων, ουσιαστικών και να αναγνωρίζει όλους τους ρηματικούς τύπους των ρημάτων που περιέχει. Αυτό μπορεί να γίνει είτε με την πολύ κοπιαστική και ποικιλοτρόπως δαπανηρή, διαδικασία της αναλυτικής καταγραφής όλων αυτών των τύπων, ή με τη χρήση βοηθητικών προγραμμάτων που θα περιέχουν τους κανόνες της γραμματικής, βάσει των οποίων παράγονται και αναγνωρίζονται οι ρηματικοί τύποι, οι κλίσεις των ουσιαστικών και των επιθέτων, κτλ.

Επίσης, η γραμματική θα πρέπει να συνοδεύεται από ένα λεξικό οργανωμένο έτσι ώστε να κατατάσσει σημασιολογικά όλες τις λέξεις ανάλογα με τον τρόπο χρήσης τους. Για παράδειγμα, τα διάφορα ουσιαστικά θα πρέπει να καταταγούν σε κατηγορίες όπως έμψυχα, φαγώσιμα, κτλ., έτσι ώστε να μην μπορεί να αναγνωριστούν και να παραχθούν προτάσεις χωρίς νόημα, όπως "*Ο Γιάννης έφαγε τον υπολογιστή*", η οποία είναι συντακτικά ορθή, αλλά νοηματικά είναι λανθασμένη. Επιπλέον, το κάθε ρήμα θα πρέπει να σχετίζεται με την κατηγορία στην οποία ανήκει το ουσιαστικό που θα είναι υποκείμενό του, καθώς και με την κατηγορία στην οποία ανήκει το ουσιαστικό που θα είναι αντικείμενό του, όταν φυσικά το ρήμα είναι μεταβατικό. Με αυτό τον τρόπο δεν θα είναι δυνατό να παραχθούν φράσεις όπως "*Ο φοιτητής είναι Μαρία*". Οι εννοιολογικοί γράφοι που παρουσιάζονται σε άλλο σημείο του βιβλίου, παρέχουν μια συστηματική προσέγγιση για την κωδικοποίηση τέτοιας πληροφορίας. Γενικότερα, τα λεξικά που οργανώνονται σημασιολογικά ονομάζονται *οντολογίες* (*ontologies*) και περιγράφονται σε σχετικό κεφάλαιο.

Μία απλή παραλλαγή της προηγούμενης γραμματικής η οποία παίρνει υπόψη χαρακτηριστικά της κλίσης των ουσιαστικών και ρημάτων, όπως το γένος, τον αριθμό και την πτώση των ουσιαστικών, καθώς και τον αριθμό των ρημάτων φαίνεται παρακάτω:

```
protasi -->
   ypokeimeno(Genos1,Arithmos1,onomastikh),
   rhma(Arithmos1),
   antikeimeno(Genos2,Arithmos2,aitiatikh).
protasi -->
   ypokeimeno(Genos,Arithmos,onomastikh),
   rhma(Arithmos),
   kathgoroymeno(Genos,Arithmos,onomastikh).
ypokeimeno(Genos,Arithmos,Ptwsh) -->
   arthro(Genos,Arithmos,Ptwsh),
   oysiastiko(Genos,Arithmos,Ptwsh).
rhma(enikos) --> [einai].
rhma(enikos) --> [exei].
kathgoroymeno(Genos,Arithmos,Ptwsh) -->
   epitheto(Genos,Arithmos,Ptwsh).
kathgoroymeno(Genos,Arithmos,Ptwsh) -->
   oysiastiko(Genos,Arithmos,Ptwsh).
antikeimeno(Genos,Arithmos,Ptwsh) -->
   oysiastiko(Genos,Arithmos,Ptwsh).
epitheto(thiliko,enikos,aitiatikh) --> [nea].
arthro(arseniko,enikos,onomastikh) --> [o].
arthro(thiliko,enikos,onomastikh) --> [h].
oysiastiko(arseniko,enikos,onomastikh) --> [foithths].
oysiastiko(arseniko,enikos,onomastikh) --> ['Giannhs'].
oysiastiko(thiliko,enikos,onomastikh) --> ['Maria'].
```

Τα χαρακτηριστικά έχουν υλοποιηθεί ως παράμετροι οι οποίες συνοδεύουν τις λέξεις του λεξικού και οι οποίες διαδίδονται στους γενικότερους κανόνες έτσι ώστε να υπάρχει συμφωνία των χαρακτηριστικών αυτών μεταξύ διαφορετικών τμημάτων της πρότασης. Για παράδειγμα, ο αριθμός του υποκειμένου πρέπει να συμφωνεί με τον αριθμό του ρήματος, ενώ το γένος, η πτώση και ο αριθμός του άρθρου του υποκειμένου πρέπει να συμφωνεί με τα αντίστοιχα χαρακτηριστικά της κλίσης του ουσιαστικού του υποκειμένου. Σύμφωνα με αυτήν τη γραμματική, προτάσεις όπως η "*Η Γιάννης είναι νέα*" δεν αναγνωρίζονται, ενώ δε συνέβαινε το ίδιο με την προηγούμενη, απλούστερη γραμματική.

Δένδρο συντακτικής ανάλυσης

Για να επιστραφεί στο χρήστη το *δένδρο* της *συντακτικής ανάλυσης* (*parse tree*) θα πρέπει οι γραμματικοί κανόνες να επεκταθούν έτσι ώστε να επιστρέφουν ως αποτέλεσμα της αναγνώρισης του τμήματος της πρότασης που τους αναλογεί, έναν σύνθετο όρο ο οποίος να αναπαριστά τη δομή του συγκεκριμένου τμήματος. Οι πιο γενικοί γραμματικοί κανόνες συνδυάζουν τους σύνθετους όρους των πιο συγκεκριμένων κανόνων σε ακόμα πιο σύνθετες δομές.

Η παρακάτω γραμματική αποτελεί επέκταση της αρχικής, απλής γραμματικής, με ένα επιπλέον όρισμα που επιστρέφει το συντακτικό δένδρο:

```
protasi(protasi(Y,R,A)) -->
   ypokeimeno(Y), rhma(R), antikeimeno(A).
protasi(protasi(Y,R,K)) -->
   ypokeimeno(Y), rhma(R), kathgoroymeno(K).
ypokeimeno(ypokeimeno(A,O)) -->
   arthro(A), oysiastiko(O).
rhma(rhma(einai)) --> [einai].
rhma(rhma(exei)) --> [exei].
kathgoroymeno(kathgoroymeno(K)) --> epitheto(K).
kathgoroymeno(kathgoroymeno(K)) --> oysiastiko(K).
antikeimeno(antikeimeno(o)) --> oysiastiko(O).
epitheto(epitheto(nea)) --> [nea].
arthro(arthro(o)) --> [o].
arthro(arthro(h)) --> [h].
oysiastiko(oysiastiko(foithths)) --> [foithths].
oysiastiko(oysiastiko('Giannhs')) --> ['Giannhs'].
oysiastiko(oysiastiko('Maria')) --> ['Maria'].
```

Τα επιπλέον ορίσματα στον αρχικό κανόνα της γραμματικής μπαίνουν μπροστά από την πρόταση προς αναγνώριση, στην κλήση που πρέπει να γίνει. Έτσι η παρακάτω κλήση επιστρέφει στη μεταβλητή **A** το συντακτικό δένδρο που φαίνεται στο Σχήμα Π1.6.

```
?- protasi(A,[o,'Giannhs',einai,foithths],[]).
A = protasi(ypokeimeno(arthro(o),oysiastiko('Giannhs')),rhma(einai),
            kathgoroymeno(oysiastiko(foithths)))
```

Βιβλιογραφία

Το πλέον διαδεδομένο βιβλίο στη γλώσσα PROLOG είναι το [Bratko, 2011] στο οποίο υπάρχουν πολλά παραδείγματα για προγραμματισμό εφαρμογών ΤΝ. Ένα πιο παλιό και πιο εισαγωγικό βιβλίο είναι το [Clocksin & Mellish, 2003], ενώ μία πιο προχωρημένη και θεωρητικά πληρέστερη αντιμετώπιση υπάρχει στο [Sterling & Shapiro, 1994]. Στην Ελληνική βιβλιογραφία υπάρχει το βιβλίο [Κατζουράκη, κ.ά., 1991], στο οποίο παρουσιάζεται η "Ελληνική" Δ- PROLOG καθώς και εφαρμογές ΤΝ.

Στην Ελληνική γλώσσα υπάρχει επιπλέον το "e-Book για την Prolog και τον Λογικό Προγραμματισμό" [Σακελλαρίου, κ.ά., 2015] που είναι διαθέσιμο on-line στον Κάλλιπο στη διεύθυνση: https://repository.kallipos.gr/handle/11419/777 .

Για τις διάφορες υλοποιήσεις της PROLOG, κάποιος μπορεί να ανατρέξει στις αντίστοιχες ιστοσελίδες, όπως για παράδειγμα στην https://sicstus.sics.se/ για τη SICSTUS PROLOG, στην https://www.lpa.co.uk/ για την LPA PROLOG, και στην https://www.swi-prolog.org/ για την SWI-PROLOG.

Αναφορές

[Bratko, 2011] I. Bratko, *Prolog Programming for Artificial Intelligence*, 4th edition, Pearson, 2011.

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

[Sterling & Shapiro, 1994] L. Sterling and E. Shapiro, *The Art of Prolog*, 2nded.MIT Press,1994

[Κατζουράκη, κ.ά., 1991] Μ. Κατζουράκη, Μ. Γεωργατσούλης και Σ. Κόκκοτος, *PROγραμματίζοντας στη LOGική*, ΕΠΥ, 1991.

[Σακελλαρίου, κ.ά., 2015] Η. Σακελλαρίου, Ν. Βασιλειάδης, Π. Κεφαλάς και Δ. Σταμάτης, *Τεχνικές Λογικού Προγραμματισμού - Η Γλώσσα Prolog*, [ηλεκτρ. βιβλ.], Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών, 2015. Διαθέσιμο στο: http://hdl.handle.net /11419/777.