Λογικός Προγραμματισμός
Η γλώσσα Prolog
(Σημειώσεις)

Χρήστος Νομικός

Ιωάννινα 2015

Περιεχόμενα

Κεφάλαιο 1

1.1 Λογικός Προγραμματισμός

Ο λογικός προγραμματισμός βασίζεται στην ιδέα να χρησιμοποιηθεί η μαθηματική λογική ως γλώσσα προγραμματισμού. Στο λογικό προγραμματισμό, το πρόγραμμα συνίσταται από ένα σύνολο προτάσεων που περιγράφουν το πρόβλημα που θέλουμε να επιλύσουμε. Η υλοποίηση της γλώσσας είναι στην ουσία ένα αποδεικτικό σύστημα, στο οποίο δίνουμε προτάσεις τις οποίες αυτό αποδεικνύει από τις προτάσεις που συνιστούν το πρόγραμμα.

1.2 Η γλώσσα Prolog

Η Γλώσσα Prolog είναι η πρώτη χρονολογικά και πιο διαδεδομένη γλώσσα λογικού προγραμματισμού. Αναπτύχθηκε στις αρχές της δεκαετίας του 1970. Στηρίχθηκε κυρίως στη δουλειά του Robert Kowalski και υλοποιήθηκε από τον Alain Colmerauer. Το όνομά της αποτελεί συντομογραφία του PROgramming in LOGic.

Ένα πρόγραμμα Prolog αποτελείται από προτάσεις της πρωτοβάθμιας λογικής οι οποίες έχουν μία ειδική μορφή (προτάσεις Horn). Επιπλέον, η Prolog διαθέτει προκαθορισμένα κατηγορήματα, που υποστηρίζουν χαρακτηριστικά δευτεροβάθμιας λογική, όπως επίσης και αριθμητικούς τελεστές.

Για να πετύχουμε το επιθυμητό αποτέλεσμα κάνουμε στη Prolog μία ερώτηση, η οποία αποτελείται από τη σύζευξη ένος πλήθους ατομικών προτάσεων και ενδέχεται να περιέχει μεταβλητές. Η Prolog προσπαθεί να αποδείξει την πρόταση από τις προτάσεις που αποτελούν το πρόγραμμα, και μας επιστρέφει τις συνθήκες κάτω από τις οποίες η πρότασή στην ερώτηση αποτελεί λογική συνέπεια του προγράμματος, οι οποίες εκφράζονται ως περιορισμοί στις τιμές των μεταβλητών.

Στη συνέχεια θα χρησιμοποιήσουμε τη γλώσσα Prolog για να κάνουμε μία εισαγωγή στις έννοιες του λογικού προγραμματισμού.

Ο σκοπός αυτών των σημειώσεων δεν είναι σε καμμία περίπτωση το να περιγραφεί πλήρως η γλώσσα Prolog. Θα δούμε πως ορίζονται κατηγορήματα πάνω σε συμβολικά αντικείμενα. Επίσης θα δούμε πώς μπορούμε να ορίσουμε αριθμητικές

συναρτήσεις και να χειριστούμε λίστες. Τέλος θα εξετάσουμε την άρνηση ως αποτυχία που χρησιμοποιεί η Prolog, τον έλεγχο της οπισθοδρόμησης και το πώς η εκτέλεση ενός προγράμματος μπορεί να τροποποιεί το ίδιο το πρόγραμμα. Αντίθετα θα παραλείψουμε την περιγραφή θεμάτων που αφορούν είσοδο-έξοδο για δημιουργία διαδραστικών εφαρμογών καθώς και την παρουσίαση πολλών προκαθορισμένων κατηγορημάτων.

Στην Prolog μπορούμε να φορτώσουμε ταυτόχρονα πολλά διαφορετικά προγράμματα (σύνολα προτάσεων). Στη συνέχεια αν στον ορισμό ενός κατηγορήματος χρησιμοποιούμε ένα κατηγόρημα που έχουμε ορίσει προηγουμένως, θα υποθέτουμε ότι και οι δύο ορισμοί περιέχονται στο ίδιο αρχείο.

1.3 Ο διερμηνέας GNU-Prolog της Prolog

Για να εκτελεστεί ο διερμηνέας GNU-Prolog σε σύστημα Unix γράφουμε στο τερματικό `prolog` και πατάμε `<ENTER>`. Ο διερμηνέας εμφανίζει την prompt

```
| ?-
```

Σε αυτή τη φάση ο διερμηνέας μπορεί να μας απαντήσει σε ερωτήσεις που περιλαμβάνουν προκαθορισμένα κατηγορήματα και τελεστές της Prolog. Ο διερμηνέας θα μας απαντήσει `yes` ή `no` ή, αν η ερώτησή μας περιέχει μεταβλητές, θα μας επιστρέψει τις τιμές των μεταβλητών για τις οποίες η πρόταση στην ερώτηση αληθεύει (θα πρέπει μετά την ολοκλήρωση της ερώτησης να βάλουμε τελεία και να πατήσουμε `<ENTER>`):

```
| ?- 1+1 =:= 2.
yes

| ?- f(a,X) = f(Y,b).
X = b
Y = a
```

Αν γράψουμε μία ερώτηση που περιέχει άλλα κατηγορήματα ο διερμηνέας απλά θα επιστρέψει ένα μήνυμα της μορφής

```
uncaught exception: error(existence_error(procedure,p/2),top_level/0)
```

(σημειώνεται ότι σε άλλες εκδόσεις της Prolog σε ανάλογη περίπτωση η απάντηση είναι `no`.) Μπορούμε να φορτώσουμε ένα αρχείο προγράμματος π.χ. το `example.pro`, το οποίο βρίσκεται στον κατάλογο από τον οποίο εκκινήσαμε τη GNU-Prolog γράφοντας:

```
| ?- consult('example.pro').
```

Το consult είναι προκαθορισμένο κατηγόρημα της Prolog (αυτό σημαίνει ότι μπορεί να περιέχεται και σε προγράμματα Prolog). Αν το αρχείο υπάρχει στον κατάλογο από τον οποίο εκτελέσαμε την εντολη prolog και είναι συντακτικά ορθό, τότε ο διερμηνέας θα μας δώσει μία απάντηση παρόμοια με την παρακάτω:

```
example.pro compiled, 2 lines read - 353 bytes written, 98 ms

yes
```

Η απάντηση αυτή μας λέει ότι το αρχείο `example.pro` φορτώθηκε και μας ενημερώνει για το μέγεθος του σε χαρακτήρες και το χρόνο που χρειάστηκε για την επεξεργασία του. Αφότου φορτώσουμε το πρόγραμμα ο διερμηνέας μπορεί να μας απαντήσει σε ερωτήσεις που περιέχουν κατηγορήματα τα οποία ορίζονται σε αυτό.

Το `consult('example.pro')` μπορεί πιο σύντομα να γραφτεί `['example.pro']`. Αν το αρχείο `example.pro` βρίσκεται σε διαφορετικό κατάλογο από αυτόν από τον οποίο εκτελέσαμε την εντολή prolog, τότε θα πρέπει να γράψουμε το πλήρες μονοπάτι ως όρισμα του `consult`. Αν το πρόγραμμα δεν είναι συντακτικά ορθό, τότε εμφανίζεται ένα μηνυμα που μας ενημερώνει για το συντακτικό λάθος. Αν το αρχείο `example.pro` δεν υπάρχει τότε εμφανίζεται ένα μήνυμα παρόμοιο με το παρακάτω:

```
uncaught exception: error(existence_error(source_sink,'example.pro'),
consult/1)
```

Οι προτάσεις που αποτελούν το πρόγραμμα το οποίο περιέχεται στο αρχείο που φορτώνεται με το `consult` αποθηκεύονται στη *βάση δεδομένων* της Prolog, οποία χρησιμοποιείται για να απαντηθούν οι ερωτήσεις που κάνουμε στο διερμηνέα. Η βάση δεδομένων όταν ξεκινάει ο διερμηνέας είναι κενή. Κάθε φορά που χρησιμοποιούμε το `consult` για να φορτώσουμε ένα αρχείο, διαγράφονται από τη βάση δεδομένων όλες οι προτάσεις που ορίζουν κάποιο κατηγόρημα p με n ορίσματα, για το οποίο υπάρχει ορισμός για ίδιο πλήθος ορισμάτων στο αρχείο που πρόκειται να φορτωθεί, ανεξάρτητα από το πώς είχαν εισαχεί στη βάση δεδομένων. Ωστόσο, οι υπόλοιπες προτάσεις παραμένουν στη βάση δεδομένων.

Τονίζεται ότι το `consult` της GNU-Prolog λειτουργεί όπως το κατηγόρημα `reconsult` σε άλλες εκδόσεις της Prolog, στις οποίες το `consult` φορτώνει το νέο αρχείο χωρίς να διαγράψει προτάσεις από τη βάση δεδομένων.

Η εκτέλεση της GNU-Prolog τερματίζεται γράφοντας `halt` ή πατώντας CTRL-D.

1.4 Όροι και Απλές Προτάσεις

Ένα πρόγραμμα Prolog αποτελείται από ένα σύνολο προτάσεων (γεγονότων και κανόνων) και το επιθυμητό αποτέλεσμα υπολογίζεται δίνοντας στο διερμηνέα της Prolog μία ερώτηση.

Βασική δομική μονάδα για το σχηματισμό των γεγονότων, των κανόνων και των ερωτήσεων είναι η ατομική πρόταση, η οποία σχηματίζεται από ένα κατηγόρημα το οποίο παίρνει ως ορίσματα ένα πλήθος όρων. Τα ορίσματα ενός κατηγορήματος δίνονται μέσα σε παρένθεση, χωρισμένα με κόμμα. Ένα κατηγόρημα με ένα όρισμα αποδίδει μία ιδιότητα στον όρο που δέχεται ως όρισμα. Ένα κατηγόρημα με περισσότερα ορίσματα περιγράφει μία σχέση ανάμεσα στους όρους που δέχεται ως ορίσματα.

Για παράδειγμα στην ατομική πρόταση `born('Robert De Niro', date(17,aug,1943))` το κατηγόρημα είναι το `born` και τα ορίσματα είναι οι όροι `'Robert De Niro'` και `date(17,aug,1943)`.

Οι όροι παριστάνουν αντικείμενα του κόσμου που θέλουμε να περιγράψουμε με το πρόγραμμα. Οι όροι μπορεί να είναι απλοί ή σύνθετοι. Οι απλοί όροι είναι σταθερές ή μεταβλητές. Οι σταθερές είναι είτε άτομα είτε αριθμοί. Οι σύνθετοι όροι σχηματίζονται από ένα συναρτησιακό σύμβολο το οποία δέχεται ως ορίσματα ένα πλήθος όρων (μέσα σε παρενθέσεις και χωρισμένα με κόμμα). Παραδείγματα όρων:

- αριθμοί: `1968`, `-12`, `3.14`
- άτομα: `aug`, `'Robert De Niro'`
- μεταβλητές: `X`, `Variable`, `_1`
- σύνθετοι όροι: `date(17,aug,1943))`, `point(2.0,3.4)`, `s(s(s(0)))`

(οι κανόνες σύνταξης για τα άτομα τα συναρτησιακά σύμβολα και τα κατηγορήματα θα δοθούν παρακάτω).

1.5 Γεγονότα

Ένα γεγονός είναι μία ατομική πρόταση η οποία αληθεύει χωρίς καμία προϋπόθεση.

Παράδειγμα 1: Τα παρακάτω είναι γεγονότα που μπορεί να περιέχονται σε ένα πρόγραμμα PROLOG:

```
lp('Chet Baker','My Funny Valentine',1954).
lp('Billie Holiday','Lady Sings the Blues',1956).
lp('Van Morrison','Astral Weeks',1968).
lp('Leonard Cohen','Songs From a Room',1969).
lp('The Rolling Stones','Sticky Fingers',1971).
lp('Tom Waits','Closing Time',1973).
lp('Pink Floyd','The Dark Side of the Moon',1973).
lp('The Jam','Sound Affects',1980).
lp('Tuxedomoon','Desire',1981).
lp('The Beautiful South','Quench',1998).
lp('Tom Waits','Alice',2002).
```

Το πρώτο γεγονός δηλώνει ότι ο Chet Baker κυκλοφόρησε το άλμπουμ My Funny Valentine το 1954. Αντίστοιχα ερμηνεύονται και τα υπόλοιπα γεγονότα. Το κατηγόρημα `lp` συσχετίζει τρία διαφορετικά αντικείμενα: έναν καλλιτέχη, τον τίτλο ενός άλμπουμ και έναν αριθμό που παριστάνει μία χρονολογία.

Η Prolog δεν απαιτεί να δηλώνουμε το πλήθος και τον τύπο των ορισμάτων ενός κατηγορήματος. Σε όλα τα παραπάνω γεγονότα, τα δύο πρώτα ορίσματα του `lp`

είναι συμβολικοί απλοί όροι (άτομα) ενώ το τρίτο είναι αριθμός. Αυτό ωστόσο δεν είναι δεσμευτικό. Για παράδειγμα θα μπορούσαμε να γράψουμε στο πρόγραμμα το γεγονός `lp(2,2).` χωρίς να υπάρχει συντακτικό λάθος.

Μπορούμε να επεκτείνουμε το πρόγραμμα με γεγονότα που περιέχουν πληροφορίες για τα τραγούδια του κάθε άλμπουμ:

```
track('Lady Sings the Blues',4,'Love Me or Leave Me',
                                          duration(2,54)).
track('Astral Weeks',2,'Beside You',duration(5,16)).
track('Sticky Fingers',3,'Wild Horses',duration(5,44)).
track('Quench',5,'Perfect Ten',duration(3,38)).
track('Alice',1,'Alice',duration(4,28)).
```

Το πρώτο γεγονός για παράδειγμα δηλώνει ότι στο αλμπουμ Lady Sings the Blues το 4ο τραγούδι είναι το Love Me or Leave Me με διάρκεια 2 λεπτά και 54 δευτερόλεπτα. Χρησιμοποιούμε έναν σύνθετο όρο για να καταγράψουμε τη διάρκεια του τραγουδιού. ∎

1.6 Ερωτήσεις

Μία ερώτηση αποτελείται από τη μία ή περισσότερες ατομικές προτάσεις, χωρισμένες με κόμμα. Όταν γράψουμε μία ερώτηση στόν διερμηνέα της Prolog, αυτός εξετάζει αν η σύζευξη των προτάσεων που την αποτελούν προκύπτει από το πρόγραμμα, δηλαδή αν είναι λογική συνέπεια των προτάσεων που αποτελουν το πρόγραμμα.

Αν η ερώτηση δεν περιέχει μεταβλητές τότε η Prolog απαντάει είτε `yes` είτε `no`.

Παράδειγμα 2: Μπορούμε αφού φορτώσουμε το πρόγραμμα που περιέχει τον ορισμό των κατηγορημάτων `lp` και `track` του προηγούμενου παραδείγματος να κάνουμε τις παρακάτω ερωτήσεις:

```
?- lp('The Rolling Stones','Sticky Fingers',1971).
yes
?- lp('Tom Waits','Alice',1973).
no
?- lp('The Rolling Stones','Aftermath',1966).
no
?- lp('Tom Waits','Alice',1973),lp('Tuxedomoon','Desire',1981).
no
?- member('Paul Weller', 'Jam').
no
```

Στην πρώτη ερώτηση η απάντηση είναι yes, καθώς η πρόταση που την αποτελεί υπάρχει μέσα στο πρόγραμμα.

Στην δεύτερη ερώτηση η απάντηση είναι no, καθώς η πρόταση που την αποτελεί δεν υπάρχει μέσα στο πρόγραμμα ούτε μπορεί να προκύψει λογικά από αυτό. Στο πρόγραμμα είναι καταχωρημένο το αλμπουμ Alice του Tom Waits αλλα η χρονολογία κυκλοφορίας δεν είναι το 1973, όπως επίσης και ένα άλμπουμ του Tom Waits με χρονολογία κυκλοφορίας το 1973 που ωστόσο δεν λέγεται Alice.

Στην τρίτη ερώτηση η απάντηση είναι επίσης no: οι Rolling Stones κυκλοφόρησαν πράγματι το άλμπουμ Aftermath το 1966, αλλά αυτό δεν υπάρχει ως γεγονός στο πρόγραμμα. Η Prolog γνωρίζει τον πραγματικό κόσμο μόνο μέσω των προτάσεων του προγράμματος.

Στην τέταρτη ερώτηση η απάντηση είναι no, επειδή μόνο μία από τις δύο προτάσεις που αποτελούν την ερώτηση προκύπτει από το πρόγραμμα.

Στην πέμπτη ερώτηση η απάντηση είναι προφανώς no. Παρατηρούμε ότι η Prolog δεν ενοχλείται από το γεγονός ότι το κατηγόρημα `member` δεν έχει οριστεί πουθενά μέσα στο πρόγραμμα. ∎

Αν η ερώτηση περιέχει μεταβλητές, τότε η Prolog προσπαθεί να βρεί κατάλληλες δεσμεύσεις για τις μεταβλητές, κάτω από τις οποίες η σύζευξη των προτάσεων που αποτελούν την ερώτηση αποτελεί λογική συνέπεια του προγράμματος.

Αν επιτύχει, τότε τυπώνει τις τιμές των μεταβλητών που υπολόγισε και αναμένει από το χρήστη να καθορίσει τι επιθυμεί να γίνει στη συνέχεια:

- αν ο χρήστης πληκτρολογήσει ; η Prolog συνεχίζει, προσπαθώντας να βρεί άλλη απάντηση (άλλες τιμές για τις μεταβλητές)
- αν ο χρήστης πληκτρολογήσει . η Prolog απαντάει yes και σταματάει (περιμένει νέα ερώτηση).

Αν η Prolog αποτύχει να βρει λύση (είτε αρχικά είτε μετά από ;) απαντάει no.

Μπορούμε να χρησιμοποιήσουμε την ανώνυμη μεταβλητή αν κάποιες από τις τιμές δεν μας ενδιαφέρουν. Η ανώνυμη μεταβλητή είναι η _ και έχει τις παρακάτω ιδιότητες

- δεν επιστρέφει τιμές
- διαφορετικές εμφανίσεις της στον ίδιο κανόνα ή ερώτηση είναι ασυσχέτιστες, μπορούν δηλαδή να πάρουν διαφορετικές τιμές.

Παράδειγμα 2 (συνέχεια): Για να μάθουμε τη χρονολογία κυκλοφορίας του άλμπουμ Sticky Fingers των Rolling Stones γράφουμε:

```
?- lp('The Rolling Stones','Sticky Fingers',Y).
Y = 1971
```

Για να μάθουμε αν οι Rolling Stones κυκλοφόρησαν κάποιο άλμπουμ το 1952 γράφουμε:

```
?- lp('The Rolling Stones',X,1952).
no
```

Για να δούμε όλα τα (καταχωρημένα) αλμπουμ του Tom Waits γράφουμε:

```
?- lp('Tom Waits',X,Y).
X = 'Closing Time'
Y = 1973 ;
X = 'Alice'
Y = 2002 ;
no
```

(Μετά από κάθε απάντηση γράφουμε ; ώστε η Prolog να συνεχίσει με την αναζήτηση άλλων απαντήσεων). Παρατηρούμε ότι η Prolog επιστρέφει τις απαντήσεις με ίδια σειρά που εμφανίζονται μέσα στο πρόγραμμα.

Για να μάθουμε αν υπάρχει άλμπουμ που περιέχει ομώνυμο τραγούδι γράφουμε:

```
?- track(X,N,X,D).
X = 'Alice'
N = 1
D = duration(4,28)
```

Η χρήση της ίδιας μεταβλητής ως πρώτο και τρίτο όρισμα, αναγκάζει αυτά τα ορίσματα να έχουν την ίδια τιμή.

Στην παραπάνω ερώτηση παίρνουμε τιμές για όλες τις μεταβλητές που εμφανίζονται. Μπορούμε να χρησιμοποιήσουμε την ανώνυμη μεταβλητή ώστε να μην εμφανιστούν τιμές που μας είναι αδιάφορες:

```
?- track(X,_,X,_).
X = 'Alice'
```

Για να μάθουμε ποιος τραγούδησε το Love Me or Leave Me γράφουμε:

```
?- track(L,_,'Love Me or Leave Me',_), lp(X,L,_).
L = 'Lady Sings the Blues'
X = 'Billie Holiday'
```

Επειδή η μεταβλητή L συνδέει τις δύο προτάσεις δεν μπορεί να αντικατασταθεί από την ανώνυμη μεταβλητή ∎

1.7 Κανόνες

Οι κανόνες είναι σύνθετες προτάσεις οι οποίες δηλώνουν ότι μία ατομική πρόταση (η κεφαλή του κανόνα) είναι αληθής αν ένα πλήθος ατομικών προτάσεων (οι οποίες αποτελούν το σώμα του κανόνα) είναι όλες αληθείς.

Οι κανόνες χρησιμοποιούνται για να ορίσουμε νέα κατηγορήματα με βάση τα ήδη υπάρχοντα, όπως επίσης και για να ορίσουμε κατηγορήματα με αναδρομή. Ένα κατηγόρημα μπορεί να ορίζεται από περισσότερους του ενός κανόνες ή ακόμη και από συνδυασμό κανόνων και γεγονότων.

Παράδειγμα 3: Ας υποθέσουμε ότι τα κατηγορήματα `parent(X,Y)` (ο/η `X` είναι γονέας του/της `Y`), `male(X)` (ο `X` είναι άντρας) και `female(X)` (η `X` είναι γυναίκα) έχουν οριστεί με ένα σύνολο γεγονότων.

Το κατηγόρημα `father(X,Y)` (ο `X` είναι πατέρας του/της `Y`) μπορεί να οριστεί από τον παρακάτω κανόνα:

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

Ο κανόνας λέει ότι ο `X` είναι πατέρας του/της `Y` αν συντρέχουν δύο προϋποθέσεις: (α) ο `X` είναι γονέας του/της `Y` και (β) ο `X` είναι αντρας.

Τα κατηγορήματα `grandfather(X,Y)` (ο `X` είναι παππούς του/της `Y`) μπορεί να οριστεί από τον παρακάτω κανόνα:

```
grandfather(X,Y) :- father(X,Z), parent(Z,Y).
```

Ο κανόνας λέει ότι ο `X` είναι παππούς του/της `Y` αν ο `X` είναι πατέρας κάποιου γονέα του `Y` (ο οποίος είναι ο `Z` στο σώμα του κανόνα).

Το κατηγόρημα `brother(X,Y)` (ο `X` είναι αδερφός του/της `Y`) ορίζεται από τον παρακάτω κανόνα:

```
brother(X,Y) :- parent(Z,X),
                parent(Z,Y),
                male(X),
                X \== Y.
```

Ο `X` είναι αδερφός του/της `Y` αν ο `X` είναι άντρας και οι `X`, `Y` έχουν κάποιον κοινό γονέα. Το `X \== Y` εξασφαλίζει ότι οι μεταβλητές `X` και `Y` δεν θα έχουν ίδια τιμή, ώστε κανείς να μην είναι αδερφός του εαυτού του.

Το κατηγόρημα `person(X)` (ο `X` είναι κάποιο φυσικό πρόσωπο) μπορεί να οριστεί χρησιμοποιώντας δύο κανόνες:

```
person(X) :- male(X).
person(X) :- female(X).
```

Ας υποθέσουμε ότι στο πρόγραμμα περιέχονται επίσης τα παρακάτω γεγονότα:

```
parent(tantalos,pelopas).
parent(dioni,pelopas).
parent(dioni,niovi).
parent(tantalos,niovi).
parent(pelopas,atreas).
male(tantalos).
male(pelopas).
male(atreas).
female(dioni).
female(niovi).
```

τότε μπορούμε να έχουμε τον παρακάτω διάλογο με το διερμηνέα της Prolog:

```
?- father(tantalos,pelopas).
yes
?- father(tantalos,X).
X = pelopas ;
X = niovi ;
no
?- father(dioni,X).
no

?- brother(pelopas,niovi).
yes
?- brother(niovi,pelopas).
no
```

1.8 Αναδρομή

Στην Prolog ονομάζουμε αναδρομή τη εμφάνιση του ίδιου κατηγορήματος (με το ίδιο πλήθος ορισμάτων) τόσο στην κεφαλή όσο και στο σώμα ενός κανόνα. Η χρήση αναδρομής είναι απαραίτητη ώστε να υλοποιήσουμε σε Prolog υπολογισμούς που απαιτούν επανάληψη.

Παράδειγμα 4: Έστω ότι θέλουμε να ορίσουμε το κατηγόρημα `predecessor(X,Y)` (ο `X` είναι πρόγονος του `Y`). Η πρόταση `predecessor(X,Y)` αληθεύει αν ο `X` και ο `Y` συνδέονται στο γενεαλογικό δέντρο που ορίζει το κατηγόρημα `parent` με ένα μονοπάτι πεπερασμένου μήκους.

Ας υποθέσουμε πρώτα ότι προσπαθούμε να ορίσουμε το `predecessor` μέσω κανόνων που δεν χρησιμοποιούν αναδρομή.

Για παράδειγμα, έστω ότι γράφουμε στο πρόγραμμα τους παρακάτω κανόνες:

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

Αν το `parent` ορίζεται από τα γεγονότα του προηγούμενου παραδείγματος, τότε οι παραπάνω κονόνες είναι επαρκείς. Αυτό όμως οφείλεται στο ότι το μέγιστο μονοπάτι στο γενεαλογικό δέντρο που περιγράφει το `parent` είναι 2. Αν προσθέσουμε τα γεγονότα

```
parent(atreas,menelaos).
parent(menelaos,nikostratos).
```

τότε η Prolog στην ερώτηση

```
?- predecessor(tantalos,nikostratos).
```

θα απαντήσει no. Αυτό οφείλεται στο ότι οι Τάνταλος και Νικόστρατος διαφέρουν 4 γενιές.

Θα μπορούσαμε να επεκτείνουμε τον ορισμό του `predecessor` προσθέτοντας και άλλους κανόνες. Ωστόσο πάντοτε θα υπήρχε ένα σύνολο γεγονότων η πρόσθεση των οποίων στο πρόγραμμα, θα καθιστούσε τον ορισμό του `predecessor` ελλιπή.

Η σωστή προσέγγιση είναι να ορίσουμε το `predecessor` αναδρομικά στηριζόμενοι στην παρακάτω παρατήρηση: Ο X είναι προγονος του Y με διαφορά 1 γενιά, αν ο X είναι γονέας του Y. Επίσης, ο X είναι προγονος του Y με διαφορά $n+1$ γενιές, αν ο X είναι γονέας κάποιου Z, ο οποίος είναι πρόγονος του Y με διαφορά n γενιές.

Στον παρακάτω ορισμό παραλείπεται το πλήθος των γενεών:

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

Εναλλακτικά θα μπορούσαμε να χρησιμοποιήσουμε αναδρομή αριστερά στο σώμα του κανόνα:

```
predecessor1(X,Y) :- parent(X,Y).
predecessor1(X,Y) :- predecessor1(X,Z),
                     parent(Z,Y).
```

Επίσης μπορεί να χρησιμοποιηθεί διπλή αναδρομή:

```
predecessor2(X,Y) :- parent(X,Y).
predecessor2(X,Y) :- predecessor2(X,Z),
                     predecessor2(Z,Y).
```

1.9 Συντακτικοί κανόνες

Οι μεταβλητές στην Prolog είναι ακολουθίες χαρακτήρων που σχηματίζονται από γράμματα του λατινικού αλφαβήτου, ψηφία και το χαρακτήρα _ (χαρακτήρας υπογράμμισης), οι οποίες αρχίζουν με κεφαλαίο γράμμα ή _.

Τα άτομα, τα κατηγορήματα και τα συναρτησιακά σύμβολα επιτρέπεται να έχουν μία από τις παρακάτω μορφές:

- ακολουθίες χαρακτήρων που σχηματίζονται από γράμματα του λατινικού αλφαβήτου, ψηφία και το χαρακτήρα _, οι οποίες αρχίζουν με μικρό γράμμα.
- ακολουθία ειδικών χαρακτήρων που είναι οι παρακάτω:
  `+ - * / < > = : . & _ ~`
- οποιαδήποτε ακολουθία χαρακτήρων κλεισμένη μέσα σε εισαγωγικά.

Ένα κατηγόρημα ή ένα συναρτησικό σύμβολο ακολουθείται από τα ορίσματά του κλεισμένα σε παρενθέσεις και χωρισμένα με κόμμα. Η αριστερή παρένθεση θα πρέπει να γράφεται αμέσως μετά το κατηγόρημα ή το συναρτησικό σύμβολο, χωρίς να παρεμβάλεται κενό.

Στο τέλος κάθε γεγονότος, κανόνα ή ερώτησης πρέπει να υπάρχει τελεία.

Στη Prolog μπορούμε να γράψουμε σχόλια κλείνοντάς τα ανάμεσα στα σύμβολα `/*` και `*/`. Εναλλακτικά τα σχόλια μπορούν να ξεκινήσουν με το σύμβολο %. Σε αυτή την περίπτωση εκτείνονται μέχρι το τέλος της γραμμής.

1.10 Ταυτοποίηση

Το επιθυμητό αποτέλεσμα στην Prolog επιτυγχάνεται δίνοντας στο διερμηνέα μία ερώτηση η οποία αποτελείται μία πρόταση την οποία η Prolog προσπαθεί να αποδείξει με βάση το πρόγραμμα (ή ακριβέστερα με βάση τις προτάσεις που έχει φορτώσει στη βάση δεδομένων της). Στη συνέχεια θα δούμε το μηχανισμό τον οποίο η Prolog χρησιμοποιεί για να απαντήσει σε μία ερώτηση. Στο μηχανισμό αυτό πολύ σημαντική είναι η έννοια της ταυτοποίησης.

Μία ισότητα της μορφής $V = t$, όπου V είναι μεταβλητή και t είναι όρος ονομάζεται δέσμευση της μεταβλητής V.

Θα χρησιμοποιούμε το όρο έκφραση για κάτι που είναι είτε όρος είτε ατομική πρόταση.

Δύο εκφράσεις ονομάζονται ταυτοποιήσιμες αν μπορούν να γίνουν ίδιες με ενδεχόμενη αντικατάσταση κάποιων από τις μεταβλητές που εμφανίζονται σε αυτές, με βάση ένα σύνολο δεσμεύσεων. Το σύνολο των δεσμεύσεων που ταυτοποιεί δύο εκφράσεις ονομάζεται ταυτοποιητής.

Η αντικατάσταση των μεταβλητών είναι ομοιόμορφη και ταυτόχρονη. Ομοιόμορφη σημαίνει ότι υπάρχει μία μόνο δέσμευση για κάθε μεταβλητή και όλες οι εμφανίσεις της μεταβλητής και στις δύο εκφράσεις αντικαθίστανται από τον ίδιο όρο. Ταυτόχρονη σημαίνει ότι οι αντικαταστάσεις δεν γίνονται η μία μετά την άλλη με κάποια αυθαίρετη σειρά άλλά όλες μαζί, έτσι ώστε αν κάποιος από τους νέους όρους περιέχει κάποια μεταβλητή απο αυτές που αντικαταστάθηκαν, οι εμφανίσεις της που προήλθαν από την εισαγωγή αυτού του όρου να μην επηρεαστούν.

Παράδειγμα 5:

- Οι εκφράσεις `south` και `north` δεν είναι ταυτοποιήσιμες, καθώς δεν είναι ίδιες ούτε περιέχουν μεταβλητές που θα μπορούσαν να αντικατασταθούν ώστε να γίνουν ίδιες.

- Οι εκφράσεις `west` και `west` είναι ταυτοποιήσιμες, καθώς είναι ίδιες.

- Οι εκφράσεις `X` και `123` είναι ταυτοποιήσιμες, καθώς μπορούν να γίνουν ίδιες αν η μεταβλητή `X` αντικατασταθεί από το όρο `123`.

- Οι εκφράσεις `X` και `Y` είναι ταυτοποιήσιμες, καθώς μπορούν να γίνουν ίδιες αν οι μεταβλητές `X` και `Y` αντικατασταθούν από τον ίδιο όρο.

- Οι εκφράσεις `date(D,may,1995)` και `date(1,M,Y)` είναι ταυτοποιήσιμες, από το σύνολο δεσμεύσεων $\{\texttt{D} = 1, \texttt{M} = \texttt{may}, \texttt{Y} = 1995\}$, με βάση το οποίο γίνονται και οι δύο `date(1,may,1995)`.

- Οι εκφράσεις `time(8,45)` και `point(X,Y)` δεν είναι ταυτοποιήσιμες: όποια αντικατάσταση μεταβλητών και να γίνει η πρώτη θα ξεκινάει με `time` και η δεύτερη με `point`.

- Οι εκφράσεις `vector(X,Y,Z)` και `vector(A,B)` δεν είναι ταυτοποιήσιμες: όποια αντικατάσταση μεταβλητών και να γίνει το `vector` στη πρώτη θα έχει τρία ορίσματα και στη δεύτερη δύο.

- Οι εκφράσεις `date(X,12,1930)` και `date(4,X,Y)` δεν είναι ταυτοποιήσιμες: για να γίνουν ίδια τα πρώτα ορίσματα του `date` η μεταβλητή `X` θα πρέπει να αντικατασταθεί από το `4`. Ωστόσο σε αυτή την περίπτωση δεν θα είναι ίδια τα δεύτερα ορίσματα του `date`.

- Οι εκφράσεις `f(Y)` και `f(f(X))` είναι ταυτοποιήσιμες, από το σύνολο δεσμεύσεων $\{\texttt{Y} = \texttt{f(Z)}, \texttt{X} = \texttt{Z}\}$, με βάση το οποίο γίνονται και οι δύο `f(f(Z))`.

- Οι εκφράσεις `f(X)` και `f(f(X))` δεν είναι ταυτοποιήσιμες: αν η `X` αντικατασταθεί με έναν όρο μήκους n χαρακτήρων η έκφραση που θα προκύψει από την πρώτη μετά την αντικατάσταση θα έχει μήκος $n + 3$, ενώ αυτή που θα προκύψει από τη δεύτερη θα έχει μήκος $n + 6$.

Δύο εκφράσεις μπορεί να έχουν περισσότερους από έναν ταυτοποιητές.

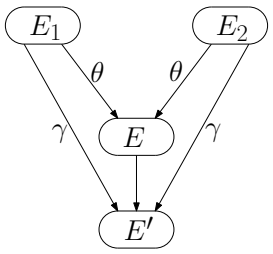
Παράδειγμα 6: Οι εκφράσεις `date(15,M,X)` και `date(D,9,Y)` έχουν τους παρακάτω ταυτοποιητές:

- $\{$`D` $=$ `15`, `M` $=$ `9`, `X` $=$ `Y`$\}$
- $\{$`D` $=$ `15`, `M` $=$ `9`, `X` $=$ `W`, `Y` $=$ `W`$\}$
- $\{$`D` $=$ `15`, `M` $=$ `9`, `X` $=$ `1900`, `Y` $=$ `1900`$\}$

Από τους παραπάνω ταυτοποιητές, ο τρίτος φαίνεται να δεσμεύει τις μεταβλητές `X`, και `Y` περισσότερο από όσο χρειάζεται ώστε να επιτευχθεί η ταυτοποίηση. Αντίθετα δεν συμβαίνει το ίδιο με τους δύο πρώτους ταυτοποιητές, οι οποίοι κάνουν την ελάχιστη δυνατή δέσμευση που απαιτείται ώστε οι εκφράσεις να γίνουν ίδιες. Τέτοιου είδους ταυτοποιητές ονομάζονται πιο γενικοί ταυτοποιητές. ∎

Πιο αυστηρά ένας ταυτοποιητής θ των εκφράσεων E_1 και E_2 ονομάζεται πιο γενικός ταυτοποιητής αν η έκφραση που προκύπτει από τις E_1 και E_2 με εφαρμογή του οποιοσδήποτε ταυτοποιητή γ, μπορεί να προκύψει με περαιτέρω αντικατάσταση μεταβλητών στην έκφραση που προκύπτει από τις E_1 και E_2 με εφαρμογή του θ.

Σχηματικά:



Η Prolog χρησιμοποιεί εκτενώς την ταυτοποίηση στη διαδικασία απάντησης μίας ερώτησης. Επίσης διαθέτει τον τελεστή `=` ο οποίος πραγματοποιεί ταυτοποίηση δύο εκφράσεων:

```
?- date(D,may,1995) = date(1,M,Y).
D = 1
M = may
Y = 1995

?- time(8,45) = point(X,Y).
no
```

Πριν περιγράψουμε το αλγόριθμο ταυτοποίησης δύο εκφράσεων που χρησιμοποιείται από την Prolog, θα εξηγήσουμε τη χρησιμότητα της ταυτοποίησης με τη βοήθεια

ενός παραδείγματος.

Παράδειγμα 7: Έστω ο παρακάτω κανόνας:

```
competition('World Cup',Y) :-
   member(Y, [1930,1934,1938,1950,1954,1958,
              1962,1970,1974,1978,1982,1986,
              1990,1994,1998,2002,2006,2010]).
```

και η ερώτηση

```
?- competition(X,2010).
```

Το κατηγόρημα `competition` συσχετίζει μία αθλητική διοργάνωση με τα έτη διεξαγωγής της.

Ας εξετάσουμε το κατά πόσο ο παραπάνω κανόνας μπορεί να χρησιμεύσει στο να απαντηθεί η ερώτηση. Ο κανόνας αφορά μία συγκεκριμένη διοργάνωση (το `'World Cup'`) και επιβάλλει μία συνθήκη στη μεταβλητή `Y`, έτσι ώστε η τιμή της να αποτελεί χρονιά διεξαγωγής της διοργάνωσης. Από την άλλη η ερώτηση δεν αναφέρεται σε συγκεκριμένη διοργάνωση, αλλά αναζητά μία διοργάνωση `X` που να διεξάξεται το 2010. Ο κανόνας θα μπορούσε ειδικότερα να εφαρμοστεί για την περίπτωση όπου το `Y` είχε την τιμή 2010. Από την άλλη μία απάντηση που θα αφορούσε τη διοργάνωση `'World Cup'` είναι συναφής με την ερώτηση. Με άλλα λόγια το στιγμιότυπο του κανόνα που προκύπτει θέτοντας $\texttt{Y} = 2010$ μπορεί να δώσει απάντηση στο στιγμιότυπο της ερώτησης που προκύπτει θέτοντας $\texttt{X} = \texttt{'World Cup'}$.

Παρατηρούμε ότι η αντικατάσταση $\{\texttt{Y} = 2010, \texttt{X} = \texttt{'World Cup'}\}$ είναι ο πιο γενικός ταυτοποιητής της πρότασης στην ερώτηση και της κεφαλής του κανόνα. Συνεπώς η διαδικασία της ταυτοποίησης μας βοηθάει να συμπεράνουμε αν ένας κανόνας μπορεί να χρησιμεύσει στο να απαντήσουμε μία ερώτηση και υπό ποιες προϋποθέσεις (για ποιές τιμές των μεταβλητών που εμφανίζονται στον κανόνα και στην ερώτηση). ∎

Η Prolog βρίσκει έναν πιο γενικό ταυτοποιητή δύο εκφράσεων, με τον παρακάτω αλγόριθμο ταυτοποίησης:

Ο αλγόριθμος δέχεται ως είσοδο δύο εκφράσεις E_1 και E_2, οι οποίες περιέχουν τις μεταβλητές $V_1, \ldots, V_k$, όπου $k \geq 0$. Κατά τη διάρκεια της εκτέλεσής του, διατηρεί το σύνολο με τις τρέχουσες δεσμεύσεις για όλες τις παραπάνω μεταβλητές. Για ομοιομορφία, παριστάνουμε την έλλειψη δέσμευσης για μία μεταβλητή V_i ώς μία δέσμευση της μορφής $V_i = V_i$.

Αρχικά όλες οι μεταβλητές είναι αδέσμευτες, συνεπώς το σύνολο δεσμεύσεων είναι $\{V_1 = V_1, \ldots, V_k = V_k\}$.

Μετά από την αρχικοποίηση του συνόλου των δεσμεύσεων, ο αλγόριθμος εκτελεί ένα πλήθος από βήματα, σε καθένα από τα οποία είτε αντικαθιστά μία μεταβλητή

με έναν όρο ή τερματίζει επιστρέφοντας το αποτέλεσμα. Πιο συγκεκριμένα σε κάθε βήμα γίνονται τα παρακάτω:

- Έστω ότι στη αρχή του βήματος το σύνολο των δεσμεύσεων είναι $\{V_1 = b_1, \ldots, V_k = b_k\}$.
- Εξετάζονται οι εκφράσεις E_1 και E_2 από αριστερά προς τα δεξιά μέχρι να βρεθεί το πρώτο σημείο στο οποίο διαφέρουν συντακτικά.
- Αν οι εκφράσεις δεν διαφέρουν σε κανένα σημείο τότε είναι ίδιες και επιστρέφεται η λίστα με τις τρέχουσες δεσμεύσεις, η οποία αποτελεί έναν πιο γενικό ταυτοποιητή.
- Αλλιώς αν στο σημείο διαφοροποίησης ξεκινούν δύο όροι t_1 και t_2 τότε
  - Αν ο t_2 είναι μία μεταβλητή τότε αυτή αντικαθίσταται με t_1 στις εκφράσεις E_1 και E_2 και στους όρους $b_1, \ldots, b_k$.
  - Αλλιώς, αν ο t_1 είναι μία μεταβλητή αυτή αντικαθίσταται με t_2 στις εκφράσεις E_1 και E_2 και και στους όρους $b_1, \ldots, b_k$.
  - Αλλιώς (αν κανένας όρος δεν είναι μεταβλητή) επιστρέφεται αποτυχία.
- Αλλιώς (αν στο σήμείο που διαφέρουν οι εκφράσεις δεν ξεκινούν δύο όροι) επιστρέφεται αποτυχία.

Παράδειγμα 8: Έστω οι εκφράσεις E_1 = `p(X,Y,Z)` και E_2 = `p(f(Y),3,X)`. Η εκτέλεση του αλγόριθμου ταυτοποίησης με είσοδο αυτές τις εκφράσεις φαίνεται παρακάτω, όπου δίνεται η τρέχουσα μορφή των εκφράσεων και οι τρέχουσες δεσμεύσεις στην αρχή της εκτέλεσης καθε βήματος. Οι όροι που ξεκινούν στο αριστερότερο σημείο στο οποίο διαφέρουν οι εκφράσεις εμφανίζονται μέσα σε πλαίσιο.

βήμα	E_1	E_2	δεσμεύσεις
1	`p(`$\boxed{\texttt{X}}$`,Y,Z)`	`p(`$\boxed{\texttt{f(Y)}}$`,3,X)`	{X = X, Y = Y, Z = Z}
2	`p(f(Y),`$\boxed{\texttt{Y}}$`,Z)`	`p(f(Y),`$\boxed{\texttt{3}}$`,f(Y))`	{X = f(Y), Y = Y, Z = Z}
3	`p(f(3),3,`$\boxed{\texttt{Z}}$`)`	`p(f(3),3,`$\boxed{\texttt{f(3)}}$`)`	{X = f(3), Y = 3, Z = Z}
4	`p(f(3),3,f(3))`	`p(f(3),3,f(3))`	{X = f(3), Y = 3, Z = f(3)}

Στη αρχή του τέταρτου βήματος οι τρέχουσες μορφές των δύο εκφράσεων είναι ίσες. Συνεπώς το τρέχον σύνολο δεσμεύσεων {X = f(3), Y = 3, Z = f(3)} αποτελεί έναν πιό γενικό ταυτοποιητή των αρχικών εκφράσεων `p(X,Y,Z)` και `p(f(Y),3,X)`. ∎

Παράδειγμα 9: Παρακάτω φαίνεται η εκτέλεση του αλγόριθμου ταυτοποίησης για τις εκφράσεις E_1 = `p(X,1,Y)` και E_2 = `p(a,Y,X)`.

βήμα	E_1	E_2	δεσμεύσεις
1	`p(`$\boxed{\texttt{X}}$`,1,Y)`	`p(`$\boxed{\texttt{a}}$`,Y,X)`	{X = X, Y = Y}
2	`p(a,`$\boxed{\texttt{1}}$`,Y)`	`p(a,`$\boxed{\texttt{Y}}$`,a)`	{X = a, Y = Y}
3	`p(a,1,`$\boxed{\texttt{1}}$`)`	`p(a,1,`$\boxed{\texttt{a}}$`)`	{X = a, Y = 1}

Στη αρχή του τρίτου βήματος οι τρέχουσες μορφές των δύο εκφράσεων διαφέρουν σε ένα σημείο στο οποίο ξεκινούν δύο όροι κανένας από τους οποίους δεν είναι μεταβλητή. Συνεπώς οι αρχικές εκφράσεις `p(X,1,Y)` και `p(a,Y,X)` δεν είναι ταυτοποιήσιμες. ∎

Παράδειγμα 10: Παρακάτω φαίνεται η εκτέλεση του αλγόριθμου ταυτοποίησης για τις εκφράσεις $E_1 =$ `q(Z,4)` και $E_2 =$ `q(f(W))`.

βήμα	E_1	E_2	δεσμεύσεις
1	`q(`[`Z`]`,4)`	`q(`[`f(W)`]`)`	{`Z` = `Z`, `W` = `W`}
2	`q(f(W)`[`,`]`4)`	`q(f(W)`[`)`]	{`Z` = `f(W)`, `W` = `W`}

Στη αρχή του δεύτερου βήματος οι τρέχουσες μορφές των δύο εκφράσεων διαφέρουν σε ένα σημείο στο οποίο δεν ξεκινούν δύο όροι. Συνεπώς οι αρχικές εκφράσεις `q(Z,4)` και `q(f(W))` δεν είναι ταυτοποιήσιμες. ∎

Παράδειγμα 11: Παρακάτω φαίνεται η εκτέλεση του αλγόριθμου ταυτοποίησης για τις εκφράσεις $E_1 =$ `f(X)` και $E_2 =$ `f(f(X))`.

βήμα	E_1	E_2	δεσμεύσεις
1	`f(`[`X`]`)`	`f(`[`f(X)`]`)`	{`X` = `X`}
2	`f(f(`[`X`]`))`	`f(f(`[`f(X)`]`))`	{`X` = `f(X)`}
3	`f(f(f(`[`X`]`)))`	`f(f(f(`[`f(X)`]`)))`	{`X` = `f(f(X))`}
4	`f(f(f(f(`[`X`]`))))`	`f(f(f(f(`[`f(X)`]`))))`	{`X` = `f(f(f(X)))`}
5	`f(f(f(f(f(`[`X`]`)))))`	`f(f(f(f(f(`[`f(X)`]`)))))`	{`X` = `f(f(f(f(X))))`}
...	...	...	...

Παρότι, όπως είδαμε στο παράδειγμα 5, οι δύο εκφράσεις δεν είναι ταυτοποιήσιμες, ο αλγόριθμος θα εκτελέσει άπειρα βήματα, προσπαθώντας να τις ταυτοποιήσει και δεν θα επιστρέψει απάντηση. ∎

Το παραπάνω παράδειγμα δείχνει ότι ο αλγόριθμος ταυτοποίησης δεν λειτουργεί σωστά για όλες τις εκφράσεις.

Η ορθότητα του αλγορίθμου θα μπορούσε να εξασφαλιστεί μία απλή τροποποίηση: όταν πρόκειται να γίνει αντικατάσταση της μεταβλητής V από τον όρο t, θα πρέπει να γίνει έλεγχος για το αν η V εμφανίζεται στον t. Αν κάτι τέτοιο ισχύει (όπως στο παράδειγμα 11) ο αλγόριθμος θα πρέπει να μην κάνει αντικατάσταση αλλά να επιστρέφει άμεσα αρνητική απάντηση.

Η Prolog σκόπιμα αποφεύγει τον παραπάνω έλεγχο εμφάνισης επειδή αυτός επιβαρύνει το συνολικό χρόνο που απαιτείται για να απαντηθεί μία ερώτηση, καθώς ότι ο αλγόριθμος ταυτοποίησης εκτελείται πάρα πολύ συχνά. Θα πρέπει να σημειωθεί ότι το παραπάνω σενάριο δεν εμφανίζεται συχνά στα προγράμματα και η αποφυγή του επαφίεται στον προγραμματιστή.

Σε πολλές υλοποιήσεις της Prolog η έλλειψη δέσμευσης για μία μεταβλητή θεωρείται ισοδύναμη ως δέσμευση σε μία νέα μεταβλητή που δεν χρησιμοποιείται πουθενά αλλού (το οποίο δεν επιβάλλει κανέναν περιορισμό στην τιμή της μεταβλητής). Αυτό εξηγεί το παρακάτω αποτέλεσμα στην υλοποίηση C-Prolog:

```
?- f(A,B,C) = X.
A = _0
B = _1
C = _2
X = f(_0,_1,_2)
```

1.11 Διαδικασία απάντησης σε ερώτηση

Έχοντας περιγράψει την έννοια της ταυτοποίησης μπορούμε να περιγράψουμε το μηχανισμό με τον οποίο η Prolog απαντάει στις ερωτήσεις. Ξεκινάμε με ορισμένα απλά παραδείγματα, στα οποία παρουσιάζονται τα βασικά στοιχεία του μηχανισμού.

Παράδειγμα 12: Στο παράδειγμα 7 είδαμε ότι ο κανόνας με κεφαλή

`competition('World Cup',Y)`

μπορεί να χρησιμοποιηθεί για να απαντηθεί η ερώτηση

`?- competition(X,2010).`

Για να καταλήξουμε στο συμπέρασμα αυτό χρειάστηκε να ταυτοποιήσουμε τις δύο εκφράσεις. Ο πιο γενικός ταυτοποιητής τους είναι $\{$`X` = `'World Cup'`, `Y` = `2010`$\}$.

Ας υποθέσουμε τώρα ότι στην ερώτηση χρησιμοποιούμε τη μεταβλητή `Y` αντί της `X`. Τότε η προσπάθεια ταυτοποίησης των εκφράσεων `competition(Y,2010)` και `competition('World Cup',Y)` θα αποτύχει! Αυτό δεν είναι επιθυμητό αποτέλεσμα για δύο λόγους: πρώτον η χρήση διαφορετικής μεταβλητής στην ερώτηση δεν διαφοροποιεί την ερώτηση και δεύτερον η χρήση της ίδιας μεταβλητής στην ερώτηση και στον κανόνα είναι συμπτωματική. ∎

Η Prolog για να αποφύγει τέτοιου είδους σενάρια, πριν εκτελέσει τον αλγόριθμο ταυτοποίησης, μετονομάζει όλες τις μεταβλητές του κανόνα, χρησιμοποιώντας νέα ονόματα που δεν εμφανίζονται στην ερώτηση ούτε έχουν χρησιμοποιηθεί προηγουμένως από τη διαδικασία απάντησης της ερώτησης. Τα νέα ονόματα δημιουργούνται αυτόματα από μία διαδικασία που εξαρτάται από την υλοποίηση.

Στο πλαίσιο των σημειώσεων όταν απαιτείται να μετονομαστούν οι μεταβλητές ενός κανόνα, θα χρησιμοποιούμε ένα όνομα το οποίο προκύπτει από το αρχικό όνομα της μεταβλητής στον κανόνα με προσθήκη αριθμητικών ψηφίων στο τέλος του. Τονίζεται ότι αυτή η προσέγγιση υιοθετείται για λόγους αναγνωσιμότητας και δεν είναι απαραίτητα η ίδια που ακολουθείται στις υλοποιήσεις.

Η Prolog για να απαντήσει μία ερώτηση στην οποία καλείται να αποδείξει μία πρόταση-στόχο, προσπαθεί να δημιουργήσει μία ακολουθία από στόχους, αντικαθιστώντας ή διαγράφοντας κάθε φορά την πρώτη πρόταση στον τρέχοντα στόχο. Στη διάρκεια της αποδεικτική διαδικασίας κρατάει μία λίστα με τις μεταβλητές που εμφανίζονται στην ερώτηση μαζί με τις τρέχουσες δεσμεύσεις τους.

Παράδειγμα 13: Ας υποθέσουμε ότι έχουμε φορτώσει το πρόγραμμα

```
parent(tantalos,pelopas).                  %  1
parent(dioni,pelopas).                     %  2
parent(dioni,niovi).                       %  3
parent(tantalos,niovi).                    %  4
parent(pelopas,atreas).                    %  5
male(tantalos).                            %  6
male(pelopas).                             %  7
male(atreas).                              %  8
female(dioni).                             %  9
female(niovi).                             % 10
father(X,Y) :- parent(X,Y), male(X).       % 11
grandfather(X,Y) :- father(X,Z), parent(Z,Y).  % 12
```

και κάνουμε την ερώτηση

```
?- grandfather(A,B).
```

Η Prolog θα προσπαθήσει να αποδείξει την πρόταση-στόχο `grandfather(A,B)` απο τις προτάσεις που αποτελούν το πρόγραμμα. Για το σκοπό αυτό εξετάζει με τη σειρά τις προτάσεις του προγράμματος, μετονομάζοντας τις μεταβλητές τους, μεχρι να βρεί είτε ένα γεγονός που ταυτοποιείται με τη `grandfather(A,B)` ή έναν κανόνα η κεφαλή του οποίου ταυτοποιείται με την `grandfather(A,B)`.

Η ταυτοποίηση θα πετύχει όταν συναντήσει τον κανόνα 12, ο οποίος μετά την μετονομασία των μεταβλητών του γίνεται

```
grandfather(X1,Y1) :- father(X1,Z1), parent(Z1,Y1).
```

Η κεφαλή του παραπάνω κανόνα ταυτοποιήται με τη πρόταση `grandfather(A,B)` με πιο γενικό ταυτοποιητή $\{$`X1` $=$ `A`, `Y1` $=$ `B`$\}$.

Η πρόταση `grandfather(A,B)` αντικαθίσταται με έναν νέο στόχο

```
father(A,Z1), parent(Z1,B)
```

ο οποίος προκύπτει από το σώμα του κανόνα με εφαρμογή του πιο γενικού ταυτοποιητή. Στη συνέχεια η Prolog προσπαθεί να αποδείξει την πρώτη πρόταση του νέου στόχου (η Prolog όταν έχει να αποδείξει ένα σύνθετο στόχο, ξεκινάει πάντα από την πρώτη πρόταση του). Η πρόταση `father(A,Z1)` ταυτοποιείται με την κεφαλή του κανόνα `father(X2,Y2) :- parent(X2,Y2), male(X2).` (ο οποίος προέκυψε

μετονομάζοντας τον κανόνα 11) με πιο γενικό ταυτοποιητή {X2 = A, Y2 = Z1}.

Ακολούθως σχηματίζεται ένας νέος στόχος, με αντικατάσταση της πρότασης father(A,Z1) από τις προτάσεις parent(A,Z1), male(A), οι οποίες προκύπτουν από το σώμα του κανόνα με εφαρμογή του πιο γενικού ταυτοποιητή. Ο νέος στόχος που προσπαθεί να αποδείξει η Prolog είναι

```
parent(A,Z1), male(A), parent(Z1,B).
```

Η πρόταση parent(A,Z1) ταυτοποιείται με το γεγονός parent(tantalos,pelopas), με πιο γενικό ταυτοποιητή {A = tantalos, Z1 = pelopas}. Η πρόταση parent(A,Z1) διαγράφεται από το στόχο (αφού ταυτοποιήθηκε με γεγονός, το οποίο αληθεύει χωρίς προϋποθέσεις). Επίσης στις προτάσεις που παραμένουν στο στόχο η μεταβλητή Z1 αντικαθίσταται από το άτομο pelopas.

Οι τρέχουσες δεσμεύσεις των μεταβλητών της αρχικής ερώτησης γίνονται {A = tantalos, B = B}.

Ο νέος στόχος είναι male(tantalos), parent(pelopas,B). Η Prolog θα βρεί το γεγονός male(tantalos) στο πρόγραμμα θα διαγράψει την πρόταση από τον στόχο. Ο στόχος γίνεται parent(pelopas,B). Η μοναδική πρόταση στο στόχο ταυτοποιείται με το γεγονός parent(pelopas,atreas), με πιο γενικό ταυτοποιητή {B = atreas}. Οι τρέχουσες δεσμεύσεις των μεταβλητών της αρχικής ερώτησης γίνονται {A = tantalos, B = atreas}.

Σε αυτό το σημείο η Prolog διαπιστώνει ότι δεν έχει τίποτα άλλο να αποδείξει (καθώς ο στόχος είναι κενός) και επιστρέφει την απάντηση

```
A = tantalos
B = atreas
```

Τα βήματα που εκτελεί η Prolog για να απαντήσει την ερώτηση συνοψίζονται παρακάτω. Για κάθε βήμα αναγράφεται ο τρέχων στόχος, οι τρέχουσες δεσμεύσεις των μεταβλητών της ερώτησης, οι προτάσεις του προγράμματος που εξετάστηκαν, η πρόταση του προγράμματος (μετά τη μετονομασία των μεταβλητών) με την οποία έγινε η ταυτοποίηση (η οποία είναι η τελευταία που εξετάστηκε), και ο πιο γενικός ταυτοποιητής. Το EOP συμβολίζει το τέλος του προγράμματος. Ο αριθμός που δίνεται σε αγκύλες πριν από κάθε στόχο δείχνει τα επιτυχή βήματα που οδηγούν σε αυτόν ξεκινώντας από τον αρχικό στόχο. Οταν ο τρέχον στόχος γίνει κενός (συμβολίζεται με #) δεν υπάρχουν οι τρεις τελευταίες πληροφορίες και η απάντηση που εκτυπώνεται δίνεται μετά την παρεμβολή μια διαχωριστικής γραμμής.

```
[0] grandfather(A,B)
    {A = A, B = B}
    1 --> 12
    grandfather(X1,Y1) :- father(X1,Z1), parent(Z1,Y1).
    {X1 = A, Y1 = B}
```

```
[1] father(A,Z1), parent(Z1,B)
    {A = A, B = B}
    1 --> 11
    father(X2,Y2) :- parent(X2,Y2), male(X2).
    {X2 = A, Y2 = Z1}

[2] parent(A,Z1), male(A), parent(Z1,B)
    {A = A, B = B}
    1 --> 1
    parent(tantalos,pelopas)
    {A = tantalos, Z1 = pelopas}

[3] male(tantalos),parent(pelopas,B)
    {A=tantalos, B=B}
    1 --> 6
    male(tantalos)
    {}

[4] parent(pelopas,B)
    {A = tantalos, B = B}
    1 --> 5
    parent(pelopas,atreas)
    {B = atreas}

[5] #
    {A = tantalos, B = atreas}
------------------------------------------------------------
A = tantalos
B = atreas
```

∎

Στο προηγούμενο παράδειγμα η Prolog δημιούργησε μία ακολουθία στόχων που οδήγησε απευθείας σε επιτυχία. Ωστόσο αυτό δεν είναι ο γενικός κανόνας. Στη γενική περίπτωση η Prolog χρειάζεται να οπισθοδρομήσει, ακυρώνοντας κάποιες δεσμεύσεις μεταβλητών και δημιουργώντας νέες.

Παράδειγμα 14: Ας υποθέσουμε ότι έχουμε φορτώσει το πρόγραμμα του παραδείγματος 13 και κάνουμε την ερώτηση

```
?- father(F,niovi).
```

Η πρότσαση `father(F,niovi)` ταυτοποιείται με την κεφαλή του κανόνα

```
father(X1,Y1) :- parent(X1,Y1), male(X1).
```

(ο οποίος προέκυψε μετονομάζοντας τον κανόνα 11) με πιο γενικό ταυτοποιητή {`X1` = `F`, `Y1` = `niovi`}.

Ο νέος στόχος που προσπαθεί να αποδείξει η Prolog είναι `parent(F,niovi), male(F)`. Η πρόταση `parent(F,niovi)` ταυτοποιείται με το γεγονός `parent(dioni,niovi)`, με

πιο γενικό ταυτοποιητή {F = dioni}. Στο σημείο αυτό δεσμεύεται η μεταβλητή F που υπήρχε στην αρχική ερώτηση.

Ο νέος στόχος αποτελείται από την πρόταση male(dioni), η οποία δεν αποδεικνύεται από το πρόγραμμα, καθώς δεν ταυτοποιείται με κανένα γεγονός ή κεφαλή κανόνα του προγράμματος.

Στο σημείο αυτό η Prolog χρειάζεται να οπισθοδρομήσει. Αυτό σημαίνει ότι πρέπει να επανέλθει στην κατάσταση που βρισκόταν ένα βήμα πριν.

Ο στόχος γίνεται πάλι parent(F,niovi), male(F) και η δέσμευση F = dioni ακυρώνεται. Στη συνέχεια η Prolog προσπαθεί να ταυτοποιήσει ξανά το parent(F,niovi) μέ κάποιο γεγονός ή κεφαλή κανόνα, εξετάζοντας το πρόγραμμα από το σημείο που έγινε η προηγούμενη ταυτοποίηση και μετά. Με άλλα λόγια αναζητάει έναν εναλλακτικό τρόπο απόδειξης της πρότασης parent(F,niovi) από το πρόγραμμα. Η πρόταση parent(F,niovi) ταυτοποιείται με το γεγονός parent(tantalos,niovi), με πιο γενικό ταυτοποιητή {F = tantalos}. Στο σημείο αυτό δεσμεύεται η μεταβλητή F που υπήρχε στην αρχική ερώτηση σε μία νέα τιμή.

Ο νέος στόχος αποτελείται από την πρόταση male(tantalos), η οποία υπάρχει στο πρόγραμμα. Τελικά η Prolog σχηματίζει τον κενό στόχο και επιστρέφει την απάντηση

```
F = tantalos
```

Τα βήματα που εκτελεί η Prolog συνοψίζονται παρακάτω. Ο στόχος στον οποίο οπισθοδρομεί η Prolog είναι η πιο πρόσφατος στόχος με αρίθμηση κατά ένα μικρότερη σε σχέση με το στόχο που αποτυγχάνει.

```
[0] father(F,niovi)
    {F = F}
    1 --> 11
    father(X1,Y1) :- parent(X1,Y1), male(X1).
    {X1 = F, Y1 = niovi}

[1] parent(F,niovi), male(F)
    {F = F}
    1 --> 3
    parent(dioni,niovi)
    {F = dioni}

[2] male(dioni)
    {F = dioni}
    1 --> EOP
    failure - backtracking

[1] parent(F,niovi), male(F)
    {F = F}
    4 --> 4
    parent(tantalos,niovi)
    {F = tantalos}
```

```
[2] male(tantalos)
    {F = tantalos}
    1 --> 6
    male(tantalos)
    {}

[3] #
    {F = tantalos}
------------------------------------------------------------
F = tantalos
```


Η Prolog για να μπορεί να οπισθοδρομεί, χρειάζεται να θυμάται, όλες τις ενδιάμεσες καταστάσεις από τις οποίες πέρασε για να φτάσει από τον αρχικό στόχο στον τρέχοντα. Για το λόγο κάθε φορά που δημιουργεί ένα νέο στόχο, αποθηκεύει τον προηγούμενο σε μία στοίβα, μαζί με τη θέση μέσα στο πρόγραμμα της πρότασης με την οποία έγινε η ταυτοποίηση και τις τρέχουσες δεσμεύσεις των μεταβλητών. Κάθε φορά που η Prolog αποτυγχάνει να ταυτοποιήσει την πρώτη πρόταση του στόχου με κάποια πρόταση του προγράμματος, εξάγεται η προηγούμενη κατάσταση από την κορυφή της στοίβας και συνεχίζεται από αυτήν η διαδικασία.

Μπορούμε τώρα να περιγράψουμε πλήρως τον αλγόριθμο με τον οποίο η Prolog απαντάει μία ερώτηση. Για λόγους συντομίας, αν μία πρόταση του προγράμματος είναι γεγονός, θα καλούμε κεφαλή της πρότασης την ατομική πρόταση που αποτελεί το γεγονός.

Ο αλγόριθμος δέχεται ως είσοδο ένα στόχο G που αποτελείται από την πεπερασμένη ακολουθία ατομικών προτάσεων που σχηματίζουν την ερώτηση.

Όπως και αλγόριθμος ταυτοποίησης, ο αλγόριθμος που περιγράφουμε διατηρεί μία λίστα με τις τρέχουσες δεσμεύσεις για όλες τις μεταβλητές της ερώτησης εκτός από την ανώνυμη μεταβλητή. Αν οι μεταβλητές που εμφανίζονται στην ερώτηση (διαφορετικές της ανώνυμης) είναι οι $V_1, \ldots, V_k$ τότε το σύνολο των δεσμεύσεων αρχικά είναι $S = \{V_1 = V_1, \ldots, V_k = V_k\}$ $(k \geq 0)$, που δηλώνουν έμμεσα την έλλειψη δέσμευσης. Οι εμφανίσεις της ανώνυμης μεταβλητής στο στόχο G αντικαθίστανται από νέες μεταβλητές (μία διαφορετική για κάθε εμφάνιση). Ο αλγόριθμος χρησιμοποιεί μία βοηθητική μεταβλητή i που αρχικά έχει τιμή 0. Η στοίβα είναι αρχικά κενή.

Κάθε βήμα του αλγορίθμου εκτελεί τις παρακάτω ενέργειες:

- Έστω ότι στη αρχή του βήματος το σύνολο των δεσμεύσεων είναι $S = \{V_1 = b_1, \ldots, V_k = b_k\}$.
- Αν ο στόχος περιέχει τουλάχιστον μία πρόταση τότε
  - Επαναλαμβάνονται οι παρακάτω ενέργειες:
    - Αυξάνεται τη τιμή του i κατά ένα.
    - Σχηματίζεται μία παραλλαγή C της i-οστής πρότασης του προγράμματος με μετονομασία των μεταβλητών της.

* Ελέγχεται αν η πρώτη πρόταση στο στόχο G ταυτοποιείται με την κεφαλή της C.

Η επανάληψη σταματάει όταν πετύχει η ταυτοποίηση ή εξαντληθούν οι προτάσεις στο πρόγραμμα.

– Αν η πρώτη πρόταση στον G ταυτοποιήθηκε μέ κάποια πρόταση C και Θ είναι ο πιο γενικός ταυτοποιητής που επιστράφηκε από τον αλγόριθμο ταυτοποίησης τότε

* Τα G, i και S αντιγράφονται στην κορυφή της στοίβας.
* Η πρώτη πρόταση του G διαγράφεται από αυτόν.
* Αν η πρόταση C είναι κανόνας τότε οι ατομικές προτάσεις που αποτελούν το σώμα της C εισάγονται στην αρχή του G (με την ίδια σειρά εμφάνισης όπως στη C).
* Γίνεται αντικατάσταση των μεταβλητών του G σύμφωνα με τον ταυτοποιητή Θ.
* Ενημερώνεται το σύνολο δεσμεύσεων S, με αντικατάσταση των μεταβλητών που εμφανίζονται στα $b_1, \ldots, b_k$ σύμφωνα με τον ταυτοποιητή Θ.
* Το i παίρνει την τιμή 0.

– Αλλιώς (αν το το πρόγραμμα εξαντλήθηκε χωρίς να γίνει ταυτοποίηση) τότε

* Αν η στοίβα δεν είναι κενή τότε γίνεται ανάκτηση των τιμών των G, i και S από τη στοίβα (οπισθοδρόμηση).
* Αλλιώς εκτυπώνεται `no` και τερματίζει ο αλγόριθμος.

• Αλλιώς (αν ο στόχος είναι κενός)

– Αν το S είναι κενό τότε εκτυπώνεται `yes` και τερματίζει ο αλγόριθμος.

– Αλλιώς

* Εκτυπώνονται οι δεσμεύσεις που περιέχονται στο σύνολο S, μία σε κάθε γραμμή και αναμένεται η οδηγία από τον χρήστη.
* Αν ο χρήστης γράψει ; τότε γίνεται επαναληπτικά ανάκτηση των τιμών των G, i και S από τη στοίβα μέχρι να βρεθεί στοχος που περιέχει μεταβλητή (αναζήτηση άλλης απάντησης).
* Αλλιώς (αν ο χρήστης γράψει .) τότε εκτυπώνεται `yes` και τερματίζει ο αλγόριθμος.

Παράδειγμα 15: Έστω το παρακάτω πρόγραμμα:

```
parent(tantalos,pelopas).          %  1
parent(dioni,pelopas).             %  2
parent(dioni,niovi).               %  3
parent(tantalos,niovi).            %  4
parent(pelopas,atreas).            %  5
male(tantalos).                    %  6
male(pelopas).                     %  7
male(atreas).                      %  8
female(dioni).                     %  9
female(niovi).                     % 10
```

```
predecessor(X,Y) :- parent(X,Y).       % 11
predecessor(X,Y) :- parent(X,Z),       % 12
                    predecessor(Z,Y).
```

Αν κάνουμε την ερώτηση

```
?- predecessor(dioni,W),male(W).
```

η Prolog θα ακολουθήσει τα παρακάτω βήματα:

```
[0] predecessor(dioni,W),male(W)
    {W = W}
    1 --> 11
    predecessor(X1,Y1) :- parent(X1,Y1).
    {X1 = dioni, Y1 = W}

[1] parent(dioni,W),male(W)
    {W = W}
    1 --> 2
    parent(dioni,pelopas).
    {W = pelopas}

[2] male(pelopas)
    {W = pelopas}
    1 --> 7
    male(pelopas)
    {}

[3] #
    {W = pelopas}
-----------------------------------------------------------------
W = pelopas ;
-----------------------------------------------------------------
[2] male(pelopas)
[1] parent(dioni,W),male(W)
    {W = W}
    3 --> 3
    parent(dioni,niovi).
    {W = niovi}

[2] male(niovi)
    {W = niovi}
    1 --> EOP
    failure - baktracking

[1] parent(dioni,W),male(W)
    {W = W}
    4 --> EOP
    failure - baktracking
```

```
[0] predecessor(dioni,W),male(W)
    {W = W}
    12 --> 12
    predecessor(X2,Y2) :- parent(X2,Z2), predecessor(Z2,Y2).
    {X2 = dioni, Y2 = W}

[1] parent(dioni,Z2), predecessor(Z2,W),male(W)
    {W = W}
    1 --> 2
    parent(dioni,pelopas).
    {Z2 = pelopas}

[2] predecessor(pelopas,W),male(W)
    {W = W}
    1 --> 11
    predecessor(X3,Y3) :- parent(X3,Y3).
    {X3 = pelopas, Y3 = W}

[3] parent(pelopas,W),male(W)
    {W = W}
    1 --> 5
    parent(pelopas,atreas).
    {W = atreas}

[4] male(atreas)
    {W = atreas}
    1 --> 8
    male(atreas).
    {}

[5] #
    {W = atreas}
------------------------------------------------------------
W = atreas.
yes
------------------------------------------------------------
```

∎

Παράδειγμα 16: Έστω το παρακάτω πρόγραμμα Prolog, στο οποίο περιγράφεται ένα γράφημα και ορίζεται η έννοια του μονοπατιού. Το κατηγόρημα `egde` ορίζεται από ένα πλήθος γεγότων που καθορίζουν τις ακμές του γραφήματος. Το κατηγόρημα `color` ορίζεται επίσης από ένα πλήθος γεγονότων που καθορίζουν αν το χρώμα μία κορυφής είναι μπλε ή κόκκινο. Τέλος το κατηγόρημα `path(A,B,N)` ορίζεται αναδρομικά από ένα γεγονός και έναν κανόνα. Η πρόταση `path(A,B,N)` δηλώνει ότι υπάρχει κατευθυνόμενο μονοπάτι μεταξύ `A` και `B` μήκους `N`. Ο αριθμός n παριτάνεται από τον όρο $\underbrace{\texttt{s(s(...s(}}_{n}\texttt{0)...))}$.

```
edge(a,b).                                % 1
edge(b,c).                                % 2
edge(b,d).                                % 3
edge(c,d).                                % 4
edge(c,e).                                % 5
color(a,red).                             % 6
color(b,blue).                            % 7
color(c,blue).                            % 8
color(d,blue).                            % 9
color(e,red).                             % 10
path(A,A,0).                              % 11
path(A,B,s(N)) :- edge(A,X), path(X,B,N). % 12
```

Αν θέλουμε να βρούμε αν υπάρχει μονοπάτι μήκους τουλάχιστον δύο που να συνδέει κόμβους του ίδιου χρώματος κανουμε στην Prolog την ερώτηση

```
?- color(A,C),path(A,B,s(s(X))),color(B,C).
```

Τα βήματα που θα ακολουθήσει η Prolog για να απαντήσει στην ερώτηση συνοψίζονται παρακάτω.

```
[0] color(A,C),path(A,B,s(s(X))),color(B,C)
    {A = A, C = C, B = B, X = X}
    1 --> 6
    color(a,red).
    {A = a, C = red}

[1] path(a,B,s(s(X))),color(B,red)
    {A = a, C = red, B = B, X = X}
    1 --> 12
    path(A1,B1,s(N1)) :- edge(A1,X1), path(X1,B1,N1).
    {A1 = a, B1 = B, N1 = s(X)}

[2] edge(a,X1),path(X1,B,s(X)),color(B,red)
    {A = a, C = red, B = B, X = X}
    1 --> 1
    edge(a,b).
    {X1 = b}

[3] path(b,B,s(X)),color(B,red)
    {A = a, C = red, B = B, X = X}
    1 --> 12
    path(A2,B2,s(N2)) :- edge(A2,X2), path(X2,B2,N2).
    {A2 = b, B2 = B, N2 = X}

[4] edge(b,X2),path(X2,B,X),color(B,red)
    {A = a, C = red, B = B, X = X}
    1 --> 2
    edge(b,c).
    {X2 = c}
```

```
[5] path(c,B,X),color(B,red)
    {A = a, C = red, B = B, X = X}
    1 --> 11
    path(A3,A3,0).
    {A3 = c, B = c, X = 0}

[6] color(c,red)
    {A = a, C = red, B = c, X = 0}
    1 --> EOP
    failure - backtracking

[5] path(c,B,X),color(B,red)
    {A = a, C = red, B = B, X = X}
    12 --> 12
    path(A4,B4,s(N4)) :- edge(A4,X4), path(X4,B4,N4).
    {A4 = c, B4 = B, X = s(N4)}

[6] edge(c,X4),path(X4,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    1 --> 4
    edge(c,d).
    {X4 = d}

[7] path(d,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    1 --> 11
    path(A5,A5,0).
    {A5 = d, B = d, N4 = 0}

[8] color(d,red)
    {A = a, C = red, B = B, X = s(0)}
    1 --> EOP
    failure - backtracking

[7] path(d,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    12 --> 12
    path(A6,B6,s(N6)) :- edge(A6,X6), path(X6,B6,N6).
    {A6 = d, B6 = B, N4 = s(N6)}

[8] edge(d,X6),path(X6,B,N6),color(B,red)
    {A = a, C = red, B = B, X = s(s(N6))}
    1 --> EOP
    failure - backtracking

[7] path(d,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    EOP
```

```
    failure - backtracking

[6] edge(c,X4),path(X4,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    5 --> 5
    edge(c,e).
    {X4 = e}

[7] path(e,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    1 --> 11
    path(A7,A7,0).
    {A7 = e, B = e, N4 = 0}

[8] color(e,red)
    {A = a, C = red, B = e, X = s(0)}
    1 --> 10
    color(e,red).
    {}

[9] #
    {A = a, C = red, B = e, X = s(0)}
------------------------------------------------------------
A = a
C = red
B = e
X = s(0) ;
------------------------------------------------------------

[8] color(e,red)
[7] path(e,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    12 --> 12
    path(A8,B8,s(N8)) :- edge(A8,X8), path(X8,B8,N8).
    {A8 = e, B8 = B, N4 = s(N8)}

[8] edge(e,X8),path(X8,B,N8),color(B,red)
    {A = a, C = red, B = B, X = s(s(N8))}
    1 --> EOP
    failure - backtracking

[7] path(e,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    EOP
    failure - backtracking

[6] edge(c,X4),path(X4,B,N4),color(B,red)
    {A = a, C = red, B = B, X = s(N4)}
    6 --> EOP
```

```
    failure - backtracking

[5] path(c,B,X),color(B,red)
    {A = a, C = red, B = B, X = X}
    EOP
    failure - backtracking

[4] edge(b,X2),path(X2,B,X),color(B,red)
    {A = a, C = red, B = B, X = X}
    3 --> 3
    edge(b,d).
    {X2 = d}

[5] path(d,B,X),color(B,red)
    {A = a, C = red, B = B, X = X}
    1 --> 11
    path(A9,A9,0).
    {A9 = d, B = d, X = 0}

[6] color(d,red)
    {A = a, C = red, B = d, X = 0}
    1 --> EOP
    failure - backtracking

[5] path(d,B,X),color(B,red)
    {A = a, C = red, B = d, X = X}
    12 --> 12
    path(A10,B10,s(N10)) :- edge(A10,X10), path(X10,B10,N10).
    {A10 = d, B10 = B, X = s(N10)}

[6] edge(d,X10),path(X10,B,N10),color(B,red)
    {A = a, C = red, B = d, X = s(N10)}
    1 --> EOP
    failure - backtracking

[5] path(d,B,X),color(B,red)
    {A = a, C = red, B = d, X = X}
    EOP
    failure - backtracking

[4] edge(b,X2),path(X2,B,X),color(B,red)
    {A = a, C = red, B = B, X = X}
    4 --> EOP
    failure - backtracking

[3] path(b,B,s(X)),color(B,red)
    {A = a, C = red, B = B, X = X}
    EOP
    failure - backtracking
```

```
[2] edge(a,X1),path(X1,B,s(X)),color(B,red)
    {A = a, C = red, B = B, X = X}
    2 --> EOP
    failure - backtracking

[1] path(a,B,s(s(X))),color(B,red)
    {A = a, C = red, B = B, X = X}
    EOP
    failure - backtracking

[0] color(A,C),path(A,B,s(s(X))),color(B,C)
    {A = A, C = C, B = B, X = X}
    7 --> 7
    color(b,blue).
    {A = b, C = blue}

[1] path(b,B,s(s(X))),color(B,blue)
    {A = b, C = blue, B = B, X = X}
    1 --> 12
    path(A11,B11,s(N11)) :- edge(A11,X11), path(X11,B11,N11).
    {A11 = b, B11 = B, N11 = s(X)}

[2] edge(b,X11),path(X11,B,s(X)),color(B,blue)
    {A = b, C = blue, B = B, X = X}
    1 --> 2
    edge(b,c).
    {X11 = c}

[3] path(c,B,s(X)),color(B,blue)
    {A = b, C = blue, B = B, X = X}
    1 --> 12
    path(A12,B12,s(N12)) :- edge(A12,X12), path(X12,B12,N12).
    {A12 = c, B12 = B, N12 = X}

[4] edge(c,X12),path(X12,B,X),color(B,blue)
    {A = b, C = blue, B = B, X = X}
    1 --> 4
    edge(c,d).
    {X12 = d}

[5] path(d,B,X),color(B,blue)
    {A = b, C = blue, B = B, X = X}
    1 --> 11
    path(A13,A13,0).
    {A13 = d, B = d, X = 0}

[6] color(d,blue)
    {A = b, C = blue, B = d, X = 0}
```

```
    1 --> 9
    color(d,blue).
    {}

[7] #
    {A = b, C = blue, B = d, X = 0}
------------------------------------------------------------
A = b
C = blue
B = d
X = 0.
yes
------------------------------------------------------------
```

1.12 Ενσωματωμένη αριθμητική

Η υποστήριξη αριθμών και αριθμητικών πράξεων αποτελεί βασική απαίτηση από μία γλώσσα προγραμματισμού, ακόμη και αν αυτή δεν προορίζεται για αμιγώς αριθμητικές εφαρμογές.

Η Prolog υποστηρίζει ακέραιους και πραγματικούς αριθμούς και τους τελεστές `+`, `-`, `*`, `//` (ακέραια διαίρεση), `/` (πραγματική διαίρεση) και `mod` (υπόλοιπο διαίρεσης).

Για να αποτιμηθεί μία παράσταση που περιέχει αριθμητικούς τελεστές χρησιμοποιείται ο τελεστής `is`:

```
?- X is 3+5.
X = 8

?- X is 5//2.
X = 2

?- X is 5/2.
X = 2.5

?- X is 16 mod 3.
X = 1
```

Συνήθως αριστερά του τελεστή `is` τοποθετείται μία μεταβλητή χωρίς δέσμευση, η οποία, μετά την αποτίμηση της αριθμητικής παράστασης που βρίσκεται δεξια του `is`, δεσμεύεται στην τιμή της παράστασης. Ωστόσο η Prolog επιτρέπει αριστερά του `is` να υπάρχει οποιοσδήποτε όρος. Μετά την αποτίμηση της παράστασης γίνεται ταυτοποίηση του όρου με την τιμή που προκύπτει από την αποτίμηση. Είναι εύκολο να διαπιστωθεί ότι η ταυτοποίηση επιτυγχάνει μόνο όταν ο όρος αριστερά του `is` είναι αριθμητική τιμή ίση με την τιμή της παράστασης, μεταβλητή που έχει δεσμευτεί σε τιμή ίση με την τιμή της παράστασης ή μεταβλητή χωρίς δέσμευση.

```
?- 5 is 2+3.
yes

?- 7 is 4*2.
no

?- a is 3-1.
no

?- 2+2 is 3+1.
no
```

Στην τελευταία ερώτηση η απάντηση είναι **no** επειδή το **is** προκαλεί αποτίμηση μόνο στην παράσταση που βρίσκεται δεξιά του.

Οι παραστάσεις δεξιά του **is** επιτρέπεται να περιέχουν περισσότερους του ενός αριθμητικούς τελεστές. Ο τελεστής **mod** έχει τη μέγιστη προτεραιότητα, οι *, // και / έχουν μεσαία προτεραιότητα και οι + και - έχουν την ελάχιστη προτεραιότητα. Τελεστές με ίδια προτεραιότητα προσεταιρίζονται από αριστερά προς τα δεξιά. Μπορούμε να χρησιμοποιούμε παρενθέσεις για να καθορίζουμε τη σειρά των πράξεων.

```
?- X is  20//5 mod 3.
X = 10

?- X is 10-2*3.
X = 4

?- X is 100//10//5.
X = 2

?- X is 100//(10//5).
X = 50
```

Οι απλοί όροι που εμφανίζονται στην παράσταση δεξιά του **is** θα πρέπει να είναι αριθμοί ή μεταβλητές που έχουν δεσμευτεί σε αριθμητικές τιμές πριν από την αποτίμηση. Αν στα δεξιά του **is** περιέχονται άτομα ή σύνθετοι όροι που σχηματίζονται απο συναρτησιακά σύμβολα που δεν είναι τελεστές αριθμητικών πράξεων ή μεταβλητές που δεν είναι δεσμευμένες σε αριθμητική τιμή, τότε η αποτίμηση της παράστασης δεν είναι δυνατή και εμφανίζεται ένα μήνημα σφάλματος (το οποίο εξαρτάται από την υλοποίηση).

```
?- X is a+1.
! Error in arithmetic expression: a is not a number

?- X is f(0).
! Error in arithmetic expression: f is not an arithmetic operator

?- X is Y+1.
! Error in arithmetic expression: not a number
```

```
?- Y = 2, X is Y+1.
Y = 2
X = 3
```

Τονίζεται ότι = κάνει ταυτοποίηση ακόμη και στην περίπτωση που κάποιο όρισμά του είναι αριθμητική παράσταση και σε καμία περίπτωση δεν προκαλεί αποτίμηση.

```
?- X = 3+5.
X = 3+5

?- 8 = 3+5.
no

?- 2*4 = 3+5.
no

?- A+B = 3+5.
A = 3
B = 5
```

Πριν εκτελεστεί ο αλγόριθμος ταυτοποίησης, οι παραστάσεις μετατρέπονται σε όρους με τη συνήθη σύνταξη, οι οποίοι προκύπτουν χρησιμοποιώντας τους αριθμητικούς τελεστές ως συναρτησιακά σύμβολα. Στην πραγματικότητα η δυνατότητα γραφής ενός τελεστή ανάμεσα στα ορίσματά του είναι μόνο μία συντακτική διευκόλυνση. Οι αριθμητικές παραστάσεις στην Prolog εσωτερικά μετατρέπονται σε σύνθετο όρο. Για παράδειγμα η παράσταση `3+5` εσωτερικά παριστάνεται ως `+(3,5)`. Επίσης η παράσταση `3*5*2-25 mod 4` είναι συντακτική παραλλαγή του `-(*(*(3,5),2),mod(25,4))`.

```
?- +(3,5) = 3+5.
yes

?- -(*(*(3,5),2),mod(25,4)) = 3*5*2-25 mod 4.
yes
```

Μπορούμε να συγκρίνουμε αριθμητικές παραστάσεις με τους τελεστές `=:=` (ίσο), `<`, `>`, `=<`, `>=` και `=\=` (άνισο). Οι τελεστές σύγκρισης αποτιμούν και τους δύο όρους που παίρνουν ως ορίσματα, οι οποίοι θα πρέπει να είναι αριθμητικές παραστάσεις. Αν στους όρους αυτούς εμφανίζονται μεταβλητές θα πρέπει να έχουν δεσμευτεί σε αριθμητικές τιμές:

```
?- 2*4 =:= 3+5.
yes

?- X =:= 3+5.
! Error in arithmetic expression: not a number

?- X =< X+1.
! Error in arithmetic expression: not a number

?- X is 2+4, X =< X+1.
X = 6
```

1.13 Χρήση όρων για αναπαράσταση αριθμών

Μπορούμε να χρησιμοποιήσουμε σύνθετους όρους για να παραστήσουμε αριθμούς. Για παράδειγμα, ο φυσικός αριθμός n μπορεί να παρασταθεί ως $\underbrace{\texttt{s(s(...s}}_{n}\texttt{(0)...))}$ (όπως κάναμε στο παράδειγμα 16). Στην αναπαράσταση αυτή ο αριθμός 3 γράφεται `s(s(s(0)))`.

Μπορούμε να ορίσουμε το κατηγόρημα `nat` το οποίο παίρνει ένα όρισμα και αληθεύει όταν η τιμή του ορίσματος είναι όρος που παριστάνει φυσικό αριθμό στην αναπαράσταση που περιγράψαμε:

```
nat(0).
nat(s(X)) :- nat(X).
```

Παράδειγμα ερωτήσεων που χρησιμοποιούν το `nat`:

```
?- nat(s(s(s(0)))).
yes

?- nat(3).
no
```

Η παραπάνω αναπαράσταση είναι δυσανάγνωστη και δε βοηθάει στη δημιουργία αποδοτικών προγραμμάτων. Από την άλλη πλευρά, το πλεονέκτημα που έχει είναι ότι η χρήση της επιτρέπει στην Prolog να απαντήσει σε τύπους ερωτήσεων, για τις οποίες η ενσωματωμένη αριθμητική θα οδηγούσε σε σφάλμα, λόγω της ύπαρξης του `is` (αυτό φαίνεται στο παράδειγμα 17 παρακάτω).

Τονίζεται ότι μπορεί κανείς να επεκτείνει την παραπάνω αναπαράσταση και για αρνητικούς αριθμούς. Επίσης είναι δυνατές και άλλες εναλλακτικές αναπαραστάσεις αριθμών με χρήση όρων.

1.14 Συναρτήσεις

Αν f είναι μία συνάρτηση με n ορίσματα τότε μπορούμε να ορίσουμε ένα αντίστοιχο κατηγόρημα p με $n+1$ ορίσματα τέτοιο ώστε το $p(x_1, x_2, \ldots, x_n, x_{n+1})$ να αληθεύει αν και μόνο αν $x_{n+1} = f(x_1, x_2, \ldots, x_n)$.

Η Prolog δεν υποστηρίζει άμεσα συναρτήσεις. Αν θέλουμε να ορίσουμε μία συνάρτηση f τότε θα πρέπει να ορίσουμε το αντίστοιχό της κατηγόρημα p. Για να υπολογίσουμε την τιμή $f(x_1, x_2, \ldots, x_n)$, κανουμε μία ερώτηση της μορφής

`?-` $p(x_1, x_2, \ldots, x_n$`,Y).`

και η ζητούμενη τιμή είναι η τιμή της μεταβλητής `Y` στην απάντηση.

Σημειώνουμε ότι έχοντας ορίσει το κατηγόρημα p, μπορούμε να κάνουμε και ερωτήσεις, οι οποίες θα περιέχουν μεταβλητές χωρίς δέσμευση σε θέσεις διαφορετικές της τελευταίας, ενδεχομένως και σε περισσότερες από μία. Το αν η Prolog μπορεί να απαντήσει σε αυτές τις ερωτήσεις, εξαρτάται από το πως υλοποιείται το κατηγόρημα.

Παράδειγμα 17: πρόσθεση ακεραίων.

Ας υποθέσουμε ότι θέλουμε να ορίσουμε στην Prolog τη συνάρτηση $sum(x, y) = x + y$. Για το σκοπό αυτό, θα ορίσουμε ένα κατηγόρημα `sum(X,Y,Z)`, το οποίο θα αληθεύει αν η τιμή της `Z` είναι το άθροισμα των τιμών των `X` και `Y`.

Υλοποιούμε πρώτα το κατηγόρημα `sum` χωρίς χρήση της ενσωματωμένης αριθμητικής παριστάνοντας τους αριθμούς ως σύνθετους όρους που σχηματίζονται με το συναρτησιακό σύμβολο `s` (για πληρότητα επαναλαμβάνουμε τον ορισμό του `nat`):

```
nat(0).                          % 1
nat(s(X)) :- nat(X).             % 2
sum(X,0,X) :- nat(X).            % 3
sum(X,s(Y),s(Z)) :- sum(X,Y,Z). % 4
```

Για να υπολογίσουμε το άθροισμα των αριθμών 3 και 2 γράφουμε την ερώτηση:

```
?- sum(s(s(s(0))),s(s(0)),S).
S = s(s(s(s(0)))))
```

Θα μπορούσαμε να σχηματίσουμε μία ερώτηση στην οποία θα είχαμε ως δεύτερο όρισμα μεταβλητή:

```
?- sum(s(0),D,s(s(s(0)))).
D = s(s(0))
```

Με την παραπάνω ερώτηση στην ουσία κάνουμε αφαίρεση, στην οποία ο μειωτέος είναι το τρίτο όρισμα και ο αφαιρετέος το πρώτο όρισμα. Θα μπορούσαμε να κάνουμε και την παρακάτω ερώτηση στην οποία χρησιμοποιούμε δύο μεταβλητές:

```
?- sum(A,B,s(s(0))).
```

Η παραπάνω ερώτηση ζητάει από την Prolog να βρεί δύο αριθμούς `A` και `B` με άθροισμα 3. Η Prolog θα απαντήσει σωστά και σε αυτή την ερώτηση. Η δυνατότητα αυτή οφείλεται στο ότι το παραπάνω πρόγραμμα δεν περιγράφει απλά το πώς υπολογιζεται το άθροισμα δύο αριθμών, αλλά διατυπώνει τις ικανές και αναγκαίες συνθήκες που πρέπει να πληρούν τρεις αριθμοί, έτσι ώστε ο τρίτος να αποτελεί το άθροισμα των δύο πρώτων, χωρίς να καθορίζεται ποιές είναι οι είσοδοι και ποια είναι η έξοδος. Τα βήματα που θα εκτελέσει η Prolog για να απαντήσει σε αυτή την ερώτηση δίνονται παρακάτω. Υπενθυμίζεται ότι όταν δώσουμε ; μετά από απάντηση, η Prolog εξάγει στόχους από τη στοίβα μέχρι να βρεί κάποιον που περιέχει μεταβλητή.

```
[0] sum(A,B,s(s(0)))
    {A = A, B = B}
    1 --> 3
    sum(X1,0,X1) :- nat(X1).
    {A = s(s(0)), B = 0, X1 = s(s(0))}
```

```
[1] nat(s(s(s(0))))
    {A = s(s(s(0))), B = 0}
    1 --> 2
    nat(s(X2)) :- nat(X2).
    {X2 = s(s(0))}

[2] nat(s(s(0)))
    {A = s(s(s(0))), B = 0}
    1 --> 2
    nat(s(X3)) :- nat(X3).
    {X3 = s(0)}

[3] nat(s(0))
    {A = s(s(s(0))), B = 0}
    1 --> 2
    nat(s(X4)) :- nat(X4).
    {X4 = 0}

[4] nat(0)
    {A = s(s(s(0))), B = 0}
    1 --> 1
    nat(0).
    {}

[5] #
    {A = s(s(s(0))), B = 0}
------------------------------------------------------------
A = s(s(s(0)))
B = 0 ;
------------------------------------------------------------
[4] nat(0)
[3] nat(s(0))
[2] nat(s(s(0)))
[1] nat(s(s(s(0))))
[0] sum(A,B,s(s(s(0))))
    {A = A, B = B}
    4 --> 4
    sum(X5,s(Y5),s(Z5)) :- sum(X5,Y5,Z5).
    {X5 = A, B = s(Y5), Z5 = s(s(0))}

[1] sum(A,Y5,s(s(0)))
    {A = A, B = s(Y5)}
    1 --> 3
    sum(X6,0,X6) :- nat(X6).
    {A = s(s(0)), Y5 = 0, X6 = s(s(0))}
```

```
[2] nat(s(s(0)))
    {A = s(s(0)), B = s(0)}
    1 --> 2
    nat(s(X7)) :- nat(X7).
    {X7 = s(0)}

[3] nat(s(0))
    {A = s(s(0)), B = s(0)}
    1 --> 2
    nat(s(X8)) :- nat(X8).
    {X8 = 0}

[4] nat(0)
    {A = s(s(0)), B = s(0)}
    1 --> 1
    nat(0).
    {}

[5] #
    {A = s(s(0)), B = s(0)}
------------------------------------------------------------
A = s(s(0))
B = s(0) ;
------------------------------------------------------------
[4] nat(0)
[3] nat(s(0))
[2] nat(s(s(0)))
[1] sum(A,Y5,s(s(0)))
    {A = A, B = s(Y5)}
    4 --> 4
    sum(X9,s(Y9),s(Z9)) :- sum(X9,Y9,Z9).
    {X9 = A, Y5 = s(Y9), Z9 = s(0)}

[2] sum(A,Y9,s(0))
    {A = A, B = s(s(Y9))}
    1 --> 3
    sum(X10,0,X10) :- nat(X10).
    {A = s(0), Y9 = 0, X10 = s(0)}

[3] nat(s(0))
    {A = s(0), B = s(s(0))}
    1 --> 2
    nat(s(X11)) :- nat(X11).
    {X11 = 0}
```

```
[4] nat(0)
    {A = s(0), B = s(s(0))}
    1 --> 1
    nat(0).
    {}

[5] #
    {A = s(0), B = s(s(0))}
------------------------------------------------------------
A = s(0)
B = s(s(0)) ;
------------------------------------------------------------
[4] nat(0)
[3] nat(s(0))
[2] sum(A,Y9,s(0))
    {A = A, B = s(s(Y9))}
    4 --> 4
    sum(X12,s(Y12),s(Z12)) :- sum(X12,Y12,Z12).
    {X12 = A, Y9 = s(Y12), Z12 = 0}

[3] sum(A,Y12,0)
    {A = A, B = s(s(s(Y12)))}
    1 --> 3
    sum(X13,0,X13) :- nat(X13).
    {A = 0, Y12 = 0, X13 = 0}

[4] nat(0)
    {A = 0, B = s(s(s(0)))}
    1 --> 1
    nat(0).
    {}

[5] #
    {A = 0, B = s(s(s(0)))}
------------------------------------------------------------
A = 0
B = s(s(s(0))) ;
------------------------------------------------------------
[4] nat(0)
[3] sum(A,Y12,0)
    {A = A, B = s(s(s(Y12)))}
    4 --> EOP
    failure - backtracking

[2] sum(A,Y9,s(0))
    {A = A, B = s(s(Y9))}
    EOP
    failure - backtracking
```

```
[1] sum(A,Y5,s(s(0)))
    {A = A, B = s(Y5)}
    EOP
    failure - backtracking

[0] sum(A,B,s(s(s(0))))
    {A = A, B = B}
    EOP
    failure
-----------------------------------------------------------------
no
```

Ορίζουμε στη συνέχεια το κατηγόρημα `sum10` για υλοποίηση της πρόσθεσης με χρήση της ενσωματωμένης αριθμητικής της Prolog.

```
sum10(X,Y,Z) :- Z is X+Y.
```

Για να υπολογίσουμε το άθροισμα των αριθμών 3 και 2 γράφουμε την ερώτηση:

```
?- sum10(3,2,S).
S = 5
```

Η παραπάνω ερώτηση χρειάζεται λιγότερα βήματα για να απαντηθεί, ενώ η απάντηση είναι σε πιο αναγνώσιμη μορφή. Αν ωστόσο κάνουμε την ερώτηση

```
?- sum10(A,B,3).
```

η Prolog θα δημιουργήσει το στόχο `3 is A+B` ο οποίος θα προκαλέσει σφάλμα χρόνου εκτέλεσης, καθώς η παράσταση δεξιά του `is` περιέχει μεταβλητές χωρίς δέσμευση. Βλέπουμε ότι η χρήση της ενσωματωμένης αριθμητικής της Prolog στον ορισμό κατηγορημάτων μειώνει τη λειτουργικότητά τους. Ωστόσο για λόγους επίδοσης και αναγνωσιμότητας θα την επιλέξουμε στη συνέχεια για την υλοποίηση αριθμητικών συναρτήσεων.

1.15 Υλοποίηση αριθμητικών συναρτήσεων με αναδρομή

Στη συνέχεια θα ορίσουμε αναδρομικά κατηγορήματα που υλοποιούν αριθμητικές συναρτήσεις. Τα κατηγορήματα αυτά θα έχουν ένα όρισμα περισσότερο σε σύγκριση με τις συναρτήσεις που υλοποιούν, το οποίο θα είναι πάντοτε το τελευταίο. Όπως έχουμε ήδη αναφέρει, αν το κατηγόρημα p υλοποιεί μία συνάρτηση f τότε θα πρέπει η απάντηση στην ερώτηση

?- $p(x_1, x_2, \ldots, x_n$,Y)

να δίνει ως τιμή της μεταβλητής `Y` το $f(x_1, x_2, \ldots, x_n)$.

Αν η n-αδα τιμών $(x_1, x_2, \ldots, x_n)$ δεν ανήκει στο πεδίο ορισμού της f μία απόλυτα σωστή υλοποίηση του p θα πρέπει να επιστρέφει την απάντηση `no` στην παραπάνω ερώτηση. Αυτό για παράδειγμα συμβαίνει αν η f ορίζεται μόνο για θετικούς ακέραιους και κάποιο από

τα x_i είναι αρνητικός αριθμός, άτομο ή σύνθετος όρος. Στους ορισμούς των κατηγορημάτων που θα περιγράψουμε στη συνέχεια χαλαρώνουμε την παραπάνω συνθήκη: αν n-αδα $(x_1, x_2, \ldots, x_n)$ δεν ανήκει στο πεδίο ορισμού της f, τότε απαιτούμε να μην επιστρέφεται τιμή για τη μεταβλητή `Y`, χωρίς απαραίτητα να επιστρέφεται η απάντηση `no`. Αν για παράδειγμα κάποιο όρισμα δεν είναι αριθμός τότε μπορεί να διακοπεί η διαδικασία υπολογισμού με μήνυμα σφάλματος, ενώ αν είναι αριθμητική τιμή που δεν ανήκει στο πεδίο ορισμού, μπορεί το πρόγραμμα να πέφτει σε άπειρο βρόχο, που θα προκαλέσει υπερχείλιση στοίβας.

Επίσης, μία απόλυτα ορθή υλοποίηση του του p, θα έπρεπε σε περίπτωση που ζητήσουμε δεύτερη τιμή για τη συνάρτηση (γράφοντας ; μετά την πρώτη απάντηση) να επιστρέφει `no`. Και εδώ χαλαρώνουμε την απαίτησή μας επιτρέποντας στην Prolog είτε να πέσει σε άπειρο βρόχο είτε να επιστρέψει τη σωστή λύση περισσότερες από μία φορές. Αργότερα όταν θα έχουμε περιγράψει ορισμένα προκαθορισμένα κατηγορήματα της Prolog, όπως επίσης και τον έλεγχο της οπισθοδρόμησης, θα επανέλθουμε για να δούμε πως μπορούν τα κατηγορήματα να οριστούν με απόλυτα ορθό τρόπο.

Παράδειγμα 18: παραγοντικό.

Για να ορίσουμε την έννοια του παραγοντικού θα πρέπει να ορίσουμε ένα κατηγόρημα `fact(N,F)` το οποίο θα αληθεύει αν και μόνο αν η τιμή της μεταβλητής `F` είναι το παραγοντικό της τιμής της `N`. Το κατηγόρημα αυτό μπορεί να οριστεί με βάση τις δύο παρακάτω προτάσεις: (α) αν το N είναι 0 και το F είναι 1 τότε το F είναι το παραγοντικό του N και (β) αν το N είναι μεγαλυτερο από το 0, το K έχει τιμή $N-1$, το παραγοντικό του K είναι G και το F έχει τιμή $N \cdot G$ τότε το F είναι το παραγοντικό του N. Οι παραπάνω προτάσεις διατυπώνονται εύκολα σε Prolog.

```
fact(N,F) :- N = 0,
             F = 1.
fact(N,F) :- N > 0,
             K is N-1,
             fact(K,G),
             F is N*G.
```

Παρατηρούμε ότι οι δύο προτάσεις που αποτελούν το σώμα του πρώτου κανόνα προσδιορίζουν με χρήση του τελεστή `=` τις τιμές των μεταβλητών `N` και `F`. Συνήθως στην Prolog αντί να γράφουμε τέτοιες προτάσεις, κάνουμε απ' ευθείας αντικατάσταση των μεταβλητών με τις τιμές τους. Αντί του παραπάνω ορισμού για το `fact` θα μπορούσαμε να γράψουμε:

```
fact2(0,1).
fact2(N,F) :- N > 0,
              K is N-1,
              fact2(K,G),
              F is N*G.
```

Ο πρώτος κανόνας στον ορισμό του `fact` αντικαταστάθηκε από ένα γεγονός το οποίο σε φυσική γλώσσα λέει ότι το παραγοντικό του 0 είναι το 1, που είναι ισοδύναμο με την πρόταση (α).

Για να υπολογίσουμε το 3! κάνουμε την ερώτηση

```
?- fact2(3,X).
```

για την απάντηση της οποίας η Prolog εκτελεί τα παρακάτω βήματα:

```
 [0] fact2(3,X)
     {X = X}
     1 --> 2
     fact2(N1,F1) :- N1 > 0, K1 is N1-1, fact2(K1,G1), F1 is N1*G1.
     {N1 = 3, F1 = X}

 [1] 3 > 0, K1 is 3-1, fact2(K1,G1), X is 3*G1
     {X = X}
     > : build-in
     {}

 [2] K1 is 3-1, fact2(K1,G1), X is 3*G1
     {X = X}
     is : build-in
     {K1 = 2}

 [3] fact2(2,G1), X is 3*G1
     {X = X}
     1 --> 2
     fact2(N2,F2) :- N2 > 0, K2 is N2-1, fact2(K2,G2), F2 is N2*G2.
     {N2 = 2, F2 = G1}

 [4] 2 > 0, K2 is 2-1, fact2(K2,G2), G1 is 2*G2, X is 3*G1
     {X = X}
     > : build-in
     {}

 [5] K2 is 2-1, fact2(K2,G2), G1 is 2*G2, X is 3*G1
     {X = X}
     is : build-in
     {K2 = 1}

 [6] fact2(1,G2), G1 is 2*G2, X is 3*G1
     {X = X}
     1 --> 2
     fact2(N3,F3) :- N3 > 0, K3 is N3-1, fact2(K3,G3), F3 is N3*G3.
     {N3 = 1, F3 = G2}

 [7] 1 > 0, K3 is 1-1, fact2(K3,G3), G2 is 1*G3, G1 is 2*G2, X is 3*G1
     {X = X}
     > : build-in
     {}
```

```
 [8] K3 is 1-1, fact2(K3,G3), G2 is 1*G3, G1 is 2*G2, X is 3*G1
     {X = X}
     is : build-in
     {K3 = 0}

 [9] fact2(0,G3), G2 is 1*G3, G1 is 2*G2, X is 3*G1
     {X = X}
     1 --> 1
     fact2(0,1).
     {G3 = 1}

[10] G2 is 1*1, G1 is 2*G2, X is 3*G1
     {X = X}
     is : build-in
     {G2 = 1}

[11] G1 is 2*1, X is 3*G1
     {X = X}
     is : build-in
     {G1 = 2}

[12] X is 3*2
     {X = X}
     is : build-in
     {X = 6}

[13] #
     {X = 6}
------------------------------------------------------------
X = 6
```

Όπως φαίνεται και από τα βήματα της παραπάνω διαδικασίας, αν χρησιμοποιήσουμε το `fact2` (ή το `fact`) για να υπολογίσουμε το παραγοντικό ενός μη αρνητικού αριθμού, τότε ο έλεγχος `N > 0` θα αληθεύει πάντα μέχρι και τη στιγμή που θα εκτυπωθεί η τιμή του παραγοντικού ως απάντηση. Στη παρακάτω υλοποίηση ο έλεγχος για το την τιμή του `N` έχει παραληφθεί:

```
fact3(0,1).
fact3(N,F) :- K is N-1,
              fact3(K,G),
              F is N*G.
```

Το `fact3` όταν χρησιμοποιηθεί σε ερώτηση με το πρώτο όρισμα να έχει μή αρνητική ακέραια τιμή, τότε έχει ακριβώς την ίδια συμπεριφορά με το `fact2`. Τα δύο κατηγορήματα έχουν διαφορετική συμπεριφορά όταν το πρώτο όρισμα είναι αρνητικός αριθμός ή όταν ζητήσουμε δεύτερη απάντηση γράφοντας `;`. Αν προσπαθήσουμε να χρησιμοποιήσουμε το δεύτερο όρισμα ως είσοδο για να υπολογίσουμε την αντίστροφη συνάρτηση, δηλαδή για να βρούμε το αριθμό το παραγοντικό του οποίου ισούται με μία δεδομένη τιμή, τότε θα προκύψει σφάλμα χρόνου εκτέλεσης. Αυτό οφείλεται στη χρήση του `is` στον αναδρομικό κανόνα το οποίο για να λειτουργήσει σωστά απαιτεί η μεταβλητή `N` να έχει δεσμευτεί σε

αριθμητική τιμή. Μπορούμε να χρησιμοποιήσουμε το `fact3` (ή κάποια από τις παραλλαγές του) για να επαληθεύσουμε την τιμή του παραγοντικού:

```
fact3(5,120).
yes
```

∎

Παράδειγμα 19: υπολογισμός δύναμης.

Η Prolog δεν παρέχει κάποιον τελεστή για ύψωση σε δύναμη. Για τον υπολογισμό μίας μή αρνητικής ακέραιας δύναμης ενός αριθμού, θα ορίσουμε ένα κατηγόρημα `power(N,K,P)` το οποίο θα αληθεύει αν η τιμή της μεταβλητής `P` ισούται με την τιμή της `N` υψώμένη στην τιμή της `K` (η οποία θα υποθέτουμε ότι είναι μή αρνητική). Το `power` μπορεί να οριστεί με βάση τις παρακάτω προτάσεις: (α) το N^0 ισούται με 1, (β) αν το K είναι θετικός και άρτιος ακέραιος, το L ισούται με το $\lfloor \frac{K}{2} \rfloor$, το R ισούται με το N^L και το P ισούται με R^2, τότε το N^K ισούται με P και (γ) αν το K είναι θετικός και περιττός ακέραιος, το L ισούται με το $\lfloor \frac{K}{2} \rfloor$, το R ισούται με το N^L και το P ισούται με $R^2 \cdot N$, τότε το N^K ισούται με P. Οι παραπάνω προτάσεις μπορούν να γραφτούν σε Prolog με τον παρακάτω τρόπο:

```
power(N,0,1).
power(N,K,P) :- K mod 2 =:= 0,
                L is K//2,
                power(N,L,R),
                P is R*R.
power(N,K,P) :- K mod 2 =:= 1,
                L is K//2,
                power(N,L,R),
                P is R*R*N.
```

Έχουμε παραλείψει από τους αναδρομικούς κανόνες τον έλεγχο `K>0`. Αυτό έχει ώς παρενέργεια το να μας επιστρέφει Prolog απεριόριστες φορές την ίδια (σωστή) απάντηση στην περίπτωση που της ζητήσουμε να βρεί και άλλες λύσεις.

Για παράδειγμα, ο υπολογισμός του 3^5 γίνεται με τον παρακάτω τρόπο:

```
 [0] power(3,5,X)
     {X = X}
     1 --> 2
     power(N1,K1,P1) :- K1 mod 2 =:= 0, L1 is K1//2,
                        power(N1,L1,R1), P1 is R1*R1.
     {N1 = 3, K1 = 5, P1 = X}

 [1] 5 mod 2 =:= 0, L1 is 5//2, power(3,L1,R1), X is R1*R1
     {X = X}
     =:= : build-in
     failure - backtracking
```

```
[0] power(3,5,X)
    {X = X}
    3 --> 3
    power(N2,K2,P2) :- K2 mod 2 =:= 1, L2 is K2//2,
                       power(N2,L2,R2), P2 is R2*R2*N2.
    {N2 = 3, K2 = 5, P2 = X}

[1] 5 mod 2 =:= 1, L2 is 5//2, power(3,L2,R2), X is R2*R2*3
    {X = X}
    =:= : build-in
    {}

[2] L2 is 5//2, power(3,L2,R2), X is R2*R2*3
    {X = X}
    is : build-in
    {L2 = 2}

[3] power(3,2,R2), X is R2*R2*3
    {X = X}
    1 --> 2
    power(N3,K3,P3) :- K3 mod 2 =:= 0, L3 is K3//2,
                       power(N3,L3,R3), P3 is R3*R3.
    {N3 = 3, K3 = 2, P3 = R2}

[4] 2 mod 2 =:= 0, L3 is 2//2, power(3,L3,R3), R2 is R3*R3, X is R2*R2*3
    {X = X}
    =:= : build-in
    {}

[5] L3 is 2//2, power(3,L3,R3), R2 is R3*R3, X is R2*R2*3
    {X = X}
    is : build-in
    {L3 = 1}

[6] power(3,1,R3), R2 is R3*R3, X is R2*R2*3
    {X = X}
    1 --> 2
    power(N4,K4,P4) :- K4 mod 2 =:= 0, L4 is K4//2,
                       power(N4,L4,R4), P4 is R4*R4.
    {N4 = 3, K4 = 1, P4 = R3}

[7] 1 mod 2 =:= 0, L4 is 1//2, power(3,L4,R4), R3 is R4*R4, R2 is R3*R3,
                                                            X is R2*R2*3
    {X = X}
    =:= : build-in
    failure - backtracking
```

```
 [6] power(3,1,R3), R2 is R3*R3, X is R2*R2*3
     {X = X}
     3 --> 3
     power(N5,K5,P5) :- K5 mod 2 =:= 1, L5 is K5//2,
                        power(N5,L5,R5), P5 is R5*R5*N5.
     {N5 = 3, K5 = 1, P5 = R3}

 [7] 1 mod 2 =:= 1, L5 is 1//2, power(3,L5,R5), R3 is R5*R5*3, R2 is R3*R3,
                                                              X is R2*R2*3
     {X = X}
     =:= : build-in
     {}

 [8] L5 is 1//2, power(3,L5,R5), R3 is R5*R5*3, R2 is R3*R3, X is R2*R2*3
     {X = X}
     is : build-in
     {L5 = 0}

 [9] power(3,0,R5), R3 is R5*R5*3, R2 is R3*R3, X is R2*R2*3
     {X = X}
     1 --> 1
     power(N6,0,1).
     {N6 = 3, R5 = 1}

[10] R3 is 1*1*3, R2 is R3*R3, X is R2*R2*3
     {X = X}
     is : build-in
     {R3 = 3}

[11] R2 is 3*3, X is R2*R2*3
     {X = X}
     is : build-in
     {R2 = 9}

[12] X is 9*9*3
     {X = X}
     is : build-in
     {X = 243}

[13] #
     {X = 243}.
------------------------------------------------------------
X = 243
```

Η παραπάνω υλοποίηση έχει ένα μειονέκτημα: όταν το `K mod 2` έχει τιμή 1, τότε η τιμή αυτή υπολογίζεται και από τον δεύτερο και από τον τριτο κανόνα. Αυτό μπορεί να αποφευχθεί με χρήση της αποκοπής που θα δούμε παρακάτω. Ένας εναλλακτικός τρόπος φαίνεται στην παρακάτω υλοποίηση. Το βοηθητικό κατηγόρημα `powerHlp` χρησιμοποιείται για να υπολογιστεί μία κατάλληλη ποσότητα η οποία θα πολλαπλασιάσει το `R*R`, η οποία έχει

τιμή 1 αν το `K mod 2` έχει τιμή 0, αλλιώς έχει τιμή ίση με `K`.

```
power2(N,0,1).
power2(N,K,P) :- L is K//2,
                 power(N,L,R),
                 M is K mod 2,
                 powerHlp(M,N,Z),
                 P is R*R*Z.
powerHlp(0,N,1).
powerHlp(1,N,N).
```

∎

Παράδειγμα 20: υπολογισμός του αθροίσματος $\sum_{i=1}^{n} i^i$.

Για τον υπολογισμό του παραπάνω αθροίσματος θα ορίσουμε ένα κατηγόρημα `sum1(N,S)`, μεταφράζοντας στη σύνταξη της Prolog τις παρακάτω προτάσεις: (α) το $\sum_{i=1}^{0} i^i$ ισούται με 0 και (β) αν το N είναι θετικός ακέραιος, το K ισούται με $N-1$, το $\sum_{i=1}^{K} i^i$ ισούται με R, το N^N ισούται με P και το S ισούται με $R+P$, τότε το $\sum_{i=1}^{N} i^i$ ισούται με S.

```
sum1(0,0).
sum1(N,S) :- K is N-1,
             sum1(K,R),
             power(N,N,P),
             S is R+P.
```

(όπως και στα προηγούμενα παραδείγματα παραλείπουμε τη συνθήκη `N > 0` από τον αναδρομικό κανόνα). ∎

Παράδειγμα 21: υπολογισμός του μέγιστου κοινού διαιρέτη δύο θετικών αριθμών.

Ο μέγιστος κοινός διαιρέτης (ΜΚΔ) μπορεί να υπολογιστεί με τον αλγόριθμο του Ευκλείδη, ο οποίος βασίζεται στις παρακάτω προτάσεις (οι (β) και (γ) είναι συμμετρικές): (α) αν $M = N$ τότε ο ΜΚΔ των M και N είναι ο N, (β) αν $M < N$, το K ισούται με $N - M$ και ο D είναι ο ΜΚΔ των M και K τότε ο D είναι ο ΜΚΔ των M και N και (γ) αν $M > N$, το K ισούται με $M - N$ και ο D είναι ο ΜΚΔ των N και K τότε ο D είναι ο ΜΚΔ των M και N.

Μπορούμε να γράψουμε τις παραπάνω προτάσεις σε Prolog:

```
gcdEuc(N,N,N).
gcdEuc(M,N,D) :- M < N,
                 K is N-M,
                 gcdEuc(M,K,D).
gcdEuc(M,N,D) :- M > N,
                 K is M-N,
                 gcdEuc(N,K,D).
```

Για παράδειγμα, η Prolog θα απαντήσει στην ερώτηση

```
?- gcdEuc(84,36,X).
```

με τον παρακάτω τρόπο:

```
[0] gcdEuc(84,36,X)
    {X = X}
    1 --> 2
    gcdEuc(M1,N1,D1) :- M1 < N1, K1 is N1-M1, gcdEuc(M1,K1,D1).
    {M1 = 84, N1 = 36, D1 = X}

[1] 84 < 36, K1 is 36-84, gcdEuc(84,K1,X)
    {X = X}
    < : build-in
    failure - backtracking

[0] gcdEuc(84,36,X)
    {X = X}
    3 --> 3
    gcdEuc(M2,N2,D2) :- M2 > N2, K2 is M2-N2, gcdEuc(N2,K2,D2).
    {M2 = 84, N2 = 36, D2 = X}

[1] 84 > 36, K2 is 84-36, gcdEuc(36,K2,X)
    {X = X}
    > : build-in
    {}

[2] K2 is 84-36, gcdEuc(36,K2,X)
    {X = X}
    is : build-in
    {K2 = 48}

[3] gcdEuc(36,48,X)
    {X = X}
    1 --> 2
    gcdEuc(M3,N3,D3) :- M3 < N3, K3 is N3-M3, gcdEuc(M3,K3,D3).
    {M3 = 36, N3 = 48, D3 = X}

[4] 36 < 48, K3 is 48-36, gcdEuc(36,K3,X)
    {X = X}
    < : build-in
    {}

[5] K3 is 48-36, gcdEuc(36,K3,X)
    {X = X}
    is : build-in
    {K3 = 12}

[6] gcdEuc(36,12,X)
    {X = X}
    1 --> 2
    gcdEuc(M4,N4,D4) :- M4 < N4, K4 is N4-M4, gcdEuc(M4,K4,D4).
```

```
     {M4 = 36, N4 = 12, D4 = X}

 [7] 36 < 12, K4 is 12-36, gcdEuc(36,K4,X)
     {X = X}
     < : build-in
     failure - backtracking

 [6] gcdEuc(36,12,X)
     {X = X}
     3 --> 3
     gcdEuc(M5,N5,D5) :- M5 > N5, K5 is M5-N5, gcdEuc(N5,K5,D5).
     {M5 = 36, N5 = 12, D5 = X}

 [7] 36 > 12, K5 is 36-12, gcdEuc(12,K5,X)
     {X = X}
     > : build-in
     {}

 [8] K5 is 36-12, gcdEuc(12,K5,X)
     {X = X}
     is : build-in
     {K5 = 24}

 [9] gcdEuc(12,24,X)
     {X = X}
     1 --> 2
     gcdEuc(M6,N6,D6) :- M6 < N6, K6 is N6-M6, gcdEuc(M6,K6,D6).
     {M6 = 12, N6 = 24, D6 = X}

[10] 12 < 24, K6 is 24-12, gcdEuc(12,K6,X)
     {X = X}
     < : build-in
     {}

[11] K5 is 24-12, gcdEuc(12,K6,X)
     {X = X}
     is : build-in
     {K6 = 12}

[12] gcdEuc(12,12,X)
     {X = X}
     1 --> 1
     gcdEuc(N7,N7,N7).
     {N7 = 12, X = 12}.

[13] #
     {X = 12}
-----------------------------------------------------------------
X = 12
```

Μπορούμε να τροποποιήσουμε τον παραπάνω ορισμό έτσι ώστε πολλές εφαρμογές του ίδιου κανόνα στις οποίες η τιμή του ενός ορίσματος παραμένει σταθερή και η τιμές του άλλου προκύπτουν με διαδοχικές αφαιρέσεις, να αντικατασταθούν με μία εφαρμογή κανόνα στην οποία λαμβάνεται το υπόλοιπο διαίρεσης των δύο ορισμάτων:

```
gcdFast(0,M,M).
gcdFast(M,N,D) :- M < N,
                  K is N mod M,
                  gcdFast(K,M,D).
gcdFast(M,N,D) :- M >= N,
                  K is M mod N,
                  gcdFast(K,N,D).
```

∎

Παράδειγμα 22: υπολογισμός του διπλού αθροίσματος $\sum_{i=a}^{b}\sum_{j=c}^{d} i^j$ με $a \leq b$ και $c \leq d$.

Το άθροισμα υπολογίζεται χρησιμοποιώντας το παρακάτω αναδρομικό κατηγόρημα `sumabcd`, ο ορισμός του οποίου βασίζεται στις παρακάτω παρατηρήσεις: Αν $a = b$ και $c = d$ τότε το ζητούμενο άθροισμα ισούται με a^c. Αν $a = b$, $c < d$, $n = \lfloor \frac{c+d}{2} \rfloor$ και $k = n + 1$ τότε το ζητούμενο άθροισμα ισούται με το αποτέλεσμα της πρόσθεσης των δύο μερικών αθροισμάτων $\sum_{i=a}^{b}\sum_{j=c}^{n} i^j$ και $\sum_{i=a}^{b}\sum_{j=k}^{d} i^j$ που προκύπτουν διαμερίζοντας τις τιμές του j σε δύο σύνολα. Αν $a < b$, $m = \lfloor \frac{a+b}{2} \rfloor$ και $\ell = m + 1$ τότε το ζητούμενο άθροισμα ισούται με το αποτέλεσμα της πρόσθεσης των δύο μερικών αθροισμάτων $\sum_{i=a}^{m}\sum_{j=c}^{d} i^j$ και $\sum_{i=\ell}^{b}\sum_{j=c}^{d} i^j$.

```
sumabcd(A,A,C,C,S) :- power(A,C,S).
sumabcd(A,A,C,D,S) :- C < D,
                      N is (C+D)//2, K is N+1,
                      sumabcd(A,A,C,N,S1),
                      sumabcd(A,A,K,D,S2),
                      S is S1+S2.
sumabcd(A,B,C,D,S) :- A < B,
                      M is (A+B)//2, L is M+1,
                      sumabcd(A,M,C,D,S1),
                      sumabcd(L,B,C,D,S2),
                      S is S1+S2.
```

∎

Παράδειγμα 23: ακέραιο μέρος της τετραγωνικής ρίζας ενός αριθμού.

Θα ορίσουμε ένα κατηγόρημα για τον υπολογισμό του $\lfloor \sqrt{n} \rfloor$ με βάση τις παρακάτω παρατηρήσεις: Ισχύει ότι $0 \leq \lfloor \sqrt{n} \rfloor \leq n$. Επίσης, αν γνωρίζουμε ότι $a \leq \lfloor \sqrt{n} \rfloor \leq b$, όπου a και b μή αρνητικοί ακέραιοι, και $c = \lfloor \frac{a+b+1}{2} \rfloor$ τότε ισχύουν τα παρακάτω: (α) αν $a = b$ τότε οι ανισότητες γίνονται ισότητες, άρα $\lfloor \sqrt{n} \rfloor = a$, (β) αν $a < b$ και $c^2 > n$, τότε $a \leq \lfloor \sqrt{n} \rfloor \leq c - 1$ και (γ) αν $a < b$ και $c^2 \leq n$, τότε $c \leq \lfloor \sqrt{n} \rfloor \leq b$. Στις περιπτώσεις (β) και (γ) μπορούμε συγκρίνοντας το c^2 με το n να μειώσουμε τον αριθμό των υποψήφιων τιμών για το $\sqrt{n}$ περίπου στο μισό.

Το κατηγόρημα sqrtInt0 υπολογίζει το $\lfloor\sqrt{n}\rfloor$, χρησιμοποιώντας ένα βοηθητικό κατηγόρημα sqrtHlp0, ο ορισμός του οποίου στηρίζεται στις παραπάνω παρατηρήσεις:

```
sqrtInt0(N,R) :- sqrtHlp0(N,0,N,R).
sqrtHlp0(N,A,A,A).
sqrtHlp0(N,A,B,R) :- A < B,
                     C is (A+B+1)//2,
                     C*C > N,
                     D is C-1,
                     sqrtHlp0(N,A,D,R).
sqrtHlp0(N,A,B,R) :- A < B,
                     C is (A+B+1)//2,
                     C*C =< N,
                     sqrtHlp0(N,C,B,R).
```

Στην παραπάνω υλοποίηση, αν δεν αληθεύει η ανισότητα C*C > N στο σώμα του πρώτου κανόνα που το περιέχει, τότε οι παραστάσεις (A+B+1)//2 και C*C, αποτιμούνται ξανά όταν εφαρμοστεί ο δεύτερος κανόνας. Αυτό μπορεί να αποφευχθεί τροποποιώντας την sqrtInt0 ώστε να χρησιμοποιεί ένα βοηθητικό κατηγόρημα, το οποίο θα καθορίζει τα νέα όρια του διαστήματος αναζήτησης (μέσω των δύο τελευταίων ορισμάτων του).

```
sqrtInt(N,R) :- sqrtHlp(N,0,N,R).
sqrtHlp(N,A,A,A).
sqrtHlp(N,A,B,R) :- A < B,
                    C is (A+B+1)//2,
                    K is C*C,
                    newInterval(N,C,K,A,B,Anew,Bnew),
                    sqrtHlp(N,Anew,Bnew,R).
newInterval(N,C,K,A,B,A,D) :- K > N, D is C-1.
newInterval(N,C,K,A,B,C,B) :- K =< N.
```

Τα βήματα για την απάντηση στην ερώτηση

```
?- sqrtInt(12,X)
```

δίνονται παρακάτω:

```
 [0] sqrtInt(12,X)
     {X = X}
     1 --> 1
     sqrtInt(N1,R1) :- sqrtHlp(N1,0,N1,R1).
     {N1 = 12, R1 = X}

 [1] sqrtHlp(12,0,12,X)
     {X = X}
     1 --> 3
     sqrtHlp(N2,A2,B2,R2) :- A2 < B2, C2 is (A2+B2+1)//2, K2 is C2*C2,
                             newInterval(N2,C2,K2,A2,B2,Anew2,Bnew2),
                             sqrtHlp(N2,Anew2,Bnew2,R2).
     {N2 = 12, A2 = 0, B2 = 12, R2 = X}
```

```
[2] 0 < 12, C2 is (0+12+1)//2, K2 is C2*C2,
           newInterval(12,C2,K2,0,12,Anew2,Bnew2),
           sqrtHlp(12,Anew2,Bnew2,X)
    {X = X}
    < : build-in
    {}

[3] C2 is (0+12+1)//2, K2 is C2*C2, newInterval(12,C2,K2,0,12,Anew2,Bnew2),
           sqrtHlp(12,Anew2,Bnew2,X)
    {X = X}
    is : build-in
    {C2 = 6}

[4] K2 is 6*6, newInterval(12,6,K2,0,12,Anew2,Bnew2),
           sqrtHlp(12,Anew2,Bnew2,X)
    {X = X}
    is : build-in
    {K2 = 36}

[5] newInterval(12,6,36,0,12,Anew2,Bnew2), sqrtHlp(12,Anew2,Bnew2,X)
    {X = X}
    1 --> 4
    newInterval(N3,C3,K3,A3,B3,A3,D3) :- K3 > N3, D3 is C3-1.
    {N3 = 12, C3 = 6, K3 = 36, A3 = 0, B3 = 12, Anew2 = 0, D3 = Bnew2}

[6] 36 > 12, Bnew2 is 6-1, sqrtHlp(12,0,Bnew2,X)
    {X = X}
    > : build-in
    {}

[7] Bnew2 is 6-1, sqrtHlp(12,0,Bnew2,X)
    {X = X}
    is : build-in
    {Bnew2 = 5}

[8] sqrtHlp(12,0,5,X)
    {X = X}
    1 --> 3
    sqrtHlp(N4,A4,B4,R4) :- A4 < B4, C4 is (A4+B4+1)//2, K4 is C4*C4,
                            newInterval(N4,C4,K4,A4,B4,Anew4,Bnew4),
                            sqrtHlp(N4,Anew4,Bnew4,R4).
    {N4 = 12, A4 = 0, B4 = 5, R4 = X}

[9] 0 < 5, C4 is (0+5+1)//2, K4 is C4*C4,
           newInterval(12,C4,K4,0,5,Anew4,Bnew4),
           sqrtHlp(12,Anew4,Bnew4,X)
    {X = X}
    < : build-in
    {}
```

```
[10] C4 is (0+5+1)//2, K4 is C4*C4, newInterval(12,C4,K4,0,5,Anew4,Bnew4),
            sqrtHlp(12,Anew4,Bnew4,X)
     {X = X}
     is : build-in
     {C4 = 3}

[11] K4 is 3*3, newInterval(12,3,K4,0,5,Anew4,Bnew4),
            sqrtHlp(12,Anew4,Bnew4,X)
     {X = X}
     is : build-in
     {K4 = 9}

[12] newInterval(12,3,9,0,5,Anew4,Bnew4), sqrtHlp(12,Anew4,Bnew4,X)
     {X = X}
     1 --> 4
     newInterval(N5,C5,K5,A5,B5,A5,D5) :- K5 > N5, D5 is C5-1.
     {N5 = 12, C5 = 3, K5 = 9, A5 = 0, B5 = 5, Anew4 = 0, D5 = Bnew4}

[13] 9 > 12, Bnew4 is 3-1, sqrtHlp(12,0,Bnew4,X)
     {X = X}
     > : build-in
     failure - backtracking

[12] newInterval(12,3,9,0,5,Anew4,Bnew4), sqrtHlp(12,Anew4,Bnew4,X)
     {X = X}
     5 --> 5
     newInterval(N6,C6,K6,A6,B6,C6,B6) :- K6 =< N6.
     {N6 = 12, C6 = 3, K6 = 9, A6 = 0, B6 = 5, Anew4 = 3, Bnew4 = 5}

[13] 9 =< 12, sqrtHlp(12,3,5,X)
     {X = X}
     =< : build-in
     {}

[14] sqrtHlp(12,3,5,X)
     {X = X}
     1 --> 3
     sqrtHlp(N7,A7,B7,R7) :- A7 < B7, C7 is (A7+B7+1)//2, K7 is C7*C7,
                             newInterval(N7,C7,K7,A7,B7,Anew7,Bnew7),
                             sqrtHlp(N7,Anew7,Bnew7,R7).
     {N7 = 12, A7 = 3, B7 = 5, R7 = X}

[15] 3 < 5, C7 is (3+5+1)//2, K7 is C7*C7,
            newInterval(12,C7,K7,3,5,Anew7,Bnew7),
            sqrtHlp(12,Anew7,Bnew7,X)
     {X = X}
     < : build-in
     {}
```

```
[16] C7 is (3+5+1)//2, K7 is C7*C7, newInterval(12,C7,K7,3,5,Anew7,Bnew7),
            sqrtHlp(12,Anew7,Bnew7,X)
     {X = X}
     is : build-in
     {C7 = 4}

[17] K7 is 4*4, newInterval(12,4,K7,3,5,Anew7,Bnew7),
            sqrtHlp(12,Anew7,Bnew7,X)
     {X = X}
     is : build-in
     {K7 = 16}

[18] newInterval(12,4,16,3,5,Anew7,Bnew7), sqrtHlp(12,Anew7,Bnew7,X)
     {X = X}
     1 --> 4
     newInterval(N8,C8,K8,A8,B8,A8,D8) :- K8 > N8, D8 is C8-1.
     {N8 = 12, C8 = 4, K8 = 16, A8 = 3, B8 = 5, Anew2 = 3, D8 = Bnew7}

[19] 16 > 12, Bnew7 is 4-1, sqrtHlp(12,3,Bnew7,X)
     {X = X}
     > : build-in
     {}

[20] Bnew7 is 4-1, sqrtHlp(12,3,Bnew7,X)
     {X = X}
     is : build-in
     {Bnew7 = 3}

[21] sqrtHlp(12,3,3,X)
     {X = X}
     1 --> 2
     sqrtHlp(N9,A9,A9,A9).
     {N9 = 12, A9 = 3, X = 3}

[22] #
     {X = 3}
------------------------------------------------------------
X = 3
```

1.16 Λίστες

Οι λίστες στην Prolog παριστάνονται παραθέτοντας τα στοιχεία τους μέσα σε αγκύλες, χωρίζοντάς τα με κόμμα. Μία λίστα επιτρέπεται να περιέχει όρους οποιασδήποτε μορφής, ακόμη και λίστες.
Για παράδειγμα οι `[a,b,c]`, `[1,2,3,4]` και `[32, 'Chris', <><><>, X, [a, 1], s(0)]` είναι αποδεκτές λίστες. Η κενή λίστα συμβολίζεται με `[]`.

Ονομάζουμε κεφαλή μιας μη κενής λίστας το πρώτο στοιχείο της και ουρά τη λίστα που απομένει αν διαγράψουμε την κεφαλή. Εναλλακτικά γράφουμε `[H|T]` για να δηλώσουμε τη λίστα με κεφαλή `H` και ουρά `T`.

```
?- [H|T] = [a,b,c].
H = a
T = [b,c]
```

Γενικότερα γραφουμε `[X1,X2,...,Xk|T]` για να παραστήσουμε τη λίστα στην οποία τα πρώτα k στοιχεία είναι τα `X1`, `X2`, ..., `Xk` και η λίστα που απομένει αν διαγράψουμε τα στοιχεία αυτά είναι η `T`

```
?- [X1,X2,X3|T] = [1,2,3,4,5].
X1 = 1
X2 = 2
X3 = 3
T = [4,5]
```

Σύμφωνα με τα παραπάνω η λίστα `[a,b,c,d]` έχει τις παρακάτω εναλλακτικές περιγραφές: `[a|[b,c,d]]`, `[a,b|[c,d]]`, `[a,b,c|[d]]` και `[a,b,c,d|[]]`

Η αναπαράσταση των στοιχείων μίας λίστας μέσα σε αγκύλες είναι η εξωτερική αναπαράσταση που χρησιμοποιείται από την Prolog για να διαβάζει και να εμφανίζει τις λίστες. Εσωτερικά μία λίστα παριστάνεται ως όρος που σχηματίζεται από το συναρτησιακό σύμβολο `.` το οποίο παίρνει ως ορίσματα την κεφαλή και την ουρά της λίστας. Για παράδειγμα η λίστα `[a,b,c,d]` εσωτερικα παριστάνεται ως: `.(a,.(b,.(c,.(d,[]))))`.

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

Η εσωτερική αναπαράσταση είναι δυσανάγνωστη και αυτός είναι ο λόγος για τον οποίο στην Prolog επιλέγεται διαφορετική εξωτερική αναπαράσταση. Από την άλλη, η εσωτερική αναπαράσταση μίας λίστας είναι όρος με τη συνήθη μορφή και μπορεί για παράδειγμα να αποτελέσει είσοδο στον αλγόριθμο ταυτοποίησης:

Παράδειγμα 24: Για να δούμε αν οι λίστες `[[a,b]|[X,Y]]` και `[Z,W|[c,d]]` μπορούν να ταυτοποιηθούν τις μετατρέπουμε στην εσωτερική τους αναπαράσταση και εκτελούμε τον αλγόριθμο ταυτοποίησης:

βήμα	E_1 / E_2	δεσμεύσεις
1	`.(`**`.(a,.(b,[]))`**`,.(X,.(Y,[])))` `.(`**`Z`**`,.(W,.(c,.(d,[]))))`	{`X` = `X`, `Y` = `Y`, `Z` = `Z`, `W` = `W`}
2	`.(.(a,.(b,[])),.(`**`X`**`,.(Y,[])))` `.(.(a,.(b,[])),.(`**`W`**`,.(c,.(d,[]))))`	{`X` = `X`, `Y` = `Y`, `Z` = `.(a,.(b,[]))`, `W` = `W`}
3	`.(.(a,.(b,[])),.(X,.(`**`Y`**`,[])))` `.(.(a,.(b,[])),.(X,.(`**`c`**`,.(d,[]))))`	{`X` = `X`, `Y` = `Y`, `Z` = `.(a,.(b,[]))`, `W` = `X`}
4	`.(.(a,.(b,[])),.(X,.(c,`**`[]`**`)))` `.(.(a,.(b,[])),.(X,.(c,`**`.(d,[])`**`)))`	{`X` = `X`, `Y` = `c`, `Z` = `.(a,.(b,[]))`, `W` = `X`}

Η ταυτοποίηση αποτυγχάνει. Αυτό είναι αναμενόμενο καθώς η μία λίστα έχει τρία στοιχεία (από τα οποία το πρώτο είναι λίστα) ενώ η δεύτερη έχει τέσσερα στοιχεία. ∎

Παράδειγμα 25: Για να δούμε αν οι λίστες `[X,Y|Y]` και `[Y,[2],Z]` μπορούν να ταυτοποιηθούν εκτελούμε τον αλγόριθμο ταυτοποίησης με είσοδο τις εσωτερικές τους αναπαραστάσεις:

βήμα	E_1 / E_2	δεσμεύσεις
1	`.(X,.(Y,Y))` `.(Y,.(.(2,[]),.(Z,[])))`	{`X` = `X`, `Y` = `Y`, `Z` = `Z` }
2	`.(X,.(X,X))` `.(X,.(.(2,[]),.(Z,[])))`	{`X` = `X`, `Y` = `X`, `Z` = `Z` }
3	`.(.(2,[]),.(.(2,[]),.(2,[])))` `.(.(2,[]),.(.(2,[]),.(Z,[])))`	{`X` = `.(2,[])`, `Y` = `.(2,[])`, `Z` = `Z` }
4	`.(.(2,[]),.(.(2,[]),.(2,[])))` `.(.(2,[]),.(.(2,[]),.(2,[])))`	{`X` = `.(2,[])`, `Y` = `.(2,[])`, `Z` = `2` }

Στην αρχή του τέταρτου βήματος οι εκφράσεις είναι ίδιες, συνεπώς οι αρχικές λίστες ταυτοποιούνται με πιο γενικό ταυτοποιητή {`X` = `[2]`, `Y` = `[2]`, `Z` = `2`}. ∎

Στη συνέχεια θα περιγράψουμε ορισμένα κατηγορήματα που παίρνουν ως ορίσματα λίστες. Για να επεξεργαστούμε τα στοιχεία μίας λίστας, χρησιμοποιούμε αναδρομή.

Παράδειγμα 26: έλεγχος για το αν ένα στοιχείο εμφανίζεται σε μία λίστα.

Το κατηγόρημα `member(X,L)` αληθεύει αν το στοιχείο `X` εμφανίζεται στη λίστα `L`. Αυτό συμβαίνει αν ισχύει ένα από τα παρακάτω: (α) το `X` είναι η κεφαλή της λίστας, που σημαίνει ότι η λίστα είναι της μορφής `[X|T]` ή (β) το `X` εμφανίζεται στην ουρά της λίστας, που σημαίνει ότι η λίστα είναι της μορφής `[H|T]` και αληθεύει το `member(X,T)`.

```
member(X,[X|T]).
member(X,[H|T]) :- member(X,T).
```

Αν η λίστα είναι κενή τότε δεν γίνεται ταυτοποίηση με κανέναν κανόνα και επιστρέφεται `no`.

Η Prolog για να απαντήσει στην ερώτηση

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

θα εκτελέσει τα παρακάτω βήματα:

```
[0] member(b,[a,b,c])
    {}
    1 --> 2
    member(X1,[H1|T1]) :- member(X1,T1).
    {X1 = b, H1 = a, T1 = [b,c]}
```

```
[1] member(b,[b,c])
    {}
    1 --> 1
    member(X2,[X2|T2]).
    {X2 = b, T2 = [c]}

[2] #
    {}
------------------------------------------------------------
yes
```

Χρησιμοποιώντας το κατηγόρημα member μπορούμε να βρούμε τα στοιχεία μίας λίστας ή να βρούμε τις λίστες που περιέχουν ένα δεδομένο στοιχείο:

```
?- member(X,[1,c,[2,3],f(2)]).
X = 1 ;
X = c ;
X = [2,3] ;
X = f(2) ;
no

?- member(b,L).
L = [b|_6] ;
L = [_5,b|_10] ;
L = [_5,_9,b|_14] ;
...
```

Η απάντηση στην τελευταία ερώτηση επιστρέφει μια προς μία όλες τις λίστες που περιέχουν το b. Στην απάντηση υπάρχει μία λίστα για κάθε πιθανή θέση εμφάνισης του b. Οι μεταβλητές αριστερά του b παριστάνουν τα στοιχεία της λίστας που προηγούνται του b και μπορεί να έχουν οποιαδήποτε τιμή. Η μεταβλητή δεξιά του b (μετά το |) παριστάνει το υπόλοιπο τμήμα της λίστας το οποίο μπορεί να έχει οποιοδήποτε μήκος και να περιέχει οποιαδήποτε στοιχεία. (Τα ονόματα των μεταβλητών προέρχονται από τη μετονομασία των μεταβλητών στους κανόνες από το διερμηνέα της C-Prolog.) ∎

Παράδειγμα 27: διαγραφή ενός δεδομένου στοιχείου από λίστα.

Το κατηγόρημα delete(X,L1,L2) αληθεύει αν η L2 προκύπτει με διαγραφή μίας οποιασδήποτε εμφάνισης του X από τη L1. Ο ορισμός του delete γίνεται με βάση τις παρακάτω παρατηρήσεις: (α) αν το στοιχείο X που διαγράφεται είναι η κεφαλή της L1, τότε η L1 είναι της μορφής [X|T] και η λίστα L2 που προκύπτει ταυτίζεται με την T και (β) αν το στοιχείο X διαγράφεται από την ουρά της L1, τότε οι L1 και L2 έχουν την ίδια κεφαλή H και η ουρά S της L2 προκύπτει με διαγραφή του X από την ουρά T της L1.

```
delete(X,[X|T],T).
delete(X,[H|T],[H|S]) :- delete(X,T,S).
```

Για να βρούμε τις λίστες που προκύπτουν από την διαγραφή του 4 από τη λίστα [7,4,5,4,1] κάνουμε την ερώτηση

```
?- delete(4,[7,4,5,4,1],L)
```

και γράφουμε ; μετά από κάθε απάντηση. Για να απαντηθεί η ερώτηση αυτή εκτελούνται τα παρακάτω βήματα:

```
[0] delete(4,[7,4,5,4,1],L)
    {L = L}
    1 --> 2
    delete(X1,[H1|T1],[H1|S1]) :- delete(X1,T1,S1).
    {L = [7|S1], X1 = 4, H1 = 7, T1 = [4,5,4,1]}

[1] delete(4,[4,5,4,1],S1)
    {L = [7|S1]}
    1 --> 1
    delete(X2,[X2|T2],T2).
    {S1 = [5,4,1], X2 = 4, T2 = [5,4,1]}

[2] #
    {L = [7,5,4,1]}
------------------------------------------------------------
L = [7,5,4,1] ;
------------------------------------------------------------
[1] delete(4,[4,5,4,1],S1)
    {L = [7|S1]}
    2 --> 2
    delete(X3,[H3|T3],[H3|S3]) :- delete(X3,T3,S3).
    {S1 = [4|S3]], X3 = 4, H3 = 4, T3 = [5,4,1]}

[2] delete(4,[5,4,1],S3)
    {L = [7,4|S3]}
    1 --> 2
    delete(X4,[H4|T4],[H4|S4]) :- delete(X4,T4,S4).
    {S3 = [5|S4], X4 = 4, H4 = 5, T4 = [4,1]}

[3] delete(4,[4,1],S4)
    {L = [7,4,5|S4]}
    1 --> 1
    delete(X5,[X5|T5],T5).
    {S4 = [1], X5 = 4, T5 = [1]}

[4] #
    {L = [7,4,5,1]}
------------------------------------------------------------
L = [7,4,5,1] ;
------------------------------------------------------------
[3] delete(4,[4,1],S4)
    {L = [7,4,5|S4]}
    2 --> 2
    delete(X6,[H6|T6],[H6|S6]) :- delete(X6,T6,S6).
    {S4 = [4|S6]], X6 = 4, H6 = 4, T6 = [1]}
```

```
[4] delete(4,[1],S6)
    {L = [7,4,5,4|S6]}
    1 --> 2
    delete(X7,[H7|T7],[H7|S7]) :- delete(X7,T7,S7).
    {S6 = [1|S7], X7 = 4, H7 = 1, T7 = []}

[5] delete(4,[],S7)
    {L = [7,4,5,4,1|S7]}
    1 --> EOP
    failure - backtraching

[4] delete(4,[1],S6)
    {L = [7,4,5,4|S6]}
    EOP
    failure - backtraching

[3] delete(4,[4,1],S4)
    {L = [7,4,5|S4]}
    EOP
    failure - backtraching

[2] delete(4,[5,4,1],S3)
    {L = [7,4|S3]}
    EOP
    failure - backtraching

[1] delete(4,[4,5,4,1],S1)
    {L = [7|S1]}
    EOP
    failure - backtraching

[0] delete(4,[7,4,5,4,1],L)
    {L = L}
    EOP
    failure
-------------------------------------------------------------
no
```

Παρατηρούμε ότι αν η L2 προκύπτει με διαγραφή μίας εμφάνισης του X από την L1, τότε η L1 προκύπτει με εισαγωγή του X σε μία οποιαδήποτε θέση της L2. Για να βρούμε τις λίστες που προκύπτουν με εισαγωγή του a στη λίστα [b,c] κάνουμε την ερώτηση

```
| ?- delete(a,L,[b,c]).
L = [a,b,c] ;
L = [b,a,c] ;
L = [b,c,a] ;
no
```

Χρησιμοποιώντας το delete μπορούμε να ελέγξουμε αν μία λίστα προκύπτει από μία άλλη με διαγραφή ενός στοιχείου

```
?-  delete(_,[a,c,b,a],[a,b,a]).
yes

?- delete(_,[a,c,b,a],[a,b]).
no
```

Μπορούμε επίσης να κάνουμε ερωτήσεις με περισσότερες από μία μεταβλητές:

```
?-  delete(X,L1,L2).
X = _0
L1 = [_0|_2]
L2 = _2 ;

X = _0
L1 = [_8,_0|_10]
L2 = [_8|_10] ;


X = _0
L1 = [_8,_15,_0|_17]
L2 = [_8,_15|_17] ;

...
```

Παράδειγμα 28: μήκος λίστας.

Ορίζουμε το κατηγόρημα `lengthList(L,N)` το οποίο αληθεύει αν το μήκος της λίστας `L` είναι `N`. Το μήκος της κενής λίστας είναι 0, ενώ το μήκος μίας μη κενής λίστας είναι ίσο με το μήκος της ουράς της αυξημένο κατά 1.

```
lengthList([],0).
lengthList([H|T],N) :- lengthList(T,M),
                       N is M+1.
```

Για να βρούμε το μήκος της λίστας `[a,b,c]` κάνουμε την ερώτηση

```
?- lengthList([a,b,c],N)
```

Η απάντηση στην ερώτηση αυτή γίνεται εκτελώντας τα παρακάτω βήματα:

```
[0] lengthList([a,b,c],N)
    {N = N}
    1 --> 2
    lengthList([H1|T1],N1) :- lengthList(T1,M1), N1 is M1+1.
    {H1 = a, T1 = [b,c], N1 = N}

[1] lengthList([b,c],M1), N is M1+1
    {N = N}
    1 --> 2
    lengthList([H2|T2],N2) :- lengthList(T2,M2), N2 is M2+1.
```

```
    {H2 = b, T2 = [c], N2 = M1}

[2] lengthList([c],M2), M1 is M2+1, N is M1+1
    {N = N}
    1 --> 2
    lengthList([H3|T3],N3) :- lengthList(T3,M3), N3 is M3+1.
    {H3 = c, T3 = [], N3 = M2}

[3] lengthList([],M3), M2 is M3+1, M1 is M2+1, N is M1+1
    {N = N}
    1 --> 1
    lengthList([],0).
    {M3 = 0}

[4] M2 is 0+1, M1 is M2+1, N is M1+1
    {N = N}
    is : build-in
    {M2 = 1}

[5] M1 is 1+1, N is M1+1
    {N = N}
    is : build-in
    {M1 = 2}

[6] N is 2+1
    {N = N}
    is : build-in
    {N = 3}

[7] #
    {N = 3}
------------------------------------------------------------
N = 3
```

Η Prolog επιστρέφει απάντηση ακόμη και αν κάνουμε ερώτηση με δύο μεταβλητές ώς ορίσματα του `lengthList(L,N)` Τα βήματα για την απάντηση σε μια τέτοια ερώτηση δίνονται παρακάτω:

```
[0] lengthList(L,N)
    {L = L, N = N}
    1 --> 1
    lengthList([],0).
    {L = [], N = 0}

[1] #
    {L = [], N = 0}
------------------------------------------------------------
L = []
N = 0 ;
------------------------------------------------------------
```

```
[0] lengthList(L,N)
    {L = L, N = N}
    2 --> 2
    lengthList([H1|T1],N1) :- lengthList(T1,M1), N1 is M1+1.
    {L = [H1|T1], N1 = N}

[1] lengthList(T1,M1), N is M1+1
    {L = [H1|T1], N = N}
    1 --> 1
    lengthList([],0).
    {T1 = [], M1 = 0}

[2] N is 0+1
    {L = [H1], N = N}
    is : build-in
    {N = 1}

[3] #
    {L = [H1], N = 1}
------------------------------------------------------------
L = [H1]
N = 1 ;
------------------------------------------------------------
[2] N is 0+1
[1] lengthList(T1,M1), N is M1+1
    {L = [H1|T1], N = N}
    2 --> 2
    lengthList([H2|T2],N2) :- lengthList(T2,M2), N2 is M2+1.
    {T1 = [H2|T2], N2 = M1}

[2] lengthList(T2,M2), M1 is M2+1, N is M1+1
    {L = [H1,H2|T2], N = N}
    1 --> 1
    lengthList([],0).
    {T2 = [], M2 = 0}

[3] M1 is 0+1, N is M1+1
    {L = [H1,H2], N = N}
    is : build-in
    {M1 = 1}

[2] N is 1+1
    {L = [H1,H2], N = N}
    is : build-in
    {N = 2}

[3] #
    {L = [H1,H2], N = 2}
```

```
-----------------------------------------------------------
L = [H1,H2]
N = 2 ;
-----------------------------------------------------------
...
```

Συνήθως οι υλοποιήσεις της Prolog παρέχουν ένα προκαθορισμένο κατηγόρημα `length` που είναι ισοδύναμο του `lengthList` ∎

Παράδειγμα 29: εύρεση του n-οστού στοιχείου μιας λίστας.

Το κατηγόρημα `elemList(N,L,X)` αληθεύει αν το `N`-οστό στοιχείο της λίστας `L` είναι το `X`. Το `elemList(N,L,X)` μπορεί να αληθεύει μόνο στην περίπτωση που η λίστα είναι μή κενή, που είναι δηλαδή της μορφής `[H|T]`. Το πρώτο στοιχείο της `[H|T]` είναι προφανώς το `H`. Το `N`-οστό στοιχείο της `[H|T]` είναι το `M`-οστό στοιχείο της `T`, όπου η τιμή του `M` είναι `N-1`.

```
elemList(1,[H|T],H).
elemList(N,[H|T],X) :- M is N-1,
                       elemList(M,T,X).
```

Αν η τιμή της `N` δεν αντιστοιχεί σε στοιχείο της λίστας τότε θα προκύψει στόχος στον οποίο του δεύτερο όρισμα του `elemList` θα είναι η κενή λίστα. Στο σημείο αυτό η ταυτοποίηση θα αποτύχει και η απάντηση θα είναι `no`

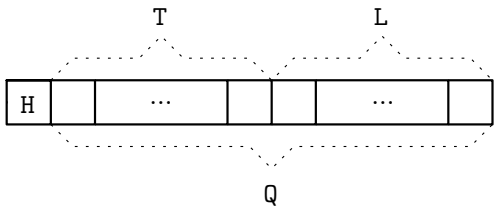
Λόγω της χρήσης του `is`, η μεταβλητή `N` θα πρέπει να είναι δεσμευμένη σε αριθμητική τιμή, αλλιώς θα προκύψει μήνυμα λάθους.

```
?- elemList(3,[1,9,6,8],X).
X = 6

?- elemList(5,[a,b,c],X).
no

?- elemList(-2,[a,b,c],X).
no

?- elemList(N,[1,9,6,8],8).
! Error in arithmetic expression: not a number
```

∎

Παράδειγμα 30: συνένωση δύο λιστών.

Το κατηγόρημα `conc(S,L,R)` αληθεύει η `R` είναι λίστα που προκύπτει με συνένωση των `S` και `L`. Ο ορισμός του `conc` στηρίζεται στις παρακάτω παρατηρήσεις: (α) το αποτέλεσμα της συνένωσης της κενής λίστας με τη λίστα `L` είναι η ίδια η `L` και (β) το αποτέλεσμα της συνένωσης μίας λίστας της μορφής `[H|T]` με την `L` είναι η λίστα με κεφαλή το `H` και ουρά την λίστα `Q` που προκύπτει από τη συνένωση των `T` και `L`.




```
conc([],L,L).
conc([H|T],L,[H|Q]) :- conc(T,L,Q).
```

Για να βρούμε τη συνένωση των λιστών [a,b] και [c,d,e] κάνουμε την ερώτηση:

```
?- conc([a,b],[c,d,e],L).
L = [a,b,c,d,e]
```

Επίσης μπορούμε να βρούμε όλα τα ζεύγη λιστών στα οποία διασπάται μία δεδομένη λίστα:

```
?- conc(A,B,[a,b,c]).
```

Τα βήματα για την απάντηση στην ερώτηση αυτή δίνονται παρακάτω:

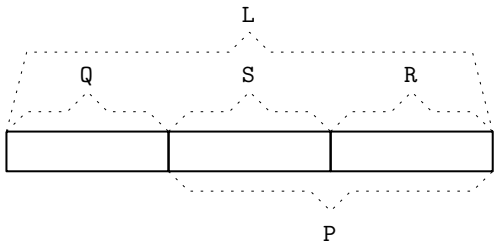
```
[0] conc(A,B,[a,b,c])
    {A = A, B = B}
    1 --> 1
    conc([],L1,L1).
    {A = [], B = [a,b,c], L1 = [a,b,c]}

[1] #
    {A = [], B = [a,b,c]}
----------------------------------------------------------------
A = []
B = [a,b,c] ;
----------------------------------------------------------------
[0] conc(A,B,[a,b,c])
    {A = A, B = B}
    2 --> 2
    conc([H2|T2],L2,[H2|S2]) :- conc(T2,L2,S2).
    {A = [a|T2], H2 = a, L2 = B, S2 = [b,c]}

[1] conc(T2,B,[b,c])
    {A = [a|T2], B = B}
    1 --> 1
    conc([],L3,L3).
    {T2 = [], B = [b,c], L3 = [b,c]}

[2] #
    {A = [a], B = [b,c]}
----------------------------------------------------------------
A = [a]
B = [b,c] ;
----------------------------------------------------------------
```

```
[1] conc(T2,B,[b,c])
    {A = [a|T2], B = B}
    2 --> 2
    conc([H4|T4],L4,[H4|S4]) :- conc(T4,L4,S4).
    {T2 = [b|T4], H4 = b, L4 = B, S4 = [c]}

[2] conc(T4,B,[c])
    {A = [a,b|T4], B = B}
    1 --> 1
    conc([],L5,L5).
    {T4 = [], B = [c], L5 = [c]}

[3] #
    {A = [a,b], B = [c]}
------------------------------------------------------------
A = [a,b]
B = [c] ;
------------------------------------------------------------
[2] conc(T4,B,[c])
    {A = [a,b|T4], B = B}
    2 --> 2
    conc([H6|T6],L6,[H6|S6]) :- conc(T6,L6,S6).
    {T4 = [c|T6], H6 = c, L6 = B, S6 = []}

[3] conc(T6,B,[])
    {A = [a,b,c|T6], B = B}
    1 --> 1
    conc([],L7,L7).
    {T6 = [], B = [], L7 = []}

[4] #
    {A = [a,b,c], B = []}
------------------------------------------------------------
A = [a,b,c]
B = [] ;
------------------------------------------------------------
[3] conc(T6,B,[])
    {A = [a,b,c|T6], B = B}
    2 --> EOP
    failure - backtracking

[2] conc(T4,B,[c])
    {A = [a,b|T4], B = B}
    EOP
    failure - backtracking

[1] conc(T2,B,[b,c])
    {A = [a|T2], B = B}
    EOP
```

```
    failure - backtracking

[0] conc(A,B,[a,b,c])
    {A = A, B = B}
    EOP
    failure
-----------------------------------------------------------------
no
```

Παράδειγμα 31: έλεγχος για το αν μία λίστα αποτελεί υπολίστα μίας άλλης.

Το κατηγόρημα `sublist(S,L)` αληθεύει η λίστα `S` αποτελεί υπολίστα της `L`, δηλαδή αν όλα τα στοχεία της `S` εμφανίζονται συνεχόμενα και με την ίδια σειρά στην `L`. Είναι εύκολο να διαπιστωθεί ότι η `S` είναι υπολίστα της `L`, αν υπάρχουν δύο λίστες `Q` και `R` τέτοιες ώστε η συνένωση των `Q`, `S` και `R` να δίνει την `L`.



Το κατηγόρημα `sublist` ορίζεται από τον παρακάτω κανόνα:

```
sublist(S,L) :- conc(Q,P,L),
                conc(S,R,P).
```

Μπορούμε να ελέγξουμε αν μία λίστα είναι υπολίστα μίας άλλης ή να βρούμε όλες τις υπολίστες μίας δεδομένης λίστας:

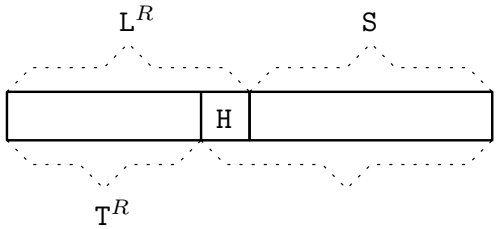
```
?- sublist([2,3],[1,2,3,4]).
yes

?- sublist([2,4],[1,2,3,4,5]).
no

?- sublist(S,[a,b,c]).
S = [] ;
S = [a] ;
S = [a,b] ;
S = [a,b,c] ;
S = [] ;
S = [b] ;
S = [b,c] ;
S = [] ;
S = [c] ;
S = [] ;
no
```

Η κενή λίστα επιστρέφεται ως τιμή της L πολλές φορές για το παρακάτω λόγω: η λίστα [a,b,c] μπορεί να διασπαστεί σε δύο λίστες Q και P με τέσσερεις διαφορετικούς τρόπους, και η P μπορεί πάντα να διασπαστεί την κενή και τον εαυτό της.

∎

Παράδειγμα 32: αντιστροφή λίστας.

Το κατηγόρημα reverse(L,R) αληθεύει αν η R είναι η αντίστροφη της L. Για τον ορισμό του reverse χρησιμοποιούμε ένα βοηθητικό κατηγόρημα concReversed(L,S,R), το οποίο αληθεύει αν η αν η R είναι το αποτέλεσμα της συνένωσης της αντίστροφης της L με την S. Είναι προφανές ότι το reverse(L,R) αληθεύει αν και μόνο αν αληθεύει το concReversed(L,[],R). Για το ορισμό του concReversed(L,S,R) παρατηρούμε ότι στη λίστα που προκύπτει αν συνενώσουμε την αντίστροφη μίας μή κενής λίστας [H|T] με την S, το H βρίσκεται αμέσως πριν από τα στοιχεία της S, ενώ αριστερά του H βρίσκονται τα στοιχεία της T σε αντίστροφη σειρά. Συνεπώς η ίδια λίστα μπορεί να προκύψει αν συνενώσουμε την αντίστροφη της T με την [H|S]




```
reverse(L,R) :- concReversed(L,[],R).

concReversed([],S,S).
concReversed([H|T],S,R) :- concReversed(T,[H|S],R).
```

Για να βρούμε την αντίστροφη της λίστας [1,2,3] κάνουμε την ερώτηση:

```
?- reverse([1,2,3],L).
L = [3,2,1]
```

Παρατηρούμε ότι το κατηγόρημα reverse είναι συμμετρικό: αν η R είναι η αντίστροφη της L τότε και η αν η L είναι η αντίστροφη της R. Συνεπώς μπορούμε να βρούμε την αντίστροφη της λίστας [1,2,3] κάνοντας την ερώτηση:

```
?- reverse(L,[1,2,3]).
L = [3,2,1]
```

Παρότι οι δύο ερωτήσεις είναι ισοδύναμες, τα βήματα που ακολουθεί η Prolog για να απαντήσει σε καθεμία από αυτές είναι διαφορετικά. Επίσης αν ζητήσουμε δεύτερη απάντηση, με την πρώτη ερώτηση η Prolog θα απαντήσει no, ενώ με τη δεύτερη θα εκτελέσει μία ατέρμονη ακολουθία βημάτων, που θα προκαλέσει υπερχείλιση στοίβας. Τα βήματα που θα εκτελεστούν για να απαντηθεί η πρώτη ερώτηση είναι τα παρακάτω:

```
[0] reverse([1,2,3],L).
    {L = L}
    1 --> 1
    reverse(L1,R1) :- concReversed(L1,[],R1).
```

```
    {L1 = [1,2,3], R1 = L}

[1] concReversed([1,2,3],[],L)
    {L = L}
    1 --> 3
    concReversed([H2|T2],L2,R2) :- concReversed(T2,[H2|L2],R2).
    {H2 = 1, T2 = [2,3], L2 = [], R2 = L}

[2] concReversed([2,3],[1],L)
    {L = L}
    1 --> 3
    concReversed([H3|T3],L3,R3) :- concReversed(T3,[H3|L3],R3).
    {H3 = 2, T3 = [3], L3 = [1], R3 = L}

[3] concReversed([3],[2,1],L)
    {L = L}
    1 --> 3
    concReversed([H4|T4],L4,R4) :- concReversed(T4,[H4|L4],R4).
    {H4 = 3, T4 = [], L4 = [2,1], R4 = L}

[4] concReversed([],[3,2,1],L)
    {L = L}
    1 --> 2
    concReversed([],L5,L5).
    {L = [3,2,1], L5 = [3,2,1]}

[5] #
    {L = [3,2,1]}
-----------------------------------------------------------------
L = [3,2,1] ;
-----------------------------------------------------------------
[4] concReversed([],[3,2,1],L)
    {L = L}
    3 --> EOP
    failure - backtracking

[3] concReversed([3],[2,1],L)
    {L = L}
    EOP
    failure - backtracking

[2] concReversed([2,3],[1],L)
    {L = L}
    EOP
    failure - backtracking

[1] concReversed([1,2,3],[],L)
    {L = L}
    EOP
```

```
    failure - backtracking

[0] reverse([1,2,3],L).
    {L = L}
    2 --> EOP
    failure
-----------------------------------------------------------
no
```

Τα βήματα που θα εκτελεστούν για να απαντηθεί η δεύτερη ερώτηση είναι τα παρακάτω:

```
[0] reverse(L,[1,2,3]).
    {L = L}
    1 --> 1
    reverse(L1,R1) :- concReversed(L1,[],R1).
    {L1 = L, R1 = [1,2,3]}

[1] concReversed(L,[],[1,2,3])
    {L = L}
    1 --> 3
    concReversed([H2|T2],L2,R2) :- concReversed(T2,[H2|L2],R2).
    {L = [H2|T2], L2 = [], R2 = [1,2,3]}

[2] concReversed(T2,[H2],[1,2,3])
    {L = [H2|T2]}
    1 --> 3
    concReversed([H3|T3],L3,R3) :- concReversed(T3,[H3|L3],R3).
    {T2 = [H3|T3], L3 = [H2], R3 = [1,2,3]}

[3] concReversed(T3,[H3,H2],[1,2,3])
    {L = [H2,H3|T3]}
    1 --> 3
    concReversed([H4|T4],L4,R4) :- concReversed(T4,[H4|L4],R4).
    {T3 = [H4|T4], L4 = [H3,H2], R4 = [1,2,3]}

[4] concReversed(T4,[H4,H3,H2],[1,2,3])
    {L = [H2,H3,H4|T4]}
    1 --> 2
    concReversed([],L5,L5).
    {T4 = [], H4 = 1, H3 = 2, H2 = 3, L5 = [1,2,3]}

[5] #
    {L = [3,2,1]}
-----------------------------------------------------------
L = [3,2,1] ;
-----------------------------------------------------------
[4] concReversed(T4,[H4,H3,H2],[1,2,3])
    {L = [H2,H3,H4|T4]}
    3 --> 3
    concReversed([H6|T6],L6,R6) :- concReversed(T6,[H6|L6],R6).
```

```
    {T4 = [H6|T6], L6 = [H4,H3,H2], R6 = [1,2,3]}

[5] concReversed(T6,[H6,H4,H3,H2],[1,2,3])
    {L = [H2,H3,H4,H6|T6]}
    1 --> 3
    concReversed([H7|T7],L7,R7) :- concReversed(T7,[H7|L7],R7).
    {T6 = [H7|T7], L7 = [H6,H4,H3,H2], R7 = [1,2,3]}

[6] concReversed(T7,[H7,H6,H4,H3,H2],[1,2,3])
    {L = [H2,H3,H4,H6,H7|T6]}
    1 --> 3
    concReversed([H8|T8],L8,R8) :- concReversed(T8,[H8|L8],R8).
    {T7 = [H8|T8], L8 = [H7,H6,H4,H3,H2], R8 = [1,2,3]}

[7] concReversed(T8,[H8,H7,H6,H4,H3,H2],[1,2,3])
    {L = [H2,H3,H4,H6,H7,H8|T6]}

...
```

Παρατηρούμε ότι για να απαντήσει στη δεύτερη ερώτηση, η Prolog παράγει τη γενική μορφή λίστας μήκους n αυξάνοντας συνεχώς το n και τη συγκρίνει με τη δεδομένη λίστα. Μετά την πρώτη επιτυχία συνεχίζει να παράγει λίστες μεγαλύτερου μήκους, συνεπώς η ταυτοποίηση με τη δεδομένη λίστα αποτυγχάνει συνεχώς. ∎

Παράδειγμα 33: ταξινόμηση λίστας με εισαγωγή.

Θέλουμε να ορίσουμε το κατηγόρημα `insSort(L,S)` το οποίο αληθεύει αν η `S` είναι η ταξινομημένη εκδοχή της `L`, δηλαδή αν περιέχει τα ίδια στοιχεία με την `L` με ίδιο αριθμό εμφανίσεων το καθένα, σε αύξουσα σειρά ως προς τον ενσωματωμένη διάταξη όρων της Prolog.

Ο ορισμός στηρίζεται στην παρακάτω παρατήρηση: η ταξινομημένη εκδοχή της λίστας `[H|T]` είναι η λίστα που προκύπτει με εισαγωγή της κεφαλής `H` στην κατάλληλη θέση της ταξινομημένης εκδοχής της ουράς `T`. Για τον ορισμό χρειαζόμαστε το κατηγόρημα `insert(N,P,R)` το οποίο αληθεύει αν η λίστα `R` προκύπτει με εισαγωγή του `N` ακριβώς πρίν από το πρώτο στοιχείο της `P` που δεν είναι μικρότερό του, ή στο τέλος της `P` αν όλα τα στοιχεία της είναι μικρότερά του. Ειδικότερα αν η λίστα `P` είναι ταξινομημένη τότε και η `R` είναι ταξινομημένη.

```
insert(N,[],[N]).
insert(N,[H|T],[N,H|T]) :- N @=< H.
insert(N,[H|T],[H|S]) :- N @> H,
                         insert(N,T,S).

insSort([H|T],S) :- insSort(T,R),
                    insert(H,R,S).
insSort([],[]).
```

Οι ενσωματωμένοι τελεστές σύγκρισης `@=<` και `@>` παίρνουν ως ορίσματα οποιουσδήποτε όρους. Αν ωστόσο οι όροι περιέχουν ελεύθερες μεταβλητές η οποίες θα δεσμευτούν σε

τιμές μετά τη σύγκριση, τα αποτελέσματα μπορεί να είναι μη αναμενόμενα (καθώς η σύγκριση θα γίνει για τα ονοματα των μεταβλητών και όχι τις τιμές που θα πάρουν, οι οποίες δεν είναι γνωστές όταν γίνεται η σύγκριση). Για το λόγο αυτό το πρώτο όρισμα της `insSort` θα πρέπει να είναι λίστα τα στοιχεία της οποίας θα είναι όροι χωρίς ελεύθερες μεταβλητές. Η ερώτηση

```
?- insSort([5,3,0],S).
```

μπορεί να χρησιμοποιηθεί για ταξινόμηση της λίστας. Η Prolog για να απαντήσει ακολουθεί τα παρακάτω βήματα:

```
[0] insSort([5,3,0],S)
    {S = S}
    1 --> 4
    insSort([H1|T1],S1) :- insSort(T1,R1), insert(H1,R1,S1).
    {H1 = 5, T1 = [3,0], S1 = S}

[1] insSort([3,0],R1), insert(5,R1,S)
    {S = S}
    1 --> 4
    insSort([H2|T2],S2) :- insSort(T2,R2), insert(H2,R2,S2).
    {H2 = 3, T2 = [0], S2 = R1}

[2] insSort([0],R2), insert(3,R2,R1), insert(5,R1,S)
    {S = S}
    1 --> 4
    insSort([H3|T3],S3) :- insSort(T3,R3), insert(H3,R3,S3).
    {H3 = 0, T3 = [], S3 = R2}

[3] insSort([],R3), insert(0,R3,R2), insert(3,R2,R1), insert(5,R1,S)
    {S = S}
    1 --> 5
    insSort([],[]).
    {R3 = []}

[4] insert(0,[],R2), insert(3,R2,R1), insert(5,R1,S)
    {R = R}
    1 --> 1
    insert(N4,[],[N4]).
    {R2 = [0], N4 = 0}

[5] insert(3,[0],R1), insert(5,R1,S)
    {S = S}
    1 --> 2
    insert(N5,[H5|T5],[N5,H5|T5]) :- N5 @=< H5.
    {R1 = [3,0], N5 = 3, H5 = 0, T5 = []}

[6] 3 @=< 0, insert(5,[3,0],S)
    {S = S}
    @=< : build-in
```

```
    failure - backtracking

 [5] insert(3,[0],R1), insert(5,R1,S)
    {S = S}
    3 --> 3
    insert(N6,[H6|T6],[H6|S6]) :- N6 @> H6, insert(N6,T6,S6).
    {R1 = [0|S6], N6 = 3, H6 = 0, T6 = []}

 [6] 3 @> 0, insert(3,[],S6), insert(5,[0|S6],S)
    {S = S}
    @> : build-in
    {}

 [7] insert(3,[],S6), insert(5,[0|S6],S)
    {S = S}
    1 --> 1
    insert(N7,[],[N7]).
    {S6 = [3], N7 = 3}

 [8] insert(5,[0,3],S)
    {S = S}
    1 --> 2
    insert(N8,[H8|T8],[N8,H8|T8]) :- N8 @=< H8.
    {S = [5,0,3], N8 = 5, H8 = 0, T8 = [3]}

 [9] 5 @=< 0
    {S = [5,0,3]}
    @=< : build-in
    failure - backtracking

 [8] insert(5,[0,3],S)
    {S = S}
    3 --> 3
    insert(N9,[H9|T9],[H9|S9]) :- N9 @> H9, insert(N9,T9,S9).
    {S = [0|S9], N9 = 5, H9 = 0, T9 = [3]}

 [9] 5 @> 0, insert(5,[3],S9)
    {S = [0|S9]}
    @> : build-in
    {}

[10] insert(5,[3],S9)
    {S = [0|S9]}
    1 --> 2
    insert(N10,[H10|T10],[N10,H10|T10]) :- N10 @=< H10.
    {S9 = [5,3], N10 = 5, H10 = 3, T10 = []}

[11] 5 @=< 3
    {S = [0,5,3]}
```

```
     @=< : build-in
     failure - backtracking

[10] insert(5,[3],S9)
     {S = [0|S9]}
     3 --> 3
     insert(N11,[H11|T11],[H11|S11]) :- N11 @> H11, insert(N11,T11,S11).
     {S9 = [3|S11], N11 = 5, H11 = 3, T11 = []}

[11] 5 @> 3, insert(5,[],S11)
     {S = [0,3|S11]}
     @> : build-in
     {}

[12] insert(5,[],S11)
     {S = [0,3|S11]}
     1 --> 1
     insert(N12,[],[N12]).
     {S11 = [5], N12 = 5}

[13] #
     {S = [0,3,5]}
-------------------------------------------------------------
S = [0,3,5]
```

Μπορούμε να ταξινομήσουμε λίστες με διάφορα είδη όρων

```
?- insSort([2,a,chris,n,g(0),f(1,2),1],L).
L = [1,2,a,chris,n,g(0),f(1,2)]
```

Παράδειγμα 34: ταξινόμηση λίστας με συγχώνεση.

Θα ορίσουμε το κατηγόρημα `mergeSort` το οποίο είναι λογικά ισοδύναμο με το `insSort`, ωστόσο ορίζεται με διαφορετικό τρόπο, που στηρίζεται στον αλγόριθμο ταξινόμησης με συγχώνευση. Έστω μία λίστα `L`. Αν η `L` περιέχει το πολύ ένα στοιχείο τότε είναι ήδη ταξινομημένη. Σε αντίθετη περίπτωση αν `L1` και `L2` είναι οι λίστες που περιέχουν τα στοιχεία που βρίσκονται αντίστοιχα στις περιττές και τις άρτιες θέσεις της `L`, `S1` είναι η ταξινομημένη εκδοχή της `L1`, `S2` είναι η ταξινομημένη εκδοχή της `L2` και `S` είναι η λίστα που προκύπτει με συγχώνευση των `S1` και `S2`, τότε η `S` αποτελεί και την ταξινομημένη εκδοχή της `L`.

Ο ορισμός του `mergeSort` χρησιμοποιεί δύο βοηθητικά κατηγορήματα `split` και `merge`. Το `split(R,R1,R2)` αληθεύει αν η `R1` αποτελείται από τα στοιχεία της `R` που βρίσκονται σε περιττές θέσεις και η `R2` αποτελείται από τα στοιχεία της `R` που βρίσκονται σε άρτιες θέσεις. Το `merge(Q1,Q2,Q)` στην περίπτωση που οι `Q1` και `Q2` είναι ταξινομημένες λίστες, αληθεύει αν η `Q` προκύπτει από τη συγχώνευσή τους, δηλαδή περιέχει τα στοιχεία και των δύο λιστών και είναι επίσης ταξινομημένη. Σημειώνεται ότι το `merge(Q1,Q2,Q)` μπορεί να αληθεύει και σε περιπτώσεις που κάποια από τις `Q1` ή `Q2` δεν είναι ταξινομημένη, αλλά αυτή η περίπτωση μας είναι αδιάφορη σε ότι αφορά τον ορισμό του `mergeSort`.

```
merge([H1|T1],[H2|T2],[H1|L]) :-
       H1 @=< H2, merge(T1,[H2|T2],L).
merge([H1|T1],[H2|T2],[H2|L]) :-
       H1 @> H2, merge([H1|T1],T2,L).
merge(L,[],L).
merge([],L,L).

split([A,B|T],[A|L1],[B|L2]) :- split(T,L1,L2).
split([X],[X],[]).
split([],[],[]).


mergeSort([A,B|T],S) :- split([A,B|T],L1,L2),
                        mergeSort(L1,S1),
                        mergeSort(L2,S2),
                        merge(S1,S2,S).
mergeSort([X],[X]).
mergeSort([],[]).
```

Οπως έχουμε αναφέρει το κατηγόρημα `mergeSort` είναι ισοδύναμο με το `insSort`:

```
?- mergeSort([2,a,chris,n,g(0),f(1,2),1],L).
L = [1,2,a,chris,n,g(0),f(1,2)]
```

∎

Παράδειγμα 35: μετάθεση λίστας.

Θέλουμε να ορίσουμε ένα κατηγορημα `perm(L,P)` το οποίο θα αληθεύει αν `L` και `P` είναι λίστες και η `P` αποτελεί μετάθεση της `L`, δηλαδή αν περιέχει ακριβώς τα ίδια στοιχεία με την `L` με ίδιο αριθμό εμφανίσεων το καθένα, ενδεχομένως σε διαφορετική σειρά. Θα ορίσουμε δύο υλοποιήσεις `perm1` και `perm2` του παραπάνω κατηγορήματος.

Ο ορισμός του `perm1(L,P)` στηρίζεται στην παρακάτω παρατήρηση: Έστω ότι `L` είναι της μορφής `[H|T]`. Τότε η `P` θα περιέχει σε κάποια θέση το `H`. Επιπλέον αν το `H` διαγραφεί από την `P`, η λίστα `Q` που θα προκύψει θα περιέχει όλα τα στοιχεία της `T` ενδεχομένως σε διαφορετική σειρά, άρα θα είναι μία μετάθεση της `T`. Συνεπώς υπάρχει μία μετάθεση `Q` της ουράς `T`, η οποία προκύπτει με διαγραφή της κεφαλής `H` από την `P`. Σημειώνεται επίσης ότι η μοναδική μετάθεση της κενής λίστας είναι η κενή λίστα.

```
perm1([],[]).
perm1([H|T],P) :- perm1(T,Q),
                  delete(H,P,Q).
```

Ο ορισμός του `perm2(L,P)` από την άλλη στηρίζεται σε μία διαφορετική παρατήρηση: Έστω ότι η `P` είναι της μορφής `[X|R]`. Τότε το `X` είναι στοιχείο της `L` και αν διαγραφεί από την `L` προκύπτει μία λίστα `S` που έχει ακριβώς τα ίδια στοιχεία με την `R` ενδεχομένως σε διαφορετική σειρά. Συνεπώς η `R` είναι μετάθεση της `S`. Στην περίπτωση αυτή προφανώς η λίστα `L` δεν είναι κενή, άρα είναι της μορφής `[H|T]`.

```
perm2([],[]).
perm2([H|T],[X|R]) :- delete(X,[H|T],S),
                      perm2(S,R).
```

Για να βρούμε τις μεταθέσεις της λίστας [a,b,c] κάνουμε μία από τις παρακάτω ερωτήσεις (δίνοντας ; μετά από κάθε απάντηση):

```
?- perm1([a,b,c],P).
P = [a,b,c] ;
P = [b,a,c] ;
P = [b,c,a] ;
P = [a,c,b] ;
P = [c,a,b] ;
P = [c,b,a] ;
no

?- perm2([a,b,c],P).
P = [a,b,c] ;
P = [a,c,b] ;
P = [b,a,c] ;
P = [b,c,a] ;
P = [c,a,b] ;
P = [c,b,a] ;
no
```

Η Prolog για να απαντήσει στην πρώτη ερώτηση θα βρεί τις μεταθέσεις της ουράς [b,c] και θα εισάγει σε κάθε μία από αυτές το στοιχείο a σε όλες τις πιθανές θέσεις. Η πρώτη μετάθεση της [b,c] είναι η ίδια η [b,c], συνεπώς οι τρεις πρώτες τιμές της P που επιστρέφει η Prolog είναι οι [a,b,c], [b,a,c] και [b,c,a]. Στη συνέχεια από την εισαγωγή του a στην [c,b] προκύπτουν οι τρείς επόμενες τιμές [a,c,b], [c,a,b] και [c,b,a]. Αν ζητήσουμε και άλλη απάντηση η Prolog θα επιστρέψει no.

Η Prolog για να απαντήσει στην δεύτερη ερώτηση θα διαγράψει πρώτα το στοιχείο a από την [a,b,c], θα βρει τις μεταθέσεις της λίστας [b,c] που είναι κατα σειρά η [b,c] και [c,b] και θα επιστρέψει ως απαντήσεις τις λίστες που προκύπτουν με την προσθήκη του a ως καφαλή στις μεταθέσεις αυτές, δηλαδή τις [a,b,c] και [a,c,b]. Στη συνέχεια θα διαγράψει το b από την [a,b,c], θα βρει τις μεταθέσεις της λίστας [a,c] και θα επιστρέψει τις μεταθέσεις [b,a,c] και [b,c,a]. Τέλος θα διαγράψει το c από την [a,b,c], θα βρει τις μεταθέσεις της λίστας [a,b] και θα επιστρέψει τις μεταθέσεις [b,a,c] και [b,c,a]. Αν ζητήσουμε και άλλη απάντηση η Prolog θα επιστρέψει no

Παρατηρούμε ότι οι ερωτήσεις είναι ισοδύναμες, ωστόσο οι απαντήσεις σχηματίζονται με διαφορετική σειρά. Στη συνέχεια δίνονται αναλυτικά τα βήματα που εκτελεί η Prolog για να απαντήσει σε κάθε μία από τις παραπάνω ερωτήσεις. Ακολουθίες βημάτων οι οποίες είναι παρόμοιες με προηγούμενες παραλείπονται (σημειώνονται με ...). Οι αριθμοί 1 και 2 αναφέρονται στις προτάσεις του ορισμού του κατηγορήματος perm1 ή perm2, ενώ οι αριθμοί 3 και 4 στις προτάσεις του ορισμού του delete. Για την πρώτη ερώτηση τα βήματα είναι τα παρακάτω:

```
[0] perm1([a,b,c],P)
    {P = P}
    1 --> 2
    perm1([H1|T1],P1) :- perm1(T1,Q1),delete(H1,P1,Q1).
    {H1 = a, T1 = [b,c], P1 = P}
```

```
[1] perm1([b,c],Q1),delete(a,P,Q1)
    {P = P}
    1 --> 2
    perm1([H2|T2],P2) :- perm1(T2,Q2),delete(H2,P2,Q2).
    {H2 = b, T2 = [c], P2 = Q1}

[2] perm1([c],Q2),delete(b,Q1,Q2),delete(a,P,Q1)
    {P = P}
    1 --> 2
    perm1([H3|T3],P3) :- perm1(T3,Q3),delete(H3,P3,Q3).
    {H3 = c, T3 = [], P3 = Q2}

[3] perm1([],Q3),delete(c,Q2,Q3),delete(b,Q1,Q2),delete(a,P,Q1)
    {P = P}
    1 --> 1
    perm1([],[]).
    {Q3 = c}

[4] delete(c,Q2,[]),delete(b,Q1,Q2),delete(a,P,Q1)
    {P = P}
    1 --> 3
    delete(X4,[X4|T4],T4).
    {Q2 = [c], X4 = c, T4 = []}

[5] delete(b,Q1,[c]),delete(a,P,Q1)
    {P = P}
    1 --> 3
    delete(X5,[X5|T5],T5).
    {Q1 = [b,c], X5 = b, T5 = [c]}

[6] delete(a,P,[b,c])
    {P = P}
    1 --> 3
    delete(X6,[X6|T6],T6).
    {P = [a,b,c], X6 = a, T6 = [b,c]}

[7] #
    {P = [a,b,c]}
------------------------------------------------------------
P = [a,b,c] ;
------------------------------------------------------------
[6] delete(a,P,[b,c])
    {P = P}
    4 --> 4
    delete(X7,[H7|T7],[H7|S7]) :- delete(X7,T7,S7).
    {P = [b|T7], X7 = a, H7 = b, S7 = [c]}

[7] delete(a,T7,[c])
```

```
    {P = [b|T7]}
    1 --> 3
    delete(X8,[X8|T8],T8).
    {T7 = [a,c], X8 = a, T8 = [c]}

[8] #
    {P = [b,a,c]}
-----------------------------------------------------------------
P = [b,a,c] ;
-----------------------------------------------------------------
[7] delete(a,T7,[c])
    {P = [b|T7]}
    4 --> 4
    delete(X9,[H9|T9],[H9|S9]) :- delete(X9,T9,S9).
    {T7 = [c|T9], X9 = a, H9 = c, S9 = []}

[8] delete(a,T9,[])
    {P = [b,c|T9]}
    1 --> 3
    delete(X10,[X10|T10],T10).
    {T9 = [a], X10 = a, T10 = []}
    {P = [b,c,a]}

[9] #
-----------------------------------------------------------------
P = [b,c,a] ;
-----------------------------------------------------------------
[8] delete(a,T9,[])
    {P = [b,c|T9]}
    4 --> EOP
    failure - backtracking

[7] delete(a,T7,[c])
    {P = [b|T7]}
    EOP
    failure - backtracking

[6] delete(a,P,[b,c])
    {P = P}
    EOP
    failure - backtracking

[5] delete(b,Q1,[c]),delete(a,P,Q1)
    {P = P}
    4 --> 4
    delete(X11,[H11|T11],[H11|S11]) :- delete(X11,T11,S11).
    {Q1 = [c|T11], X11 = b, H11 = c, T11 = []}

[6] delete(b,T11,[]),delete(a,P,[c|T11])
```

```
    {P = P}
    1 --> 3
    delete(X12,[X12|T12],T12).
    {T11 = [b], X12 = b, T12 = []}

[7] delete(a,P,[c,b])
    {P = P}
...
------------------------------------------------------------
P = [a,c,b] ;
------------------------------------------------------------
...
------------------------------------------------------------
P = [c,a,b] ;
------------------------------------------------------------
...
------------------------------------------------------------
P = [c,b,a] ;
------------------------------------------------------------
...
[5] delete(b,Q1,[c]),delete(a,P,Q1)
    {P = P}
    EOP
    failure - backtracking

[4] delete(c,Q2,[]),delete(b,Q1,Q2),delete(a,P,Q1)
    {P = P}
    4 --> EOP
    failure - backtracking

[3] perm1([],Q3),delete(c,Q2,Q3),delete(b,Q1,Q2),delete(a,P,Q1)
    {P = P}
    2 --> EOP
    failure - backtracking

[2] perm1([c],Q2),delete(b,Q1,Q2),delete(a,P,Q1)
    {P = P}
    3 --> EOP
    failure - backtracking

[1] perm1([b,c],Q1),delete(a,P,Q1)
    {P = P}
    3 --> EOP
    failure - backtracking

[0] perm1([a,b,c],P)
    {P = P}
    3 --> EOP
    failure
```

```
----------------------------------------------------------
no
```

Τα βήματα που ακολουθέι η Prolog για να απαντήσει στη δεύτερη ερώτηση είναι τα παρακάτω:

```
[0] perm2([a,b,c],P)
    {P = P}
    1 --> 2
    perm2([H1|T1],[X1|R1]) :- delete(X1,[H1|T1],S1),perm2(S1,R1).
    {P = [X1|R1], H1 = a, T1 = [b,c]}

[1] delete(X1,[a,b,c],S1),perm2(S1,R1)
    {P = [X1|R1]}
    1 --> 3
    delete(X2,[X2|T2],T2).
    {X1 = a, S1 = [b,c], X2 = a, T2 = [b,c]}

[2] perm2([b,c],R1)
    {P = [a|R1]}
    1 --> 2
    perm2([H3|T3],[X3|R3]) :- delete(X3,[H3|T3],S3),perm2(S3,R3).
    {R1 = [X3|R3], H3 = b, T3 = [c]}

[3] delete(X3,[b,c],S3),perm2(S3,R3)
    {P = [a,X3|R3]}
    1 --> 3
    delete(X4,[X4|T4],T4).
    {X3 = b, S3 = [c], X4 = b, T4 = [c]}

[4] perm2([c],R3)
    {P = [a,b|R3]}
    1 --> 2
    perm2([H5|T5],[X5|R5]) :- delete(X5,[H5|T5],S5),perm2(S5,R5).
    {R3 = [X5|R5], H5 = c, T5 = []}

[5] delete(X5,[c],S5),perm2(S5,R5)
    {P = [a,b,X5|R5]}
    1 --> 3
    delete(X6,[X6|T6],T6).
    {X5 = c, S5 = [], X6 = c, T6 = []}

[6] perm2([],R5)
    {P = [a,b,c|R5]}
    1 --> 1
    perm2([],[]).
    {R5 = []}

[7] #
    {P = [a,b,c]}
```

```
------------------------------------------------------------
P = [a,b,c] ;
------------------------------------------------------------
[6] perm2([],R5)
    {P = [a,b,c|R5]}
    2 --> EOP
    failure - backtracking

[5] delete(X5,[c],S5),perm2(S5,R5)
    {P = [a,b,X5|R5]}
    4 --> 4
    delete(X7,[H7|T7],[H7|S7]) :- delete(X7,T7,S7).
    {X7 = X5, H7 = c, T7 = [], S5 = [c|S7]}

[6] delete(X5,[],[c|S7]),perm2([c|S7],R5)
    {P = [a,b,X5|R5]}
    1 --> EOP
    failure - backtracking

[5] delete(X5,[c],S5),perm2(S5,R5)
    {P = [a,b,X5|R5]}
    EOP
    failure - backtracking

[4] perm2([c],R3)
    {P = [a,b|R3]}
    3 --> EOP
    failure - backtracking

[3] delete(X3,[b,c],S3),perm2(S3,R3)
    {P = [a,X3|R3]}
    4 -> 4
    delete(X8,[H8|T8],[H8|S8]) :- delete(X8,T8,S8).
    {X8 = X3, H8 = b, T8 = [c], S3 = [b|S8]}

[4] delete(X3,[c],S8),perm2([b|S8],R3)
    {P = [a,X3|R3]}
    1 -> 3
    delete(X9,[X9|T9],T9).
    {X3 = c, S8 = [], X9 = c, T9 = []}

[5] perm2([b],R3)
    {P = [a,c|R3]}
...
------------------------------------------------------------
P = [a,c,b] ;
------------------------------------------------------------
...
[3] delete(X3,[b,c],S3),perm2(S3,R3)
```

```
    {P = [a,X3|R3]}
    EOP
    failure - backtracking

[2] perm2([b,c],R1)
    {P = [a|R1]}
    3 --> EOP
    failure - backtracking

[1] delete(X1,[a,b,c],S1),perm2(S1,R1)
    {P = [X1|R1]}
    4 --> 4
    delete(X13,[H13|T13],[H13|S13]) :- delete(X13,T13,S13).
    {X13 = X1, H13 = a, T13 = [b,c], S1 = [a|S13]}

[2] delete(X1,[b,c],S13),perm2([a|S13],R1)
    {P = [X1|R1]}
    1 --> 3
    delete(X14,[X14|T14],T14).
    {X1 = b, S13 = [c], X14 = b, T14 = [c]}

[3] perm2([a,c],R1)
    {P = [b|R1]}
...
------------------------------------------------------------
P = [b,a,c] ;
------------------------------------------------------------
...
------------------------------------------------------------
P = [b,c,a] ;
------------------------------------------------------------
...

[4] perm2([a,b],R1)
    {P = [c|R1]}
...
------------------------------------------------------------
P = [c,a,b] ;
------------------------------------------------------------
...
------------------------------------------------------------
P = [c,b,a] ;
------------------------------------------------------------
...
    failure - backtracking

[1] delete(X1,[a,b,c],S1),perm2(S1,R1)
    {P = [X1|R1]}
    EOP
```

```
    failure - backtracking

[0] perm2[a,b,c],P)
    {P = P}
    3 --> EOP
    failure
-----------------------------------------------------------------
no
```

Παρατηρούμε ότι τα κατηγορήματα `perm1` και `perm2` είναι συμμετρικά: αν η `P` είναι μετάθεση της `L` τότε και η `L` είναι μετάθεση της `P`.

Έστω ότι επιχειρούμε να βρούμε τις μεταθέσεις της λίστας `[a,b,c]` κάνοντας την ερώτηση

```
?- perm1(P,[a,b,c]).
```

Η Prolog για να απαντήσει στην ερώτηση αυτή σχηματίζει το στόχο

```
perm1(T1,Q1),delete(H1,[a,b,c],Q1)
```

Για να ικανοποιηθεί αυτός ο στόχος θα πρέπει να βρεθεί μία λίστα `T1` και μία μετάθεσή της `Q1`, τέτοιες ώστε η `Q1` να προκύπτει από την δεδομένη λίστα `[a,b,c]` με διαγραφή κάποιου στοιχείου. Σημειώνεται ότι κατά την αναζήτηση των `T1` και `Q1` δεν χρησιμοποιείται καμία πληροφορία σχετική με τη δεδομένη λίστα. Η Prolog κατασκευάζει λίστες διαρκώς αυξανόμενου μεγέθους με στοιχεία ελεύθερες μεταβλητές, σχηματίζει τις μεταθέσεις τους και ελέγχει αν κάποια από αυτές μπορεί να προκύψει με διαγραφή στοιχείου από την `[a,b,c]`. Αυτό θα πετύχει όταν η λίστα `T1` έχει μήκος 2, οπότε από τις δύο μεταθέσεις αυτής της και τους τρεις τρόπους διαγραφής στοιχείου από την `[a,b,c]` θα σχηματισούν οι έξι μεταθέσεις της `[a,b,c]` (με διαφορετική σειρά σε σχέση με τις προηγούμενες ερωτήσεις). Αν ωστόσο ζητήσουμε μία ακόμη λύση, η Prolog θα συνεχίσει να κατασκευάζει όλο και μεγαλύτες λίστες, χωρίς να μπορέσει να διαπιστώσει ότι αυτό δεν οδηγεί σε άλλη απάντηση ώστε να επιστρέψει `no`. Έτσι τελικά θα προκληθεί υπερχείλιση στοίβας.

```
[0] perm1(P,[a,b,c])
    {P = P}
    1 --> 2
    perm1([H1|T1],P1) :- perm1(T1,Q1),delete(H1,P1,Q1).
    {P = [H1|T1], P1 = [a,b,c]}

[1] perm1(T1,Q1),delete(H1,[a,b,c],Q1)
    {P = [H1|T1]}
    1 --> 1
    perm1([],[]).
    {T1 = [], Q1 = []}

[2] delete(H1,[a,b,c],[])
    {P = [H1]}
    1 --> EOP
    failure - backtracking
```

```
[1] perm1(T1,Q1),delete(H1,[a,b,c],Q1)
    {P = [H1|T1]}
    2 --> 2
    perm1([H2|T2],P2) :- perm1(T2,Q2),delete(H2,P2,Q2).
    {T1 = [H2|T2], P2 = Q1}

[2] perm1(T2,Q2),delete(H2,Q1,Q2),delete(H1,[a,b,c],Q1)
    {P = [H1,H2|T2]}
    1 --> 1
    perm1([],[]).
    {T2 = [], Q2 = []}

[3] delete(H2,Q1,[]),delete(H1,[a,b,c],Q1)
    {P = [H1,H2]}
    3 --> 3
    delete(X3,[X3|T3],T3).
    {Q1 = [H2], X3 = H2, T3 = []}

[4] delete(H1,[a,b,c],[H2])
    {P = [H1,H2]}
    1 --> 4
    delete(X4,[H4|T4],[H4|S4]) :- delete(X4,T4,S4).
    {H2 = a, X4 = H1, H4 = a, T4 = [b,c], S4 = []}

[5] delete(H1,[b,c],[])
    {P = [H1,a]}
    1 --> EOP
    failure - backtracking

[4] delete(H1,[a,b,c],[H2])
    {P = [H1,H2]}
    EOP
    failure - backtracking

[3] delete(H2,Q1,[]),delete(H1,[a,b,c],Q1)
    {P = [H1,H2]}
    4 --> EOP
    failure - backtracking

[2] perm1(T2,Q2),delete(H2,Q1,Q2),delete(H1,[a,b,c],Q1)
    {P = [H1,H2|T2]}
    2 --> 2
    perm1([H5|T5],P5) :- perm1(T5,Q5),delete(H5,P5,Q5).
    {T2 = [H5|T5], P5 = Q2}

[3] perm1(T5,Q5),delete(H5,Q2,Q5),delete(H2,Q1,Q2),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5|T5]}
    1 --> 1
    perm1([],[]).
```

```
    {T5 = [], Q5 = []}

[4] delete(H5,Q2,[]),delete(H2,Q1,Q2),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5]}
    1 --> 3
    delete(X6,[X6|T6],T6).
    {Q2 = [H5], X6 = H5, T6 = []}

[5] delete(H2,Q1,[H5]),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5]}
    1 --> 3
    delete(X7,[X7|T7],T7).
    {Q1 = [H2,H5], X7 = H2, T7 = [H5]}

[6] delete(H1,[a,b,c],[H2,H5])
    {P = [H1,H2,H5]}
    1 --> 3
    delete(X8,[X8|T8],T8).
    {H1 = a, H2 = b, H5 = c, X8 = a, T8 = [b,c]}

[7] #
    {P = [a,b,c]}
------------------------------------------------------------
P = [a,b,c] ;
------------------------------------------------------------
[6] delete(H1,[a,b,c],[H2,H5])
    {P = [H1,H2,H5]}
    4 --> 4
    delete(X9,[H9|T9],[H9|S9]) :- delete(X9,T9,S9).
    {H2 = a, X9 = H1, H9 = a, T9 = [b,c], S9 = [H5]}

[7] delete(H1,[b,c],[H5])
    {P = [H1,a,H5]}
    1 --> 3
    delete(X10,[X10|T10],T10).
    {H1 = b, H5 = c, X10 = b, T10 = [c]}

[8] #
    {P = [b,a,c]}
------------------------------------------------------------
P = [b,a,c] ;
------------------------------------------------------------
[7] delete(H1,[b,c],[H5])
    {P = [H1,a,H5]}
    4 --> 4
    delete(X11,[H11|T11],[H11|S11]) :- delete(X11,T11,S11).
    {H5 = b, X11 = H1, H11 = b, T11 = [c], S11 = []}

[8] delete(H1,[c],[])
```

```
    {P = [H1,a,b]}
    1 --> 3
    delete(X12,[X12|T12],T12).
    {H1 = c, X12 = c, T12 = []}

[9] #
    {P = [c,a,b]}
-------------------------------------------------------------
P = [c,a,b] ;
-------------------------------------------------------------
[8] delete(H1,[c],[])
    {P = [H1,a,b]}
    4 --> EOP
    failure - backtracking

[7] delete(H1,[b,c],[H5])
    {P = [H1,a,H5]}
    EOP
    failure - backtracking

[6] delete(H1,[a,b,c],[H2,H5])
    {P = [H1,H2,H5]}
    EOP
    failure - backtracking

[5] delete(H2,Q1,[H5]),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5]}
    4 --> 4
    delete(X13,[H13|T13],[H13|S13]) :- delete(X13,T13,S13).
    {X13 = H2, Q1 = [H5|T13], H13 = H5, S13 = []}

[6] delete(H2,T13,[]),delete(H1,[a,b,c],[H5|T13])
    {P = [H1,H2,H5]}
    1 --> 3
    delete(X14,[X14|T14],T14).
    {T13 = [H2], X14 = H2, T14 = []}

[7] delete(H1,[a,b,c],[H5,H2])
    {P = [H1,H2,H5]}
...
-------------------------------------------------------------
P = [a,c,b] ;
-------------------------------------------------------------
...
-------------------------------------------------------------
P = [b,c,a] ;
-------------------------------------------------------------
...
-------------------------------------------------------------
```

```
P = [c,b,a] ;
------------------------------------------------------------
...
    failure - backtracking

[5] delete(H2,Q1,[H5]),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5]}
    EOP
    failure - backtracking

[4] delete(H5,Q2,[]),delete(H2,Q1,Q2),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5]}
    4 --> EOP
    failure - backtracking

[3] perm1(T5,Q5),delete(H5,Q2,Q5),delete(H2,Q1,Q2),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5|T5]}
    2 --> 2
    perm1([H15|T15],P15) :- perm1(T15,Q15),delete(H15,P15,Q15).
    {T5 = [H15|T15], P15 = Q5}

[4] perm1(T15,Q15),delete(H15,Q5,Q15),delete(H5,Q2,Q5),delete(H2,Q1,Q2),delete(H1,[a,b,c]
    {P = [H1,H2,H5,H15|T15]}
    1 --> 1
    perm1([],[]).
    {T15 = [], Q15 = []}

[5] delete(H15,Q5,Q15),delete(H5,Q2,Q5),delete(H2,Q1,Q2),delete(H1,[a,b,c],Q1)
    {P = [H1,H2,H5,H15]}
...
```

Αν επιχειρήσουμε να βρούμε τις μεταθέσεις της λίστας `[a,b,c]` κάνοντας την ερώτηση

```
?- perm2(P,[a,b,c]).
```

τότε τα πράγματα είναι χειρότερα. Ο ορισμός του `perm2` οδηγεί την Prolog στο να αναζητήσει πρώτα λιστες που περιέχουν τα `a` και `b` ως αρχικά στοιχεία και το `c` να ακολουθεί σε μία από τις επόμενες θέσεις. Υπάρχει μόνο μία τέτοια μετάθεση (η `[a,b,c]`) αλλά άπειρες λίστες της παραπάνω μορφής, και η Prolog εγκλωβίζεται σε αυτές εκτελώντας μία ατέρμονη ακολουθία βημάτων (που θα οδηγήσει σε υπερχείλιση στοίβας) χωρίς να βρεί άλλη μετάθεση.

```
[0] perm2(P,[a,b,c])
    {P = P}
    1 --> 2
    perm2([H1|T1],[X1|R1]) :- delete(X1,[H1|T1],S1),perm2(S1,R1).
    {P = [H1|T1], X1 = a, R1 = [b,c]}
```

```
[1] delete(a,[H1|T1],S1),perm2(S1,[b,c])
    {P = [H1|T1]}
    1 --> 3
    delete(X2,[X2|T2],T2).
    {H1 = a, T1 = S1, X2 = a, T2 = S1}

[2] perm2(S1,[b,c])
    {P = [a|S1]}
    1 --> 2
    perm2([H3|T3],[X3|R3]) :- delete(X3,[H3|T3],S3),perm2(S3,R3).
    {S1 = [H3|T3], X3 = b, R3 = [c]}

[3] delete(b,[H3|T3],S3),perm2(S3,[c])
    {P = [a,H3|T3]}
    1 --> 3
    delete(X4,[X4|T4],T4).
    {H3 = b, T3 = S3, X4 = b, T4 = S3}

[4] perm2(S3,[c])
    {P = [a,b|S3]}
    1 --> 2
    perm2([H5|T5],[X5|R5]) :- delete(X5,[H5|T5],S5),perm2(S5,R5).
    {S3 = [H5|T5], X5 = c, R5 = []}

[5] delete(c,[H5|T5],S5),perm2(S5,[])
    {P = [a,b,H5|T5]}
    1 --> 3
    delete(X6,[X6|T6],T6).
    {H5 = c, T5 = S5, X6 = c, T6 = S5}

[6] perm2(S5,[])
    {P = [a,b,c|S5]}
    1 --> 1
    perm2([],[]).
    {S5 = []}

[7] #
    {P = [a,b,c]}
------------------------------------------------------------
P = [a,b,c] ;
------------------------------------------------------------
[6] perm2(S5,[])
    {P = [a,b,c|S5]}
    2 --> EOP
    failure - backtracking

[5] delete(c,[H5|T5],S5),perm2(S5,[])
    {P = [a,b,H5|T5]}
    4 --> 4
```

```
    delete(X7,[H7|T7],[H7|S7]) :- delete(X7,T7,S7).
    {S5 = [H5|S7], X7 = c, H7 = H5, T7 = T5}

[6] delete(c,T5,S7),perm2([H5|S7],[])
    {P = [a,b,H5|T5]}
    1 -->3
    delete(X8,[X8|T8],T8).
    {T5 = [c|S7], X8 = c, T8 = S7}

[7] perm2([H5|S7],[])
    {P = [a,b,H5,c|S7]}
    1 --> EOP
    failure - backtracking

[6] delete(c,T5,S7),perm2([H5|S7],[])
    {P = [a,b,H5|T5]}
    4 --> 4
    delete(X9,[H9|T9],[H9|S9]) :- delete(X9,T9,S9).
    {T5 = [H9|T9], S7 = [H9|S9], X9 = c}

[7] delete(c,T9,S9),perm2([H5,H9|S9],[])
    {P = [a,b,H5,H9|T9]}
    1 -->3
    delete(X10,[X10|T10],T10).
    {T9 = [c|S9], X10 = c, T10 = S9}

[8] perm2([H5,H9|S9],[])
    {P = [a,b,H5,H9,c|S9]}
    1 -->EOP
    failure - backtracking

...
```

Τα στοιχεία μίας λίστας ενδέχεται να είναι επίσης λίστες. Χρησιμοποιώντας λίστες αυτής της μορφής μπορούμε να παραστήσουμε πίνακες. Κάθε εσωτερική λίστα θα πρέπει να είναι μή κενή και αντιστοιχεί σε μία γραμμή του πίνακα. Όλες οι εσωτερικές λίστες θα πρέπει να έχουν το ίδιο πλήθος στοιχείων.

Παράδειγμα 36: Μοναδιαίος πίνακας.

Θα ορίσουμε το κατηγόρημα **unitMatrix(N,M)** το οποίο αληθεύει αν M είναι η λίστα από λίστες που παριστάνει τον μοναδιαίο N×N πίνακα. Ο ορισμός του **unitMatrix** στηρίζεται στην παρακάτω παρατήρηση: Έστω ότι N > 1 και ότι το K έχει την τιμή N-1. Ο μοναδιαίος πίνακας K×K προκύπτει από τον μοναδιαίο πίνακα N×N με διαγραφή της πρώτης γραμμής και της πρώτης στήλης. Συνεπώς, αν ο μοναδιαίος K×K παριστάνεται από τη λίστα S και ο ο μοναδιαίος N×N παριστάνεται από τη λίστα [H|T], τότε ισχύουν οι δύο παρακάτω συνθήκες: (α) κάθε στοιχείο (λίστα) της T προκύπτει με προσθήκη του 0 στην αρχή του αντίστοιχου στοιχείου της S (ελέγχεται από το βοηθητικό κατηγόρημα **expand**) και (β) το H είναι η λίστα που έχει ως κεφαλή το 1 και ως ουρά τη λίστα που αποτελείται

από K μηδενικά (ελέγχεται από από το βοηθητικό κατηγόρημα zeros).

```
unitMatrix(1,[[1]]).
unitMatrix(N,[[1|R]|T]) :- N > 0,
                           K is N-1,
                           unitMatrix(K,S),
                           expand(S,T),
                           zeros(K,R).

zeros(0,[]).
zeros(K,[0|T]) :- I is K-1, zeros(I,T).

expand([],[]).
expand([L|M],[[0|L]|R]) :- expand(M,R).
```

Για να βρουμε το μοναδιαίο 3×3 κάνουμε την ερώτηση:

```
?- unitMatrix(3,M).
M = [[1,0,0],[0,1,0],[0,0,1]]
```

1.17 Η σειρά των προτάσεων

Ο αλγόριθμος που χρησιμοποιεί η Prolog για να απαντήσει σε μία ερώτηση, εξετάζει τις προτάσεις που ορίζουν ένα κατηγόρημα με τη σειρά που εμφανίζονται στο πρόγραμμα (και η οποία διατηρείται κατά τη φόρτωση του προγράμματος στη βάση δεδομένων). Επίσης σε κάθε βήμα εξετάζεται η πρώτη ατομική πρόταση του στόχου, κάτι που σημαίνει ότι οι προτάσεις στο σώμα ενός κανόνα εξετάζονται από αριστερά προς τα δεξιά. Συνεπώς, η σειρά των προτασεων σε ένα πρόγραμμα είναι σημαντική, και αν αλλάξει ενδέχεται η συμπεριφορά της Prolog όταν απαντάει στην ίδια ερώτηση πριν και μετά την αλλαγή να είναι διαφορετική. Το ίδιο ισχύει και για τη σειρά των προτάσεων στο σώμα ενός κάνόνα ή σε μία ερώτηση.

Παράδειγμα 37: Έστω το παρακάτω πρόγραμμα:

```
p(0).
p(s(X)) :- p(X).
p(X) :- p(s(X)).
```

Αν κάνουμε την ερώτηση

```
?- p(s(s(0))).
```

η Prolog χρησιμοποιώντας τον κανόνα `p(s(X)) :- p(X).` θα σχηματίσει τους ενδιάμεσους στόχους `p(s(0))` και `p(0)`, και επειδή η πρόταση που αποτελεί τον τελευταίο στόχο υπάρχει στο πρόγραμμα, θα προκύψει ο κενός στόχος και θα επιστραφεί η απάντηση `yes`.

Αν τώρα εναλλάξουμε τη σειρά των δύο κανόνων στο πραπάνω πρόγραμμα, τότε η Prolog στην προσπάθειά της να απαντήσει στην ίδια ερώτηση θα σχηματίσει χρησιμοποιώντας τον κανόνα `p(X) :- p(s(X)).` μία άπειρη ακολουθία από στόχους

```
p(s(s(0)))
p(s(s(s(0))))
p(s(s(s(s(0)))))
...
```

που θα προκαλέσει υπερχείλιση μνήμης και δεν θα επιστρέψει απάντηση. ∎

Παράδειγμα 38: Έστω το παρακάτω πρόγραμμα:

```
p(X) :- q(X),p(s(X)).
```

Αν κάνουμε την ερώτηση

```
?- p(s(s(0))).
```

η Prolog θα σχηματίσει το στόχο `q(s(s(0))),p(s(s(s(0))))` ο οποίος προφανώς θα αποτύχει, καθώς δεν υπάρχει ορισμός για το `q`, και θα επιστρέψει την απάντηση `no`.

Ας υποθέσουμε τωρα ότι αλλάζουμε τη σειρά των δύο προτάσεων στο σώμα του κανόνα:

```
p(X) :-  p(s(X)),q(X).
```

Η Prolog για να απαντήσει στην ίδια ερώτηση θα σχηματίσει μία άπειρη ακολουθία από στόχους

```
p(s(s(0)))
p(s(s(s(0)))), q(s(s(0)))
p(s(s(s(s(0))))), q(s(s(s(0)))), q(s(s(0)))
...
```

που θα προκαλέσει υπερχείλιση μνήμης και δεν θα επιστρέψει απάντηση. ∎

Γενικά αν έχουμε δύο προγράμματα που δεν περιέχουν αποκοπή (την οποία θα εξατάσουμε παρακάτω) τέτοια ώστε το ένα να προκύπτει με μετάθεση των προτάσεων του άλλου η/και μετάθεση των προτάσεων στα σώματα των κανόνων και κάνουμε μία ερώτηση που δεν περιέχει μεταβλητές τότε θα συμβεί ένα από τα παρακάτω:

- Και για τα δύο προγράμματα θα επιστραφεί η ίδια απάντηση `yes` ή `no`.
- Για κανένα πρόγραμμα δεν θα επιστραφεί απάντηση.
- Για το ένα πρόγραμμα η απάντηση θα είναι `yes` και για το άλλο δεν θα επιστραφεί απάντηση.
- Για το ένα πρόγραμμα η απάντηση θα είναι `no` και για το άλλο δεν θα επιστραφεί απάντηση.

Σε καμία περίπτωση πάντως δεν θα επιστραφεί διαφορετική απάντηση (δηλαδή `yes` για το ένα πρόγραμμα και `no` για το άλλο). Αυτό μπορεί να συμβεί μόνο αν τα προγράμματα περιέχουν αποκοπή.

1.18 Αποκοπή

Η οπισθοδρόμηση αποτελεί βασικό χαρακτηριστικό της Prolog και γίνεται αυτόματα από τη γλώσσα. Ο αλγόριθμος απάντησης σε ερώτηση, δημιουργεί μία ακολουθία στόχων μέχρι να προκύψει ο κενός στόχος, οπότε και έχει βρει μία απάντηση. Αν ωστόσο σε κάποιο σημείο διαπιστώσει ότι δεν υπάρχει τρόπος για να σχηματίσει τον επόμενο στόχο, οπισθοδρομεί και σχηματίζει μία εναλλάκτική ακολουθία στόχων.

Παρότι ο χειρισμός της οπισθοδρόμησης από τη ίδια τη γλώσσα αποτελεί ένα από τα βασικά πλεονεκτήματα της Prolog, υπάρχουν περιπτώσεις στις οποίες ο προγραμματιστής γνωρίζει ότι η οπισθοδρόμηση δεν πρόκειται να οδηγήσει σε υπολογισμό μίας λύσης και οτι απλά θα σπαταλήσει χρόνο δημιουργώντας στόχους που τελικά θα αποτύχουν.

Παράδειγμα 39: Έστω το παρακάτω πρόγραμμα υπολογισμού δύναμης, το οποίο προκύπτει προσθέτωντας τη συνθήκη `K =\= 0` στους αναδρομικούς κανόνες του κατηγορήματος `power` του παραδείγματος 19.

Με αυτή την προσθήκη το πρόγραμμα επιστρέφει μία μόνο φορά τη σωστή απάντηση για μη αρνητικό εκθέτη (και εξακολουθεί να επιστρέφει λάθος αποτέλεσμα για αρνητικό εκθέτη).

```
power3(N,0,1).
power3(N,K,P) :- K =\= 0,
                 K mod 2 =:= 0,
                 L is K//2,
                 power3(N,L,R),
                 P is R*R.
power3(N,K,P) :- K =\= 0,
                 K mod 2 =:= 1,
                 L is K//2,
                 power3(N,L,R),
                 P is R*R*N.
```

Ας υποθέσουμε ότι κάνουμε την ερώτηση

```
?- power3(2,0,P), P > 2.
```

Η Prolog θα υπολογίσει την τιμή `P = 1`, για την οποία δεν αληθεύει η δεύτερη πρόταση του στόχου. Συνεπώς η Prolog θα οπισθοδρομήσει και θα προσπαθήσει να βρεί και άλλες τιμές του `P`, χρησιμοποιώντας τους δύο αναδρομικούς κανόνες. Με αυτόν τον τρόπο θα σπαταλήσει χρόνο για να σχηματίσει δύο νέους στόχους και για να ελέγξει δύο φορές τη συνθήκη `0 =\= 0` που δεν είναι αληθής. Αυτό οφείλεται στο ότι η Prolog δεν γνωρίζει ότι το κατηγόρημα `power3` υπολογίζει μία συνάρτηση, κάτι που γνωρίζει ο προγραμματιστής αλλά δεν φαίνεται στον ορισμό του κατηγορήματος. ∎

Η Prolog παρέχει έναν μηχανισμό με τον οποίο ο προγραμματιστής μπορεί να ελέγξει την οπισθοδρόμηση και να την αποτρέψει σε περιπτώσεις που είναι άσκοπη. Ο έλεγχος της οπισθοδρόμησης γίνεται με χρήση της αποκοπής που συμβολίζεται με !. Το ! μπορεί να τοποθετηθεί στο σώμα ενός κανόνα σαν μία ατομική πρόταση. Την πρώτη φορά που ο αλγόριθμος απάντησης σε ερώτηση θα συναντήσει μία εμφάνιση του ! ως η πρώτη πρόταση του τρέχοντος στόχου, το διαγράφει (ακριβώς όπως θα έκανε και με ένα γεγονός

που περιέχεται στο πρόγραμμα) και συνεχίζει με τις υπόλοιπες προτάσεις του στόχου. Αν αργότερα ο αλγόριθμος συναντήσει την ίδια εμφάνιση του !. μετά από οπισθοδρόμηση, τότε προκαλείται άμεση αποτυχία όλων των προηγούμενων στόχων μέχρι και τον στόχο κατά την επεξεργασία του οποίου προέκυψε το !.

Παράδειγμα 39 (συνέχεια): Το κατηγόρημα power3 μπορεί να τροποποιηθεί με χρήση αποκοπής ώστε να αποφεύγεται η άσκοπη οπισθοδρόμηση:

```
power4(N,0,1) :- !.
power4(N,K,P) :- K =\= 0,
                 K mod 2 =:= 0,
                 !,
                 L is K//2,
                 power4(N,L,R),
                 P is R*R.
power4(N,K,P) :- K =\= 0,
                 K mod 2 =:= 1,
                 L is K//2,
                 power4(N,L,R),
                 P is R*R*N.
```

Αν η Prolog υπολογίσει την μηδενική δύναμη ενός αριθμού από τον πρώτο κανόνα, τότε το ! θα την εμποδίσει από το να επιχειρήσει να βρεί άλλες τιμές εξετάζοντας τους άλλους κανόνες. Παρόμοια αν η Prolog υπολογίσει τη δύναμη ενός αριθμού για άρτιο εκθέτη με βάση το δεύτερο κανόνα του ορισμού, τότε το ! θα την αποτρέψει από το να επιχειρήσει να βρεί και άλλες τιμές εφαρμόζοντας τον τρίτο κανόνα.

Παρατηρούμε ότι αν τελικά η Prolog εξετάσει τον δεύτερο κανόνα τότε δεν έχει συναντήσει το ! του πρώτου κανόνα. Αυτό όμως σημαίνει ότι η μεταβλητή K δεν έχει τιμή 0. Συνεπώς μπορούμε να παραλείψουμε τη συνθήκη K =\= 0 από το δεύτερο κανόνα. Επίσης αν η Prolog εξετάσει τον τρίτο κανόνα τότε δεν έχει συναντήσει το ! σε κανέναν από τους δύο προηγούμενους κανόνες. Αυτό όμως σημαίνει ότι η μεταβλητή K δεν έχει τιμή 0 (αφού δεν συνάντησε το ! στον πρώτο κανόνα) και επίσης ότι η συνθήκη K mod 2 =:= 0 δεν είναι αληθής (αφού δεν συνάντησε το ! στον δεύτερο κανόνα), που συνεπάγεται ότι ισχύει K mod 2 =:= 1. Αρα μπορούμε να παραλείψουμε τις δύο πρώτες συνθήκες από τον τρίτο κανόνα. και να έχουμε τον παρακάτω απλουστευμένο ορισμό:

```
power5(N,0,1) :- !.
power5(N,K,P) :- K mod 2 =:= 0,
                 !,
                 L is K//2,
                 power5(N,L,R),
                 P is R*R.
power5(N,K,P) :- L is K//2,
                 power5(N,L,R),
                 P is R*R*N.
```

Υπάρχει μία σημαντική διαφορά στη χρήση του ! στους ορισμούς των power4 και power5. Στον ορισμό του power4 το ! δεν αλλάζει την σημασία του προγράμματος, απλά αποτρέπει την άσκοπη οπισθοδρόμηση. Αν παραλειφθεί το ! προκύπτει ισοδύναμος ορισμός (αυτός

του power3, από τον οποίο ξεκινήσαμε). Αντίθετα αν διαγράψουμε το ! από τον δεύτερο κανόνα του power5. τότε προκύπτει ένα πρόγραμμα που δεν είναι ισοδύναμο με το power5.

```
power6(N,0,1) :- !.
power6(N,K,P) :- K mod 2 =:= 0,
                 L is K//2,
                 power6(N,L,R),
                 P is R*R.
power6(N,K,P) :- L is K//2,
                 power6(N,L,R),
                 P is R*R*N.
```

Οι διαφορά των προγραμάτων φαίνεται από τις παρακάτω ερωτήσεις:

```
?- power5(2,10,P).
P = 1024 ;
no

?- power6(2,10,P).
P = 1024 ;
P = 16384 ;
P = 2048 ;
P = 32768 ;
no
```

Αυτό συμβαίνει επειδή η συνθήκη K mod 2 =:= 1 είναι απαραίτητη ώστε να είναι σωστός ο ορισμός, και στο power5 η συνθήκη αυτή εξασφαλίζεται λόγω της χρήσης του !. ∎

Το παρακάτω τεχνητό παράδειγμα παρουσιάζει τα βασικότερα σενάρια που προκύπτουν από τη χρήση του !.

Παράδειγμα 40: Έστω το παρακάτω πρόγραμμα:

```
g(N,K) :- s(N),!,t(N,K),z(K). %  1
g(N,K) :- f(K,N).             %  2
g(N,0) :- z(N).               %  3

p(a,1).                       %  4
p(b,2).                       %  5
p(c,3).                       %  6
p(d,4).                       %  7
p(e,2).                       %  8
p(e,1).                       %  9

s(2).                         % 10
s(3).                         % 11
s(4).                         % 12

t(3,0).                       % 13
t(3,2).                       % 14
t(4,1).                       % 15
```

```
f(5,1).                         % 16
f(4,4).                         % 17

z(1).                           % 18
z(2).                           % 19

r(a,0).                         % 20
r(b,0).                         % 21
r(c,2).                         % 22
r(d,4).                         % 23
r(e,5).                         % 24
```

Αν κάνουμε την ερώτηση

```
?- p(a,X),g(X,Y),r(a,Y).
```

η Prolog ικανοποιεί την πρώτη πρόταση θέτοντας `X = 1`. Στη συνέχεια προσπαθεί να ικανοποιήσει την προταση `g(1,Y)` ξεκινώντας από το πρώτο κανόνα. Επειδή η πρόταση `s(1)` δεν αληθεύει, ο πρώτος κανόνας αποτυγχάνει πριν από το `!`. Συνεπώς η Prolog οπισθοδρομεί κανονικά, και προσπαθεί να ικανοποιήσει την προταση `g(1,Y)` με βάση το δεύτερο κανόνα. Απο τον κανόνα αυτόν υπολογίζει την τιμή `Y = 5`, ωστόσο στη συνέχεια αποτυγχάνει, καθώς η πρόταση `r(a,5)` δεν αληθεύει. Η Prolog οπισθοδρομεί ξανά, υπολογίζει την τιμή `Y = 0` από τον τρίτο κανόνα, και επιστρέφει τελικά την απάντηση `X = 1, Y = 0`, καθώς η πρόταση `r(a,0)` υπάρχει στο πρόγραμμα. Αναλυτικά:

```
[0] p(a,X),g(X,Y),r(a,Y)
    {X = X, Y = Y}
    1 --> 4
    p(a,1).
    {X = 1}

[1] g(1,Y),r(a,Y)
    {X = 1, Y = Y}
    1 --> 1
    g(N1,K1) :- s(N1),!,t(N1,K1),z(K1).
    {N1 = 1, K1 = Y}

[2] s(1),!,t(1,Y),z(Y),r(a,Y)
    {X = 1, Y = Y}
    1 --> EOP
    failure - backtracking

[1] g(1,Y),r(a,Y)
    {X = 1, Y = Y}
    2 --> 2
    g(N2,K2) :- f(K2,N2).
    {N2 = 1, K2 = Y}

[2] f(Y,1),r(a,Y)
```

```
    {X = 1, Y = Y}
    1 --> 16
    f(5,1).
    {Y = 5}

[3] r(a,5)
    {X = 1, Y = 5}
    1 --> EOP
    failure - backtracking

[2] f(Y,1),r(a,Y)
    {X = 1, Y = Y}
    17 --> EOP
    failure - backtracking

[1] g(1,Y),r(a,Y)
    {X = 1, Y = Y}
    3 --> 3
    g(N3,0) :- z(N3).
    {N3 = 1, Y = 0}

[2] z(1),r(a,0)
    {X = 1, Y = 0}
    1 --> 18
    z(1).
    {}

[3] r(a,0)
    {X = 1, Y = 0}
    1 --> 20
    r(a,0).
    {}

[4] #
    {X = 1, Y = 0}
------------------------------------------------------------
X = 1
Y = 0
------------------------------------------------------------
```

Αν κάνουμε την ερώτηση

```
?- p(b,X),g(X,Y),r(b,Y).
```

η Prolog ικανοποιεί την πρώτη πρόταση θέτοντας `X = 2`. Στη συνέχεια προσπαθεί να ικανοποιήσει την προταση `g(2,Y)` ξεκινώντας από το πρώτο κανόνα. Η πρόταση `s(2)` υπάρχει στο πρόγραμμα και η Prolog συναντάει το `!`. Στη συνέχεια αποτυγχάνει να ικανοποιήσει την πρόταση `t(2,Y)` και το `!` εμποδίζει την οπισθοδρόμηση για ικανανοποίηση της πρότασης `g(2,Y)`, η οποία αποτυγχάνει άμεσα. Επειδή δεν υπάρχουν άλλες τιμές του `X` για τις οποίες να ικανοποιείται η πρόταση `p(b,X)` η Prolog επιστρέφει `no`. Αναλυτικά:

```
[0] p(b,X),g(X,Y),r(b,Y)
    {X = X, Y = Y}
    1 --> 5
    p(b,2).
    {X = 2}

[1] g(2,Y),r(b,Y)
    {X = 2, Y = Y}
    1 --> 1
    g(N1,K1) :- s(N1),!,t(N1,K1),z(K1).
    {N1 = 2, K1 = Y}

[2] s(2),!,t(2,Y),z(Y),r(b,Y)
    {X = 2, Y = Y}
    1 --> 10
    s(2).
    {}

[3] !,t(2,Y),z(Y),r(b,Y)
    {X = 2, Y = Y}
    ! : succeeds

[4] t(2,Y),z(Y),r(b,Y)
    {X = 2, Y = Y}
    1 --> EOP
    failure - backtracking

[3] !,t(2,Y),z(Y),r(b,Y)
    {X = 2, Y = Y}
    ! : controls backtracking

[0] p(b,X),g(X,Y),r(b,Y)
    {X = X, Y = Y}
    6 --> EOP
    failure
-------------------------------------------------------------
no
-------------------------------------------------------------
```

Αν κάνουμε την ερώτηση

```
?- p(c,X),g(X,Y),r(c,Y).
```

η Prolog ικανοποιεί την πρώτη πρόταση θέτοντας `X = 3`. Στη συνέχεια προσπαθεί να ικανοποιήσει την προταση `g(3,Y)` ξεκινώντας από το πρώτο κανόνα. Η πρόταση `s(3)` υπάρχει στο πρόγραμμα και η Prolog συναντάει το `!`. Η πρόταση `t(3,Y)` αληθεύει για `Y = 0`, ωστόσο δεν αληθεύει η `z(0)`. Η Prolog οπισθοδρομεί στο προηγούμενο στόχο χωρίς να συναντήσει το `!` και ικανοποιεί το `t(3,Y)` θέτοντας `Y = 2`. Οι προτάσεις `z(2)` και `r(c,2)` αληθεύουν, και τελικά επιστρέφεται η απάντηση `X = 3, Y = 2`.

Αναλυτικά:

```
[0] p(c,X),g(X,Y),r(c,Y)
    {X = X, Y = Y}
    1 --> 6
    p(c,3).
    {X = 3}

[1] g(3,Y),r(c,Y)
    {X = 3, Y = Y}
    1 --> 1
    g(N1,K1) :- s(N1),!,t(N1,K1),z(K1).
    {N1 = 3, K1 = Y}

[2] s(3),!,t(3,Y),z(Y),r(c,Y)
    {X = 3, Y = Y}
    1 --> 11
    s(3).
    {}

[3] !,t(3,Y),z(Y),r(c,Y)
    {X = 3, Y = Y}
    ! : succeeds

[4] t(3,Y),z(Y),r(c,Y)
    {X = 3, Y = Y}
    1 --> 13
    t(3,0).
    {Y = 0}

[5] z(0),r(c,0)
    {X = 3, Y = 0}
    1 --> EOP
    failure - backtracking

[4] t(3,Y),z(Y),r(c,Y)
    {X = 3, Y = Y}
    14 --> 14
    t(3,2).
    {Y = 2}

[5] z(2),r(c,2)
    {X = 3, Y = 2}
    1 --> 19
    z(2).
    {}

[6] r(c,2)
    {X = 3, Y = 2}
```

```
    1 --> 22
    r(c,2).
    {}

[7] #
    {X = 3, Y = 2}
------------------------------------------------------------
X = 3
Y = 2
------------------------------------------------------------
```

Έστω τώρα ότι κάνουμε την ερώτηση

```
?- p(d,X),g(X,Y),r(d,Y).
```

η Prolog ικανοποιεί την πρώτη πρόταση θέτοντας `X = 4` και τη δεύτερη προταση θέτωντας `Y = 1`, με εφαρμογή του πρώτου κανόνα. Επειδή η πρόταση `r(4,1)` δεν αληθεύει, γίνεται οπισθοδρόμηση η οποία συναντάει το `!`. Αυτό αποτρέπει την Prolog από το εξετάσει το δεύτερο και τον τρίτο κανόνα στον ορισμό του `g` για να βρει άλλες τιμές της `Y` για τις οποίες αληθεύει η πρόταση `g(4,Y)`. Επειδή δεν υπάρχουν άλλες τιμές του `X` για τις οποίες να αληθεύει η πρόταση `p(d,X)` η Prolog επιστρέφει `no`.

Αναλυτικά:

```
[0] p(d,X),g(X,Y),r(d,Y)
    {X = X, Y = Y}
    1 --> 7
    p(d,4).
    {X = 4}

[1] g(4,Y),r(d,Y)
    {X = 4, Y = Y}
    1 --> 1
    g(N1,K1) :- s(N1),!,t(N1,K1),z(K1).
    {N1 = 4, K1 = Y}

[2] s(4),!,t(4,Y),z(Y),r(d,Y)
    {X = 4, Y = Y}
    1 --> 12
    s(4).
    {}

[3] !,t(4,Y),z(Y),r(d,Y)
    {X = 4, Y = Y}
    ! : succeeds

[4] t(4,Y),z(Y),r(d,Y)
    {X = 4, Y = Y}
    1 --> 15
    t(4,1).
```

```
    {Y = 1}

[5] z(1),r(d,1)
    {X = 4, Y = 1}
    1 --> 18
    z(1).
    {}

[6] r(d,1)
    {X = 4, Y = 1}
    1 --> EOP
    failure - backtracking

[5] z(1),r(d,1)
    {X = 4, Y = 1}
    19 --> EOP
    failure - backtracking

[4] t(4,Y),z(Y),r(d,Y)
    {X = 4, Y = Y}
    16 --> EOP
    failure - backtracking

[3] !,t(4,Y),z(Y),r(d,Y)
    {X = 4, Y = Y}
    ! : controls backtracking

[0] p(d,X),g(X,Y),r(d,Y)
    {X = X, Y = Y}
    8 --> EOP
    failure
-----------------------------------------------------------
no
-----------------------------------------------------------
```

Έστω τέλος η ερώτηση

```
?- p(e,X),g(X,Y),r(e,Y).
```

Παρόμοια με την ερώτηση για το `b` η Prolog ικανοποιεί την πρώτη πρόταση θέτοντας `X = 2`. Στη συνέχεια προσπαθεί να ικανοποιήσει την προταση `g(2,Y)` ξεκινώντας από το πρώτο κανόνα, αποτυγχάνει να ικανοποιήσει το `t(2,Y)` και λόγω του `!` αποτυγχάνει άμεσα να ικανοποιήσει το `g(2,Y)`. Στη συνέχεια επιστρέφει στον αρχικό στόχο και ικανοποιεί το `p(b,X)` θέτοντας `X = 1`. Τελικά επιστρέφει την απάντηση `X = 1, Y = 5`. Βλέπουμε ότι το `!` δεν εμποδίζει την αναζήτηση άλλων τιμών για τη `X` ώστε να αληθεύει η πρόταση `p(e,X)`, αφού ο έλεγχος της οπισθοδρόμησης αφορά μόνο τον ορισμό του `g`. Αναλυτικά:

```
[0] p(e,X),g(X,Y),r(e,Y)
    {X = X, Y = Y}
    1 --> 8
```

```
    p(e,2).
    {X = 2}

[1] g(2,Y),r(e,Y)
    {X = 2, Y = Y}
    1 --> 1
    g(N1,K1) :- s(N1),!,t(N1,K1),z(K1).
    {N1 = 2, K1 = Y}

[2] s(2),!,t(2,Y),z(Y),r(e,Y)
    {X = 2, Y = Y}
    1 --> 10
    s(2).
    {}

[3] !,t(2,Y),z(Y),r(e,Y)
    {X = 2, Y = Y}
    ! : succeeds

[4] t(2,Y),z(Y),r(e,Y)
    {X = 2, Y = Y}
    1 --> EOP
    failure - backtracking

[3] !,t(2,Y),z(Y),r(e,Y)
    {X = 2, Y = Y}
    ! : controls backtracking

[0] p(e,X),g(X,Y),r(e,Y)
    {X = X, Y = Y}
    9 --> 9
    p(e,1).
    {X = 1}

[1] g(1,Y),r(e,Y)
    {X = 1, Y = Y}
    1 --> 1
    g(N1,K1) :- s(N1),!,t(N1,K1),z(K1).
    {N1 = 1, K1 = Y}

[2] s(1),!,t(1,Y),z(Y),r(e,Y)
    {X = 1, Y = Y}
    1 --> EOP
    failure - backtracking

[1] g(1,Y),r(e,Y)
    {X = 1, Y = Y}
    2 --> 2
    g(N2,K2) :- f(K2,N2).
```

```
    {N2 = 1, K2 = Y}

[2] f(Y,1),r(e,Y)
    {X = 1, Y = Y}
    1 --> 16
    f(5,1).
    {Y = 5}

[3] r(e,5)
    {X = 1, Y = 5}
    1 --> 24
    r(c,5).
    {}

[4] #
    {X = 1, Y = 5}
------------------------------------------------------------
X = 1
Y = 5
------------------------------------------------------------
```

Σημειώνεται ότι αν παραλείπαμε το ! τότε η σημασία του προγράμματος θα ήταν διαφορετική (θα παίρναμε απάντηση διαφορετική του **no** σε όλες τις ερωτήσεις). Το ίδιο θα συνέβαινε και αν διατηρήσουμε το ! και μεταφέραμε τον κανόνα που το περιέχει στο τέλος του ορισμού του κατηγορήματος **g**. Συνεπώς το ! εκτός του ότι ελέγχει την οπισθοδρόμηση, καταστρέφει τον δηλωτικό χαρακτήρα της Prolog. ∎

Παράδειγμα 41: Εύρεση πρώτης εμφάνισης στοιχείου σε λίστα.

Το παρακάτω κατηγόρημα `find(X,L,N)` αληθεύει αν το στοιχείο `X` εμφανίζεται στη θέση `N` της λίστας `L`.

```
find(X,[X|T],1).
find(X,[H|T],N):- find(X,T,K),
                  N is K+1.
```

Ένα στοιχείο μπορεί να εμφανίζεται σε πολλές θέσεις μίας λίστας:

```
find(1,[5,1,2,3,1,1,2],N).
N = 2 ;
N = 5 ;
N = 6 ;
no
```

Μπορούμε να ορίσουμε ένα παρόμοιο καντηγόρημα `find1(X,L,N)` το οποίο θα αληθεύει αν το στοιχείο `X` εμφανίζεται για πρώτη φορά στη θέση `N` της λίστας `L`. Ο ορισμός του `find1` μπορεί να προκύψει με μία απλή τροποποίηση του ορισμού του `find`, εισάγοντας το ! στην πρώτη πρόταση ώστε να αποφεύγεται η οπισθοδρόμηση που οδηγεί σε εύρεση πολλαπλών απαντήσεων.

```
find1(X,[X|T],1) :- !.
find1(X,[H|T],N):- find1(X,T,K),
                   N is K+1.
```

Οι ερωτήσεις που σχηματίζονται με το `find1`, επιστρέφουν το πολύ μία απάντηση:

```
find1(1,[5,1,2,3,1,1,2],N).
N = 2 ;
no

find1(X,[5,1,2,3,1,1,2],N).
X = 5
N = 1 ;
no
```

Παράδειγμα 42: Διαγραφή όλων των εμφανίσεων στοιχείου από λίστα.

Θα ορίσουμε το κατηγόρημα `deleteAll(X,S,L)` το οποίο αληθεύει αν η λίστα `L` προκύπτει με διαγραφή όλων των εμφανίσεων του `X` από την `S`. Ο ορισμός στηρίζεται στις παρακάτω παρατηρήσεις: (α) αν το στοιχείο `X` που διαγράφεται είναι η κεφαλή της `S`, τότε η `S` είναι της μορφής `[X|T]` και η λίστα `L` προκύπτει με διαγραφή όλων των εμφανίσεων της `X` από την `T`. (β) αν το στοιχείο `X` δεν είναι η κεφαλή της `S`, τότε η `S` είναι της μορφής `[H|T]`, με `H` διάφορο του `X`, και η `L` είναι της μορφής `[H|R]` (έχει δηλαδή την ίδια κεφαλή με την `S`), όπου `R` είναι η λίστα που προκύπτει με διαγραφή όλων των εμφανίσεων της `X` από την `T`.

Στον παρακάτω ορισμό η συνθήκη που απαιτείται από το (β) εξασφαλίζεται με χρήση του `!`: αν το `X` είναι η κεφαλή της λίστας, τότε ο πρώτος αναδρομικός κανόνας θα φτάσει στο `!` που θα αποτρέψει την εξέταση του τελευταίου κανόνα. Συνεπώς αν εξεταστεί ο τελευταίος κανόνας, το `X` δεν θα είναι η κεφαλή της λίστας και θα πρέπει να συμπεριληφθεί στη λίστα θα προκύψει από τη διαγραφή.

```
deleteAll(X,[],[]).
deleteAll(X,[X|T],L) :- !,
                        deleteAll(X,T,L).
deleteAll(X,[H|T],[H|R]) :- deleteAll(X,T,R).
```

∎

Στα παραπάνω παραδείγματα φαίνεται η χρήση της αποκοπής ώστε να αποφεύγεται η άσκοπη οπισθοδρόμηση, να αποφεύγεται η εύρεση εναλλακτικών απαντήσεων και να αποτρέπεται η εξέταση κανόνων που έπονται ενός κανόνα που οδηγεί σε επιτυχία, χωρίς να απαιτείται οι κανόνες αυτοί να περιέχουν τις αρνήσεις συνθηκών που ήδη έχουν ελεγχθεί. Στη συνέχεια θα δούμε πώς μπορούμε να χρησιμοποιήσουμε την αποκοπή για να ενσωματώσουμε στην Prolog μια μορφή άρνησης

1.19 Αρνηση ως Αποτυχία

Στην Prolog μπορούμε να δηλώσουμε άμεσα ότι μία πρόταση είναι αληθής, ωστόσο δεν μπορούμε να δηλώσουμε ότι μία πρόταση είναι ψευδής. Μια πρόταση είναι ψευδής για την Prolog αν δεν μπορεί να αποδείξει ότι προκύπτει από το πρόγραμμα. Χρησιμοποιώντας την αποκοπή μπορούμε να ορίσουμε μία μορφή άρνησης, η οποία ονομάζεται άρνηση ως αποτυχία.

Παράδειγμα 43: Θα ορίσουμε το κατηγόρημα `not_parent(X,Y)` το οποίο αληθεύει αν ο `X` δεν είναι γονέας του `Y`. Στον ορισμό χρησιμοποιούμε το κατηγόρημα `parent`, όπως επίσης και το ενσωματωμένο κατηγόρημα `fail`, το οποίο αποτυγχάνει πάντα.

```
not_parent(X,Y) :- parent(X,Y),!,fail.
not_parent(X,Y).
```

Αν ο `X` είναι γονέας του `Y`, τότε η πρόταση `parent(X,Y)` στο σώμα του κανόνα αληθεύει, η Prolog συναντάει το ! και στη συνέχεια αποτυγχάνει λόγω του `fail`. Το ! αποτρέπει την οπισθοδρόμηση, συνεπώς το `not_parent(X,Y)` δεν προκύπτει από το πρόγραμμα. Αν αντίθετα ο `X` δεν είναι γονέας του `Y`, τότε η πρόταση `parent(X,Y)` δεν αληθεύει, και άρα το `not_parent(X,Y)` αληθεύει με βάση το δεύτερο κανόνα. ∎

Παράδειγμα 44: Μπορούμε να ορίσουμε ένα κατηγόρημα `'Not'` (υψηλότερης τάξης) το οποίο θα πέρνει ως όρισμα μία πρόταση και θα αληθεύει αν η πρόταση δεν προκύπτει λογικά από το πρόγραμμα. Ο ορισμός του είναι παρόμοιος με αυτόν του `not_member(X,L)` στο προηγούμενο παράδειγμα.

```
'Not'(P) :- P,!,fail.
'Not'(P).
```

Μπορούμε να χρησιμοποιήσουμε το `'Not'` για να σχηματίσουμε την άρνηση κατηγορημάτων που έχουμε ορίσει:

```
?- 'Not'(member(2,[1,2,3])).
no
?- 'Not'(member(4,[1,2,3])).
yes

?- 'Not'(member(X,[1,2,3])).
no
?- 'Not'(member(X,[])).
X = _0
```

Παρατηρούμε ότι όταν η ερώτηση περιέχει μεταβλητές, τότε αν υπάρχει έστω και ένας συνδυασμός τιμών για τις μεταβλητές αυτές για τις οποίες το όρισμα του `'Not'` να αληθεύει, η Prolog επιστρέφει απάντηση `no`, παρότι για κάποιες άλλες τιμές των μεταβλητών όρισμα του `'Not'` μπορεί να μη αληθεύει. Αν το όρισμα του `'Not'` δεν αληθεύει για καμία τιμή των μεταβλητών τότε η Prolog επιτυγχάνει, χωρίς όμως να δεσμεύει τις μεταβλητές που περιέχονται στην ερώτηση. ∎

Η Prolog διαθέτει τον τελεστή `not`, ο οποίος ορίζεται όπως το κατηγόρημα `'Not'` (απλά δεν απαιτεί το όρισμά του να κλείνεται σε παρανθέσεις). Οταν κάνουμε στη Prolog μία ερώτηση της μορφής

```
?- not P.
```

(όπου `P` ατομική πρόταση ή ακολουθία ατομικών προτάσεων μέσα σε παρένθεση) η Prolog προσπαθεί να αποδείξει την `P`. Αν πετύχει απαντάει no, αλλιώς επιστρέφει θετική απάντηση (yes, ή τις μεταβλητές της ερώτησης χωρίς δέσμευση).

Η άρνηση που ορίζεται με τον τελεστή `not` διαφέρει από τη λογική άρνηση. Η άρνηση στην Prolog είναι άρνηση ως αποτυχία: η πρόταση `not P` αληθεύει στην περίπτωση που η `P` δεν αποδεικνύεται από το πρόγραμμα. Αυτό όμως δεν σημαίνει ότι η `not P` αποτελεί λογική συνέπεια του προγράμματος.

Παράδειγμα 45: Το παρακάτω πρόγραμμα αποτελείται από προτάσεις που αφορούν τις καθημερινές πτήσεις από Αθήνα προς Ιωάννινα και αντίστροφα:

```
flight('ATHENS','IOANNINA',time(13,15)).
flight('ATHENS','IOANNINA',time(15,50)).
flight('IOANNINA','ATHENS',time(14,45)).
flight('IOANNINA','ATHENS',time(17,20)).
```

Αν κάνουμε την ερώτηση

```
?- not flight('ATHENS', 'IOANNINA', time(8,00)).
```

η Prolog θα απαντήσει `yes`. Ωστόσο η πρόταση στην ερώτηση δεν προκύπτει λογικά από το πρόγραμμα. Η απάντηση αυτή μπορεί να θεωρηθεί εύλογη αν κάνουμε την υπόθεση ότι όλες οι πτήσεις μεταξύ των δύο πόλεων είναι καταγεγραμένες στο πρόγραμμα. ∎

Όπως φαίνεται και από το παραπάνω παράδειγμα η Prolog θεωρεί ψευδείς όσες προτάσεις δεν προκύπτουν από το πρόγραμμα και συνεπώς θεωρεί αληθείς τις αρνήσεις τους. Αυτό ονομάζεται 'υπόθεση του κλειστού κόσμου' η οποία μπορεί να εφαρμοστεί σε πολλές περιπτώσεις, όπως στο παραπάνω παράδειγμα.

Φυσικά η υπόθεση του κλειστού κόσμου δεν είναι καθολικά εφαρμόσιμη: αν για παράδειγμα στο βιβλίο των Διακριτών Μαθηματικών δεν αναφέρεται το Πυθαγόριο θεώρημα, αυτό δεν σημαίνει ότι δεν ισχύει και ότι μπορούμε στο πλαίσιο του μαθήματος αυτού να θεωρούμε ότι η άρνησή του είναι αληθής.

Στο παρακάτω παράδειγμα φαίνεται η διαφορά της άρνησης στη μαθηματική λογική και της άρνησης ως αποτυχία.

Παράδειγμα 46: Στη μαθηματική λογική οι τρείς παρακάτω προτάσεις είναι ισοδύναμες

- $male(X) \vee female(X)$
- $\neg male(X) \rightarrow female(X)$
- $\neg female(X) \rightarrow male(X)$

(το $\vee$ συμβολίζει το λογικό 'ή' ενώ το $\neg$ τη λογική άρνηση).

Η δεύτερη πρόταση αντιστοιχεί στον παρακάτω κανόνα Prolog:

```
female(X) :- not male(X).
```

Αν φορτώσουμε το πρόγραμμα που αποτελείται μόνο από τον παραπάνω κανόνα τότε παίρνουμε τις παρακάτω απαντήσεις:

```
?- male('Laurence').
no

?- female('Laurence').
yes
```

Η τρίτη πρόταση αντιστοιχεί στον παρακάτω κανόνα Prolog:

```
male(X) :- not female(X).
```

Αν φορτώσουμε το πρόγραμμα που αποτελείται μόνο από τον παραπάνω κανόνα τότε παίρνουμε τις παρακάτω απαντήσεις:

```
?- male('Laurence').
yes

?- female('Laurence').
no
```

Παρατηρούμε ότι η χρήση της άρνησης ως αποτυχία στη θέση της κλασικής άρνησης έχει ως αποτέλεσμα οι κανόνες που αντιστοιχούν σε δύο ισοδύναμες προτάσεις της λογικής να έχουν διαφορετική σημασία για την Prolog. ∎

1.20 Ενσωματωμένα Κατηγορήματα

Η Prolog διαθέτει ένα πλήθος ενσωματωμένων κατηγορημάτων που επιτρέπουν να ελέγχουμε τον 'τύπο' των όρων:

- `number`: αληθεύει αν το όρισμά του είναι αριθμός ή μεταβλητή δεσμευμένη σε αριθμό
- `integer`: αληθεύει αν το όρισμά του είναι ακέραιος ή μεταβλητή δεσμευμένη σε ακέραιο
- `float`: αληθεύει αν το όρισμά του είναι πραγματικός ή μεταβλητή δεσμευμένη σε πραγματικό
- `atom`: αληθεύει αν το όρισμά του είναι άτομο ή μεταβλητή δεσμευμένη σε άτομο
- `atomic`: αληθεύει αν το όρισμά του είναι άτομο ή αριθμός, ή μεταβλητή δεσμευμένη σε άτομο ή αριθμό
- `compound`: αληθεύει αν το όρισμά του είναι σύνθετος όρος ή μεταβλητή δεσμευμένη σε σύνθετο όρο
- `var`: αληθεύει αν το όρισμά του μεταβλητή χωρίς δέσμευση.
- `nonvar`: αληθεύει αν το όρισμά δεν είναι μεταβλητή ή είναι δεσμευμένη μεταβλητή

Παραδείγματα χρήσης των παραπάνω κατηγορημάτων:

```
?- integer(1).
yes

?- integer(X).
no

?- X = 1, integer(X).
X = 1

?- atom(a).
yes

?- atom(0).
no

?- atomic(a).
yes

?- atomic(0).
yes

?- atomic(f(X)).
no

?- var(X).
X = _0

?- X = f(X), var(X).
no
```

Χρησιμοποιώντας τα παραπάνω κατηγορήματα μπορούμε ελέγχουμε τον τύπο των ορισμάτων στους κανόνες που ορίζουν ένα κατηγόρημα, έτσι ώστε να απομονώνουμε τις προβληματικές περιπτώσεις.

Παράδειγμα 47: Αριθμητικές συναρτήσεις.

Χρησιμοποιώντας το ενσωματωμένο κατηγόρημα `integer` και το ! μπορούμε να ορίσουμε την απόλυτα ορθή υλοποίηση του κατηγορήματος για υπολογισμό του παραγοντικού. Το `integer` χρησιμοποιείται για να εξασφαλίσει ότι το όρισμα είναι ακέραιος αριθμός. Στη συνέχεια ο τελεστής `>=` χρησιμοποιείται για να εξασφαλίσει το ότι ο αριθμός είναι μη αρνητικός. Το ! χρησιμοποιείται για να αποφεύγεται η χρήση του δεύτερου κανόνα όταν ζητηθούν εναλλακτικές τιμές για το 0!, χωρίς να απαιτείται η σύγκριση `N > 0` στον αναδρομικό κανόνα:

```
factGen(N,F) :- integer(N),
                N >= 0,
                fact4(N,F).
```

```
fact4(0,1) :- !.
fact4(N,F) :- K is N-1,
              fact4(K,G),
              F is N*G.
```

Αν το πρώτο όρισμα είναι μη αρνητικός ακέραιος (ή μεταβλητή που έχει δεσμευτεί σε μη αρνητικό ακέραιο) επιστρέφεται μία μόνο απάντηση που είναι η σωστή τιμή του παραγοντικού:

```
?- factGen(4,X).
X = 24 ;
no

?- factGen(-3,X).
no

?- factGen(N,120).
no

?- N = 3, factGen(N,X).
N = 3
X = 6

?- factGen(a,X).
no
```

Αντίστοιχα μπορούμε να εισάγουμε έλεγχο τύπου και οπισθοδρόμησης στους ορισμούς των κατηγορημάτων για υπολογισμό δύναμης, μέγιστου κοινού διαιρέτη κλπ.

```
powerGen(N,K,P) :- number(N),
                   integer(K),
                   K >= 0,
                   power5(N,K,P).

power5(N,0,1) :- !.
power5(N,K,P) :- K mod 2 =:= 0,
                 !,
                 L is K//2,
                 power5(N,L,R),
                 P is R*R.
power5(N,K,P) :- L is K//2,
                 power5(N,L,R),
                 P is R*R*N.

gcdEucGen(M,N,D):- integer(M),
                   integer(N),
                   M > 0,
                   N > 0,
                   gcdEuc3(M,N,D).
gcdEuc3(N,N,N) :- !.
```

```
gcdEuc3(M,N,D) :- M < N,
                  !,
                  K is N-M,
                  gcdEuc3(M,K,D).
gcdEuc3(M,N,D) :- K is M-N,
                  gcdEuc3(N,K,D).
```

Μέχρι τώρα για να πάρουμε όλες τις διαφορετικές απαντήσεις σε μία ερώτηση, γράφαμε ; μετά από κάθε απάντηση. Ενδέχεται όμως να θέλουμε να συγκεντρώσουμε όλες τις απαντήσεις ώστε να μπορούμε να τις επεξεργαστούμε συνολικά. Αυτό μπορεί να γίνει με τα ενσωματωμένα κατηγορήματα `bagof` και `setof`.

Αν `X` είναι μεταβλητή και `Q` είναι μία ερώτηση στην οποία η μοναδική μεταβλητή είναι η `X`, τότε το `bagof(X,Q,L)` αληθεύει αν `L` είναι η λίστα με όλες τις τιμές της `X` που θα πέρναμε ως απαντήσεις αν κάναμε την ερώτηση

```
?- Q.
```

Για παράδειγμα για να βρούμε τη λίστα με όλα τα παιδιά της Διώνης κάνουμε την ερώτηση

```
?- bagof(X,parent(dioni,X),L).
X = _0
L = [pelopas,niovi]
```

ενώ για να βρούμε τη λίστα με όλες τις μεταθέσεις της `[1,2,3]`, κάνουμε την ερώτηση

```
?- bagof(P,perm1([1,2,3],P),L).
P = _0
L = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

Αν η ερώτηση είναι σύνθετη, τότε την κλείνουμε σε παρενθέσεις:

```
?- bagof(X,(predecessor(tantalos,X),male(X)),L).
X = _0
L = [pelopas,atreas]
```

Στη γενική περίπτωση, το πρώτο όρισμα μπορεί να είναι όρος `T` οποιασδήποτε μορφής που ενδέχεται να περιέχει οποιοδήποτε πλήθος μεταβλητών. Σε αυτή την περίπτωση αυτή η λίστα `L` αποτελείται από όλους τους όρους που προκύπτουν από τον `T` με αντικατάσταση των μεταβλητών σύμφωνα με τις δεσμεύσεις σε κάθε απάντηση για την `Q`. Αν η `Q` δεν μπορεί να ικανοποιηθεί τότε επιστρέφεται `no` και όχι η κενή λίστα.

```
?- bagof(A+B, sum(A,B,s(s(0))),L).
A = _0
B = _1
L = [s(s(0))+0,s(0)+s(0),0+s(s(0))]

?- bagof(A, sum(A,s(s(s(0))),0),L).
no
```

Αν η ερώτηση `Q` περιέχει μεταβλητές που δεν εμφανίζονται στον όρο `T`, τότε η Prolog μας επιστρέφει μία ξεχωριστή λίστα για κάθε συνδυασμό των πρόσθετων μεταβλητών για τις οποίες υπάρχει απάντηση στην `Q`

```
?- bagof(X,parent(P,X),L).
X = _0
P = dioni
L = [pelopas,niovi] ;

X = _0
P = pelopas
L = [atreas] ;

X = _0
P = tantalos
L = [pelopas,niovi] ;
```

Μπορούμε να πάρουμε μία συνολική λίστα χρησιμοποιώντας τον τελεστή ^ ο οποίος αποτρέπει τη δέσμευση της μεταβλητής που βρίσκεται αριστερά του:

```
?- bagof(X,P^parent(P,X),L).
X = _0
P = _1
L = [pelopas,pelopas,niovi,niovi,atreas]
```

Μπορούμε να χρησιμοποιήσουμε το bagof για τον ορισμό άλλων κατηγορημάτων:

```
numOfChildren(X,N) :- bagof(Y,parent(X,Y),L),
                      length(L,N).
numOfChildren(X,0) :- male(X),
                      not parent(X,_).
numOfChildren(X,0) :- female(X),
                      not parent(X,_).
```

Κάνουμε τις παρακάτω ερωτήσεις:

```
?- numOfChildren(dioni,N).
N = 2

?- numOfChildren(niovi,N).
N = 0

?- bagof(X-N,numOfChildren(X,N),L).
X = _0
N = _1
L = [dioni-2,pelopas-1,tantalos-2,atreas-0,niovi-0]
```

Το setof(T,Q,L) λειτουργεί παρόμοια με το bagof με τη διαφορά ότι διαγράφει πολλαπλές εμφανίσεις του ίδιου στοιχείου από την L και εμφανίζει τα στοιχεία της ταξινομημένα:

```
?- bagof(S,sublist(S,[3,2,1]),L).
S = _0
L = [[],[3],[3,2],[3,2,1],[],[2],[2,1],[],[1],[]]

?- setof(S,sublist(S,[3,2,1]),L).
S = _0
L = [[],[1],[2],[2,1],[3],[3,2],[3,2,1]]
```

Ο τελεστής =.. χρησιμοποιείται για να συνθέσει ένα όρο ή αντίστροφα να αποσυνθέσει έναν όρο. Το T =.. L αληθευει αν T είναι σύνθετος όρος και L η λίστα με κεφαλή το συναρτησιακό σύμβολο που σχηματίζει τον T και ουρά τα ορίσματα του T. Επίσης το T =.. L αληθευει αν T είναι άτομο ή αριθμός και L η λίστα με μοναδικό στοιχείο το άτομο ή τον αριθμό αυτό.

```
?- f(1,2,g(0)) =.. L.
L = [f,1,2,g(0)]

?- T =.. [a,0,1].
T = a(0,1)

?- T =.. [f(0),1,1].
no

?- T =.. [0,a,b].
no

?- [1,2] =.. L.
L = [.,1,[2]]
```

Το κατηγόρημα call(G) προκαλεί την εκτέλεση του στόχου G, ο οποίος μπορεί να έχει δημιουργηθεί δυναμικά. Μπορούμε να χρησιμοποιήσουμε τον τελεστή =.. σε συνδυασμό με το G ώστε να δημιουργήσουμε κατηγορήματα υψηλότερης τάξης.

Παράδειγμα 48: υπολογισμός αθροίσματος.

Θα ορίσουμε ένα κατηγόρημα με το οποίο θα μπορούμε να υπολογίσουμε ένα άθροισμα της μορφής $\sum_{i=1}^{n} f(i)$. Το κατηγόρημα sumF(F,N,S) αληθεύει αν η τιμή της S είναι το άθροισμα $\sum_{i=1}^{\mathtt{N}} f(i)$, όπου f η συνάρτηση που ορίζεται από το κατηγόρημα F.

```
sumF(F,0,0).
sumF(F,N,S) :- K is N-1,
               G =.. [F,N,X],
               call(G),
               sumF(F,K,R),
               S is R+X.
```

Μπορούμε να χρησιμοποιήσουμε το sumF με πρώτο όρισμα το όνομα κάποιου κατηγορήματος που υλοποιεί μία αριθμητική συνάρτηση μίας μεταβλητής:

```
?- sumF(fact,4,S).
S = 33

?- sumF(sqrtInt,10,S).
S = 19
```

∎

Παράδειγμα 49: Απεικόνιση στοιχείων λίστας.

Το παρακάτω κατηγόρημα `map(P,L,S)` αληθεύει αν οι λίστες `L` και `S` έχουν το ίδιο μήκος και τα στοιχεία τους που βρίσκονται σε αντίστοιχες θέσεις συνδέονται μέσω του `P`. Αν το `P` υλοποιεί μία συνάρτηση f τότε τα στοιχεία της `S` προκύπτουν από τα στοιχεία της `L` με απεικόνιση μέσω της f.

```
map(P,[],[]).
map(P,[H|T],[X|S]) :- G =.. [P,H,X],
                      call(G),
                      map(P,T,S).
```

Μπορούμε να χρησιμοποιήσουμε το `map` με πρώτο όρισμα το όνομα κάποιου κατηγορήματος (με δύο ορίσματα) που έχουμε ήδη ορίσει:

```
?- map(fact,[1,2,3,4,5],L).
L = [1,2,6,24,120]

?- map(reverse,[[1,2,3],[a,b,c,d],[f(0),g(1)]],L).
L = [[3,2,1],[d,c,b,a],[g(1),f(0)]]

?- map(perm1,[[1,2],[a,b]],L).
L = [[1,2],[a,b]] ;
L = [[1,2],[b,a]] ;
L = [[2,1],[a,b]] ;
L = [[2,1],[b,a]] ;
no
```

1.21 Χειρισμός της βάσης δεδομένων

Όπως έχουμε ήδη αναφέρει η Prolog για να απαντήσει σε μία ερώτηση χρησιμοποιεί τη βάση δεδομένων, η οποία περιέχει ένα σύνολο προτάσεων. Μπορούμε να φορτώσουμε ένα αρχείο προγράμματος χρησιμοποιώντας το ενσωματωμένο κατηγόρημα `consult`. Οι προτάσεις που περιέχονται στο αρχείο που δίνεται ως όρισμα στο `consult` προσθέτονται στο τέλος της βάσης δεδομένων. Οι προτάσεις που ήταν τυχόν φορτωμένες πρίν τη εκτέλεση του `consult` εξακολουθούν να παραμένουν στην βάση δεδομένων.

Εναλλακτικά μπορούμε να φορτώσουμε ένα αρχείο προγράμματος χρησιμοποιώντας το ενσωματωμένο κατηγόρημα `reconsult`. Το `reconsult` φορτώνει τις προτάσεις από το αρχείο που δίνεται ως όρισμα στη βάση δεδομένων, αφού πρώτα διαγράψει προτάσεις που ορίζουν κάποια κατηγορήματα τα οποία ορίζονται και στο υπό φόρτωση αρχείο. Προτάσεις που ήταν φορτωμένες πρίν τη εκτέλεση του `reconsult` και ορίζουν άλλα κατηγορήματα εξακολουθούν να παραμένουν στην βάση δεδομένων.

Τα `consult` και `reconsult` επιτρέπεται να περιέχονται και σε προτάσεις του προγράμματος. Με αυτόν τον τρόπο μπορεί να τροποποιείται δυναμικά η βάση δεδομένων κατά την εκτέλεση του προγράμματος (δηλαδή κατά την εκτέλεση της διαδικασίας απάντησης σε μία ερώτηση). Η Prolog παρέχει ορισμένα πρόσθετα κατηγορήματα με τα οποία μπορεί

να αλλάζει η βάση δεδομένων δυναμικά. Το κατηγόρημα `assert(C)` εισάγει την πρόταση `C` στη βάση δεδομένων (αν η πρόταση `C` είναι κανόνας θα πρέπει να δοθεί μέσα σε παρενθέσεις). Το κατηγόρημα `retract(C)` διαγράφει την πρόταση `C` από βάση δεδομένων. Αν η `C` περιέχει μεταβλητές, τότε διαγράφεται η πρώτη πρόταση από τη βάση δεδομένων που ταυτοποιείται με τη `C`.

Σε ορισμένες περιπτώσεις παίζει ρόλο η θέση μέσα στη βάση δεδομένων όπου θα εισαχθεί μία πρόταση. Η Prolog διαθέτει τα κατηγορήματα `asserta(C)` και `assertz(C)` που εισάγουν την πρόταση `C` αντίστοιχα στην αρχή ή στο τέλος της βάσης δεδομένων.

Παράδειγμα 50: Φωτεινοί σηματοδότες.

Το κατηγόρημα `lights` χρησιμοποιείται για να δηλώσουμε την τρέχουσα κατάσταση ενός φωτεινού σηματοδότη. Η προτάσεις `lights(green)` και `lights(red)` δηλώνουν ότι ο φωτεινός σηματοδότης δείχνει πράσινο ή αντίστοιχα κόκκινο και θέλουμε κάθε χρονική στιγμή μία μόνο από τις δύο να αληθεύει. Στο παρακάτω πρόγραμμα ορίζεται το κατηγόρημα `initLights`, με το οποίο γίνεται η αρχικοποίηση του σηματοδοτη στο παράσινο. Επίσης ορίζεται το κατηγόρημα (χωρίς ορίσματα) `changeLights`, το οποίο διαγράφει από τη βάση δεδομένων την πρόταση που δηλώνει την τρέχουσα κατάσταση του σηματοδότη και εισάγει την πρόταση που δηλώνει την αντίθετη κατάσταση.

```
initLights :- asserta(lights(green)).
changeLights :- lights(green),
                retract(lights(green)),
                asserta(lights(red)).
changeLights :- lights(red),
                retract(lights(red)),
                asserta(lights(green)).
```

Η χρήση του `changeLights` φαίνεται στον παρακάτω διάλογο με το διρμηνέα της Prolog:

```
?- initLights.
yes

?- lights(green).
yes

?- lights(red).
no

?- changeLights.
yes

?- lights(green).
no

?- lights(red).
yes
```

```
?-  changeLights.
yes

?- lights(X).
X = green ;
no
```

∎

Παράδειγμα 51: Στοίβες και Ουρές.

Χρησιμοποιούμε το κατηγόρημα `stack(Stack,Element)` για να δηλώσουμε ότι το στοιχείο `Element` περιέχεται στη στοίβα `Stack`. Για να εισάγουμε ένα στοιχείο στη στοίβα (κατηγόρημα **push**) χρησιμοποιούμε το **asserta** έτσι ώστε το στοιχείο που εισάγεται τελευταίο σε κάποια στοίβα να περιέχεται σε μία πρόταση που βρίσκεται πιο κοντά στην αρχή της βάσης δεδομένων σε σύγκριση με τα στοιχεία της ίδιας στοίβας που έχουν εισαχθεί πιο πριν. Συνεπώς η εξαγωγή του στοιχείου (κατηγόρημα **pop**) μπορεί να γίνει με χρήση του `retract`.

```
push(Stack,Element) :-
     asserta(stack(Stack,Element)).
pop(Stack,Element) :-
     retract(stack(Stack,Element)).
```

Παράδειγμα χρήσης των **push** και **pop**

```
?- push(mystack,a).
yes

?- push(mystack,b).
yes

?-  pop(mystack,X).
X = b

?- push(mystack,c).
yes

?- pop(mystack,X).
X = c

?-  pop(mystack,X).
X = a

?- pop(mystack,X).
no

?- push(stack(1),a).
yes
```

```
?- push(stack(2),b).
yes

?- push(stack(2),c).
yes

?-  push(stack(1),d).
yes

?- pop(stack(2),X).
X = c
```

Τελείως ανάλογα μπορεί να οριστούν οι βασικές λειτουργίες σε ουρά. Η διαφορά σε σχέση με τη στοίβα είναι ότι η εισαγωγή γίνεται με χρήση του `assertz`.

```
enqueue(Queue,Element) :-
        assertz(queue(Queue,Element)).
dequeue(Queue,Element) :-
        retract(queue(Queue,Element)).
```

Παράδειγμα χρήσης των `enqueue` και `dequeue`:

```
?- enqueue(myqueue,a).
yes

?- enqueue(myqueue,b).
yes

?- dequeue(myqueue,X).
X = a

?- dequeue(myqueue,X).
X = b

?- dequeue(myqueue,X).
no
```

∎

Παράδειγμα 52: Προστακτικός προγραμματισμός.

Στην Prolog η έννοια της μεταβλητής είναι διαφορετική από την αντίστοιχη έννοια στις προστακτικές γλώσσες προγραμματισμού. Στο παράδειγμα αυτό θα δούμε πως μπορούμε να μιμηθούμε τη λειτουργία των προστακτικών προγραμμάτων, χρησιμοποιώντας άτομα ως μεταβλητές. Θα χρησιμοποιούμε το κατηγόρημα `value` για να καθορίζουμε τις τρέχουσες αριθμητικές τιμές που συνδέονται με κάθε άτομο-μεταβλητή. Για παράδειγμα το `value(x,0)` θα δηλώνει ότι το άτομο-μεταβλητή `x` έχει την τιμή 0. Θα φροντίσουμε ώστε σε κάθε χρονική στιγμή, για κάθε άτομο-ματαβλητή `x` να υπάρχει στη βάση δεδομένων μία το πολύ πρόταση της παραπάνω μορφής που θα καθορίζει την τιμή του `x`.

Το κατηγόρημα **run** μας επιτρέπει να εκτελέσουμε μία σειρά από εντολές ανάθεσης, οι οποίες αντικαθιστούν την τιμή ενός ατόμου-μεταβλητής μέ την τιμή που προκύπτει από την αποτίμηση μία αριθμητικής παράστασης. Στον ορίσμό του βοηθητικού κατηγορήματος **eval** εκμεταλλευόμαστε την εσωτερική αναπαράσταση των παραστάσεων από όρους που έχουν τη συνήθη σύνταξη, έτσι ώστε να μη χρειάζεται να ασχοληθούμε με προτεραιότητες τελεστών. Το κατηγόρημα **allValues** χρησιμοποιείται για να δούμε τις τιμές όσων ατόμων-μεταβλητών έχουν πάρει τιμή μέχρι την τρέχουσα χρονική στιγμή.

```
run([]).
run([=(X,E)|T]) :- atom(X),
                   eval(E,V),
                   update(X,V),
                   run(T).
update(X,V):- retract(value(X,N)),
              !,
              asserta(value(X,V)).
update(X,V):- asserta(value(X,V)).

allValues(L) :- bagof(X = V,value(X,V),L).
eval(N,N) :- number(N).
eval(A,V) :- atom(A),
             value(A,V).
eval(+(E1,E2),V) :- eval(E1,V1),
                    eval(E2,V2),
                    V is V1+V2.
eval(-(E1,E2),V) :- eval(E1,V1),
                    eval(E2,V2),
                    V is V1-V2.
eval(*(E1,E2),V) :- eval(E1,V1),
                    eval(E2,V2),
                    V is V1*V2.
eval(/(E1,E2),V) :- eval(E1,V1),
                    eval(E2,V2),
                    V is V1/V2.
eval(//(E1,E2),V) :- eval(E1,V1),
                     eval(E2,V2),
                     V is V1//V2.
eval(mod(E1,E2),V) :- eval(E1,V1),
                      eval(E2,V2),
                      V is V1 mod V2.
```

Μπορούμε να εκτελέσουμε προστακτικά προγράμματα που αποτελούνται μόνο από εντολές ανάθεσης με χρήση του **run**. Οι τιμές των ατόμων-μεταβλητών διατηρούνται και ανάμεσα σε διαδοχικές εκτελέσεις του **run**.

```
?- run([x = 2, y = 3, z = 5]).
yes

?- value(x,X).
X = 2
```

```
?- value(y,Y).
Y = 3

?- value(z,Z).
Z = 5

?- value(w,W).
no

?- run([w = x*y*z - (x+y+z) mod 4 + 32 // z]).
yes

?- value(w,W).
W = 34

?- run([x = x*2+4]).
yes

?-  value(x,X).
X = 8

?- allValues(L).
L = [y=3,z=5,w=34,x=8]
```

∎

Παράδειγμα 53: Λαβύρινθος.

Στο παράδειγμα αυτό θα σχεδιάσουμε ένα πρόγραμμα το οποίο θα βρίσκει τη διαδρομή για να βγούμε από έναν λαβύρινθο. Ο λαβύρινθος περιγράφεται ως μία λίστα από ζεύγη ακεραίων, που περιγράφουν τις ελεύθερες θέσεις στο λαβύρινθο, δηλαδή τα σημεία από τα οποία μπορούμε να περάσουμε. Μία άλλη λίστα της ίδιας μορφής περιγράφει τα σημεία του λαβύρινθου στα οποία υπάρχει έξοδος. Η αρχική θέση περιγράφεται από ένα ζεύγος ακεραίων. Επιτρέπεται να κινούμαστε μόνο οριζόντια ή κατακόρυφα (όχι διαγώνια) και μόνο προς ελεύθερες θέσεις.

Χρησιμοποιούμε το `assert` για να εισάγουμε στη βάση δεδομένων προτάσεις που δηλώνουν τις ελεύθερες θεσεις και τις εξόδους . Αυτό γίνεται χρησιμοποιώντας τα κατηγορήματα `initFree` και `initExits`, τα οποία εισάγουν στη βάση δεδομένων γεγονότα της μορφής `free(I,J)` και `exit(I,J)`. Επίσης χρησιμοποιούμε το `retract` για να διαγράφουμε από τη βάση δεδομένων το γεγονός `free(I,J)` όταν επισκεφτόμαστε το σημείο `[I,J]`. Αυτό γίνεται ώστε να αποφεύγονται κύκλοι στο λαβύρινθο. Το `clear` χρησιμοποιείται στην αρχή ώστε να διαγράψει προηγούμενους ορισμούς των `free` και `exit`.

```
solvePuzzle(FreeBlocks,Exits,Start,[start|Path]):-
                        clear,
                        initFree(FreeBlocks),
                        initExits(Exits),
                        findPath(Start,Path).
```

```
initFree([]).
initFree([[I,J]|T]) :- integer(I),
                       integer(J),
                       assertz(free(I,J)),
                       initFree(T).

initExits([]).
initExits([[I,J]|T]) :- integer(I),
                        integer(J),
                        assertz(exit(I,J)),
                        initExits(T).

findPath([I,J],[[I,J],exit]) :- exit(I,J).
findPath(A,[A,D|P]) :- canMove(A,B,D),
                       B = [I,J],
                       retract(free(I,J)),
                       findPath(B,P).

canMove([I,J],[K,J],up) :- K is I-1.
canMove([I,J],[K,J],down) :- K is I+1.
canMove([I,J],[I,K],left) :- K is J-1.
canMove([I,J],[I,K],right) :- K is J+1.

clear :- retract(free(I,J)),
         clear.
clear :- clear2.

clear2 :- retract(exit(I,J)),
          clear2.
clear2.
```

Παράδειγμα εξόδου από λαβύρινθο:

```
?- solvePuzzle([ [1,1],[1,2],[1,3],      [1,5],[1,6],
                 [2,1],                  [2,5],
                       [3,2],[3,3],      [3,5],
                             [4,3],      [4,5],
                 [5,1],[5,2],[5,3],[5,4],[5,5],
                 [6,1],                  [6,5],[6,6]
               ],
               [[1,1],[1,6]], [3,3], P).
P = [start,[3,3],down,[4,3],down,[5,3],right,[5,4],
    right,[5,5],up,[4,5],up,[3,5],up,[2,5],up,
    [1,5],right,[1,6],exit]
```