Η ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ PROLOG

(ΜΕΘΟΔΟΣ ΤΑΧΕΙΑΣ ΕΚΜΑΘΗΣΗΣ ΒΑΣΕΙ ΠΑΡΑΔΕΙΓΜΑΤΩΝ)

Έκδοση 2.0

Κυριάκος Σγάρμπας, Επίκ.Καθηγητής
Εργαστήριο Ενσύρματης Τηλεπικοινωνίας, Τμήμα Ηλεκτρολόγων Μηχανικών & Τεχνολογίας Υπολογιστών, Πανεπιστήμιο Πατρών

ΤΙΤΛΟΣ ΕΝΤΥΠΟΥ: Η Γλώσσα Προγραμματισμού Prolog
ΕΙΔΟΣ ΕΝΤΥΠΟΥ: Σημειώσεις Μαθήματος
ΠΕΡΙΕΧΟΜΕΝΟ:
ΟΝΟΜΑ ΑΡΧΕΙΟΥ: PROLOG.PDF
ΕΚΔΟΣΗ ΕΝΤΥΠΟΥ: 2.0
ΗΜΕΡΟΜΗΝΙΑ: 21/8/2006
ΣΥΓΓΡΑΦΕΑΣ: Κ.Σγάρμπας
ΦΟΡΕΑΣ: Εργαστήριο Ενσύρματης Τηλεπικοινωνίας (ΕΕΤ) - Τμήμα Ηλεκτρολόγων Μηχανικών & Τεχνολογίας Υπολογιστών Πανεπιστημίου Πατρών
ΣΗΜΕΙΩΣΕΙΣ: Έκδοση 2.0: ISO-based, μετατροπή παραδειγμάτων σε SWI-Prolog.
Εκδόσεις 1.x: Edinburgh-based.

ΠΡΟΛΟΓΟΣ

Το έντυπο αυτό παρέχει μια σύντομη αλλά περιεκτική κάλυψη της γλώσσας προγραμματισμού Prolog για τις ανάγκες φοιτητών που παρακολουθούν το μάθημα της Τεχνητής Νοημοσύνης ή μαθήματα ανάλογου περιεχομένου.

Προηγούμενες εκδόσεις του παρόντος εντύπου (από το 1990) έχουν χρησιμοποιηθεί με τη μορφή σημειώσεων ή διδακτικών βοηθημάτων για τη διδασκαλία της γλώσσας Prolog στο Πανεπιστήμιο Πατρών (Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών – Μάθημα "Τεχνητή Νοημοσύνη"), στο Ελληνικό Ανοικτό Πανεπιστήμιο (ΠΣ Πληροφορική – ΘΕ ΠΛΗ31 "Τεχνητή Νοημοσύνη") καθώς και σε διάφορα μεταπτυχιακά προγράμματα σπουδών.

Η διδακτική προσέγγιση που έχει ακολουθηθεί βασίζεται στην βήμα-προς-βήμα παρουσίαση της γλώσσας με χρήση προσεκτικά επιλεγμένων παραδειγμάτων και στη συχνή εναλλαγή των θεωρητικών στοιχείων με τις πρακτικές λεπτομέρειες. Με αυτόν τον τρόπο ο φοιτητής είναι σε θέση από τις πρώτες κιόλας σελίδες να χρησιμοποιεί τη γλώσσα και να γράφει προγράμματα με σταδιακά αυξανόμενο βαθμό πολυπλοκότητας. Συνιστάται παράλληλα με τη μελέτη η δοκιμή των παραδειγμάτων σε κάποιον compiler ή interpreter της Prolog.

Από την έκδοση 2.0 του εντύπου η διάλεκτος της Prolog που έχει προτιμηθεί είναι η Standard ISO Prolog, ενώ έχουν παραμείνει και αναφορές στο Standard της Edinburgh Prolog για λόγους συμβατότητας. Τα παραδείγματα έχουν δοκιμαστεί στην SWI-Prolog.

Σε περίπτωση που διαπιστώσετε κάποιο σφάλμα σε αυτό το έντυπο, παρακαλώ ενημερώστε τον συγγραφέα με e-mail στην διεύθυνση sgarbas@upatras.gr, ώστε να διορθωθεί στην επόμενη έκδοση.

Περιεχόμενα

1. Εισαγωγή

Η γλώσσα προγραμματισμού Prolog δημιουργήθηκε στις αρχές της δεκαετίας του 1970 από τον Alain Colmerauer του Πανεπιστημίου της Μασσαλίας και θεωρείται ακόμα και σήμερα μια από τις πιο επιτυχημένες γλώσσες για προγραμματισμό εφαρμογών Τεχνητής Νοημοσύνης. Η Prolog έχει εντελώς διαφορετική φιλοσοφία από τις γνωστές γλώσσες γενικού σκοπού (Pascal, Basic, C, FORTRAN, κλπ). Στις παραπάνω γλώσσες ο προγραμματισμός είναι ***διαδικαστικός*** (ή ***αλγοριθμικός***), δηλαδή λέμε στη γλώσσα να εκτελέσει μια αυστηρά καθορισμένη ακολουθία ενεργειών (εντολών) προκειμένου να βρεί τη λύση στο πρόβλημά μας. Φυσικά, αυτό προϋποθέτει ότι εμείς (οι προγραμματιστές) γνωρίζουμε εκ των προτέρων τον αλγόριθμο, δηλαδή τον τρόπο που λύνεται το πρόβλημα και θα μπορούσαμε ενδεχομένως να το λύσουμε χωρίς υπολογιστή χρησιμοποιώντας μολύβι, χαρτί και πολύ χρόνο. Καταλαβαίνουμε λοιπόν ότι ο αλγοριθμικός προγραμματισμός χρησιμοποιεί τον υπολογιστή σαν μια γρήγορη αριθμομηχανή μάλλον, παρά σαν ένα "έξυπνο" μηχάνημα. Και αυτή την έννοια έχει η χρησιμότητα του αναλυτή εφαρμογών. Είναι ο άνθρωπος που βρίσκει τους αλγορίθμους, είναι αυτός που λύνει το πρόβλημα και όχι ο υπολογιστής.

Στην προσπάθειά μας να μεταφέρουμε στον υπολογιστή και το φορτίο της ανάλυσης, οφείλεται η ανάπτυξη του ***δηλωτικού*** προγραμματισμού. Για να λύσουμε το πρόβλημα δε χρειάζεται να γνωρίζουμε εκ των προτέρων κάποιον αλγόριθμο. Αρκεί να το ορίσουμε **πλήρως** και ο υπολογιστής αναλαμβάνει να το λύσει.

Η Prolog είναι μια γλώσσα δηλωτικού προγραμματισμού. Από τη στιγμή που θα της πούμε ποιό είναι το πρόβλημα, αναλαμβάνει να κάνει πλήρη διερεύνηση και να βρεί όλες τις δυνατές λύσεις του. Αυτό το πετυχαίνει με τη βοήθεια μιας ενσωματωμένης μηχανής αναζήτησης τύπου depth-first η οποία ενεργοποιείται αυτόματα κάθε φορά που ρωτάμε κάτι τη γλώσσα. Βέβαια η δυσκολία τώρα έχει μεταφερθεί από την εύρεση του κατάλληλου αλγορίθμου, στη δημιουργία του σωστού και πλήρους ορισμού του προβλήματος. Εκ πρώτης όψεως λοιπόν, δε μπορεί να πεί κανείς ότι κερδίσαμε κάτι ιδιαίτερα σημαντικό με την εισαγωγή του δηλωτικού προγραμματισμού. Αποκτήσαμε όμως ένα δεύτερο εργαλείο. Τα κλασσικά προβλήματα στα οποία μπορούμε να βρούμε αλγόριθμο επίλυσης θα συνεχίσουμε να τα αντιμετωπίζουμε με τις κλασσικές αλγοριθμικές γλώσσες. Για τα προβλήματα που είτε δε μπορούμε να βρούμε αλγόριθμο είτε κωδικοποιούνται πιό εύκολα με τον δηλωτικό προγραμματισμό, θα χρησιμοποιούμε γλώσσες σαν την Prolog.

Η Prolog λοιπόν, ως δηλωτική γλώσσα που είναι, χαρακτηρίζεται από μια "αυτενέργεια" που μερικές φορές ξενίζει τον προγραμματιστή που έχει συνηθίσει στον πλήρη έλεγχο που προσφέρουν οι αλγοριθμικές γλώσσες. Χαρακτηριστικό παράδειγμα είναι ότι σε προβλήματα με πολλές λύσεις, το ποιά λύση θα βρεθεί πρώτη και ποιά δεύτερη είναι κάτι που το αποφασίζει η γλώσσα (ο εσωτερικός μηχανισμός αναζήτησης) κι όχι ο προγραμματιστής. Βέβαια, μόλις ο προγραμματιστής φτάσει στο σημείο να καταλαβαίνει τον τρόπο με τον οποίο "σκέφτεται" η γλώσσα, θα μπορεί να προβλέψει τη σειρά των απαντήσεων και να κατευθύνει την Prolog να επιλέξει τη σειρά που επιθυμεί ο ίδιος, τροποποιώντας όμως τον ορισμό του προβλήματος και όχι τον μηχανισμό αναζήτησης. Από αυτό το γεγονός φαίνεται και η μεγάλη διαφορά που έχουν οι δηλωτικές γλώσσες σε σύγκριση με τις αλγοριθμικές: ο προγραμματιστής έχει ένα λογικό "εργαλείο" που για κάποιες αρχικές συνθήκες του δίνει κάποιο αποτέλεσμα. Για να πάρει το επιθυμητό αποτέλεσμα δεν επεμβαίνει στο ίδιο το εργαλείο (πρόγραμμα) αλλά αλλάζει τις αρχικές συνθήκες (δηλώσεις του προβλήματος). Το σύνολο των δηλώσεων με τις οποίες ορίζεται ένα πρόβλημα, το λέμε

(καταχρηστικά) "πρόγραμμα".

Σύμφωνα με τα όσα είπαμε ως τώρα, το βασικό πρώτο βήμα για να μάθει κανείς Prolog είναι να καταλάβει τον ιδιαίτερο τρόπο με τον οποίο η γλώσσα αυτή "σκέφτεται" και ενεργεί. Αυτό θα το δούμε στη συνέχεια, βήμα-βήμα, βάσει παραδειγμάτων. Τα παραδείγματα είναι βασισμένα στο ISO Standard και έχουν δοκιμαστεί στην SWI-Prolog[1]. Εδώ πρέπει να εξηγήσουμε ότι για τις εκδόσεις Prolog υπάρχουν δυο standards. Υπάρχει το παλιότερο Edinburgh-standard και το πιο σύγχρονο ISO-standard. Τα δυο standards διαφέρουν κυρίως στα ονόματα των built-in συναρτήσεων που αναγνωρίζει η γλώσσα. Οι νέοι compilers ακολουθούν το ISO-standard, όμως για λόγους συμβατότητας πολλοί αναγνωρίζουν και το Edinburgh.

1 Η SWI-Prolog είναι μια πολύ αξιόλογη έκδοση της Prolog που έχει αναπτυχθεί στο Πανεπιστήμιο του Amsterdam και διατίθεται δωρεάν από το site http://www.swi-prolog.org

2. Ένα Απλό Πρόγραμμα

Ο καλύτερος τρόπος για να μάθει κανείς μια γλώσσα προγραμματισμού είναι να πειραματιστεί σε έναν υπολογιστή με τη βοήθεια του manual. Το έντυπο αυτό σε καμμία περίπτωση δεν αντικαθιστά το manual της γλώσσας, αλλά έχει γραφτεί με μορφή "παραδειγμάτων συνεχούς ροής" προσπαθώντας να εξομοιώσει τον διάλογο με τον υπολογιστή, έτσι ώστε αν μετά την ανάγνωση αυτού του κειμένου κάποιος βρεθεί αντιμέτωπος με έναν compiler Prolog να αισθάνεται σε αρκετά πλεονεκτική θέση.

Στα παραδείγματα που ακολουθούν σημειώνουμε με έντονους χαρακτήρες τόσο τα προγράμματα, όσο και οποιαδήποτε inputs δίνουμε στη γλώσσα.

Οι παρακάτω τρείς γραμμές[1] αποτελούν ένα πρόγραμμα Prolog:

```
man(peter).
man(jimmy).
woman(helen).
```

Με αυτές τις τρείς εντολές δηλώνουμε ότι ο Peter είναι άνδρας (man), όπως επίσης και ο Jimmy, ενώ η Helen είναι γυναίκα (woman). Παρατηρήστε ότι **κάθε εντολή στην Prolog τελειώνει με τελεία**. Αυτός είναι απαράβατος κανόνας. Κάθε μιά από τις τρείς παραπάνω δηλώσεις-εντολές λέγεται ***γεγονός*** ή ***fact*** ή πιό γενικά ***συνάρτηση***[2] και αποτελείται από ένα ***κατηγόρημα*** (αλλιώς ***predicate*** ή ***functor*** – εδώ κατηγορήματα είναι τα man και woman) και ένα ***όρισμα*** ή ***argument*** (peter, jimmy, helen). Μια συνάρτηση μπορεί να έχει πολλά ορίσματα. Το πλήθος τους λέγεται ***arity***. Για τη συνάρτηση man το arity είναι 1 και συμβολίζουμε[3]: man/1

Αφού γράψουμε το παραπάνω πρόγραμμα σε έναν editor και πούμε στην Prolog να το εκτελέσει (θα εξηγήσουμε στην ενότητα 2 πώς γίνονται όλα αυτά), θα βρεθούμε μπροστά στο ***prompt*** της Prolog:

```
?-
```

Τώρα η Prolog περιμένει τις ερωτήσεις μας. Μπορούμε για παράδειγμα να επιβεβαιώσουμε ότι έχει καταλάβει τις δηλώσεις που κάναμε:

```
?- man(peter).
Yes

?- woman(helen).
Yes
```

Σε κάθε μια από τις ερωτήσεις μας η Prolog απαντάει με ένα "Yes" αφού συμφωνούν με τα facts που γνωρίζει (το πρόγραμμά μας). Μπορούμε ακόμα να ρωτήσουμε και για γεγονότα που δεν έχουν

1 Στα παραδείγματα που ακολουθούν σημειώνουμε με έντονους χαρακτήρες τόσο τα προγράμματα, όσο και οποιαδήποτε inputs δίνουμε στη γλώσσα.

2 Κάτω από την έννοια **συνάρτηση** περιλαμβάνονται και οι κανόνες, στους οποίους θα αναφερθούμε στην ενότητα 7.

3 Επειδή η Prolog επιτρέπει τον ορισμό διαφορετικών συναρτήσεων με ίδιο functor αλλά διαφορετικά arities, το πλήρες όνομα μιας συνάρτησης περιέχει και το αντίστοιχο arity. Έτσι, οι συναρτήσεις func/1 και func/2 είναι διαφορετικές.

σχέση με τις αρχικές μας δηλώσεις:

```
?- man(helen).
No

?- woman(jimmy).
No

?- woman(jenny).
No
```

Η απάντηση της Prolog είναι "No". Προσέξτε ότι στην τρίτη ερώτηση (για την Jenny) η απάντηση είναι αρνητική ενώ πιό λογικοφανής θα ήταν η απάντηση: "ΔΕΝ ΞΕΡΩ", αφού στο πρόγραμμά μας δεν έχουμε πεί τίποτα για την Jenny. Βλέπουμε λοιπόν ότι η Prolog έχει **δύο *επίπεδα λογικής***: αν κάτι το γνωρίζει, απαντάει "Yes" (η πρόταση είναι TRUE). Αν κάτι δεν το γνωρίζει απαντάει "No" (η πρόταση είναι FALSE). Έτσι λοιπόν και στις δυο πρώτες ερωτήσεις η Prolog δεν απαντά "no" βάσει κάποιου συλλογισμού της μορφής: "γνωρίζω ότι το woman(helen) είναι TRUE, άρα το man(helen) θα είναι FALSE, επομένως απαντώ No", αλλά λέει: "θέλω να ελέγξω αν το man(helen) είναι TRUE. Υπάρχει το man(helen) μέσα στις δηλώσεις του προγράμματος; Όχι. Επομένως το man(helen) είναι FALSE και απαντώ No". Αν δηλαδή το πρόγραμμά μας είχε και μια τέταρτη δήλωση που θα έλεγε ότι το man(helen) είναι γεγονός, τότε η Prolog θα απαντούσε "Yes" και στο man(helen) και στο woman(helen). Εκ πρώτης όψεως αυτή η συμπεριφορά δεν δείχει και πολύ έξυπνη. Όμως, αναλογιστείτε ότι για την Prolog οι λέξεις man και woman που χρησιμοποιήσαμε ως predicate-names, δεν έχουν κανένα σημασιολογικό νόημα. Θα μπορούσαμε να είχαμε γράψει rpd001 και qlx422 στη θέση τους. Δε μπορεί κανείς να αποκλείσει το ενδεχόμενο για το rpd001(helen) και το qlx422(helen) να είναι και τα δυο TRUE.

Στις ερωτήσεις μας μπορούμε ακόμα να χρησιμοποιήσουμε και ***λογικούς τελεστές***. Ο τελεστής άρνησης (NOT) στην Prolog συμβολίζεται[1] με "\+".

```
?- \+ man(peter).
No

?- \+ woman(peter).
Yes
```

Εδώ η Prolog εξετάζει τη λογική τιμή της παράστασης που ακολουθεί το \+, το man(peter) ή το woman(peter), και στη συνέχεια προσδίδει την αντίθετη τιμή στην ολική παράσταση. Αυτό φαίνεται χαρακτηριστικά στο παρακάτω παράδειγμα:

```
?- \+ woman(jenny).
Yes

?- \+ man(jenny).
```

1 Στο Edinburgh standard ο τελεστής άρνησης συμβολίζεται με "not". Πχ: "not woman(jenny)". Στην SWI-Prolog το not δουλεύει, αλλά ως όνομα συνάρτησης. Δηλαδή πρέπει να γράψουμε "not(woman(jenny))". Στο ISO standard ο τελεστής μετονομάστηκε σε \+ ώστε να μην μπερδεύει τους νεο-εισαγόμενους στη γλώσσα επειδή λειτουργεί λίγο διαφορετικά από ότι το σημασιολογικό NOT με το οποίο έχουμε συνηθίσει. Σε κάθε περίπτωση, η Prolog μας επιτρέπει να αλλάξουμε τα ονόματα των τελεστών ή να ορίσουμε νέους. Στην ενότητα 18 θα δούμε πώς γίνεται αυτό.

```
Yes
```

Οι απαντήσεις της Prolog είναι απόλυτα λογικές αν σκεφτούμε ότι η έννοια "jenny" είναι απολύτως άγνωστη στο πρόγραμμά μας. Αν αντί για "jenny" είχαμε τη λέξη "house", οι απαντήσεις θα ήταν ίδιες, και στην περίπτωση αυτή είναι εμφανές ότι η Prolog έχει δίκιο!

Με άλλα λόγια, ο τελεστής \+ στην πραγματικότητα έχει τη σημασία του "μη αποδείξιμου". Είναι μη αποδείξιμο ότι το woman(jenny) ισχύει; Ναι. Είναι μη αποδείξιμο ότι το man(jenny) ισχύει; Επίσης ναι.

Οι λογικοί τελεστές AND και OR συμβολίζονται στην Prolog με "**,**" και "**;**" αντίστοιχα:

```
?- man(peter), man(jimmy).
Yes
```

δηλαδή ισχύει **και** το man(peter) **και** το man(jimmy).

```
?- man(peter), \+ man(jimmy).
No
```

"No", επειδή το not man(jimmy) είναι FALSE.

```
?- man(peter); \+ man(jimmy).
Yes
```

Αλλά εδώ αρκεί που το man(peter) είναι TRUE, γιατί οι δυο συναρτήσεις είναι συνδεδεμένες με λογικό OR.

Εδώ πρέπει να πούμε ότι το "," εκτός από AND έχει και την έννοια του διαχωριστή ορισμάτων, όταν βρίσκεται μέσα στην παρένθεση μιας συνάρτησης. Πχ. **f(x,y)**. Σε αυτήν την περίπτωση το "," απλώς διαχωρίζει το x από το y. Δεν έχει την έννοια του x AND y. Συναρτήσεις με περισσότερα από ένα ορίσματα θα εξετάσουμε λεπτομερώς στην ενότητα 4.

Ενα δεύτερο σημείο που θα πρέπει να τονίσουμε είναι ότι στον κώδικα του προγράμματός μας δε μπορούμε να έχουμε γεγονότα ορισμένα με τελεστές. Δηλαδή κάτι τέτοιο:

```
\+ man(helen).
man(peter); man(jimmy).
```

είναι απαράδεκτο για πρόγραμμα Prolog, αφού οι ορισμοί δεν είναι **πλήρεις**. Πχ. για την helen λέμε τι δεν είναι, αλλά δε λέμε τι είναι![1]

Στα παραδείγματα που είδαμε ως τώρα, οι απαντήσεις της Prolog ήταν λακωνικές. Ενα "Yes" ή ένα

1 Δε μπορούμε να ορίσουμε κάτι με την άρνηση μιας έννοιας. Βέβαια υπάρχουν τρόποι να κάνουμε την Prolog να καταλάβει και τέτοιες ασαφείς έννοιες, αλλά απαιτούν πιο σύνθετο προγραμματισμό.

"no" ανάλογα με τη λογική τιμή της ερώτησης που κάναμε. Μπορούμε όμως να κάνουμε και ερωτήσεις που απαιτούν περισσότερες πληροφορίες. Για παράδειγμα, μπορούμε να ρωτήσουμε ποιούς άντρες αναγνωρίζει το πρόγραμμά μας:

```
?- man(X).
X = peter ;
X = jimmy ;
No
```

Με αυτή την ερώτηση συναντάμε για πρώτη φορά την έννοια της ***μεταβλητής***. Θα αναρωτηθήκατε ίσως γιατί τόσην ώρα γράφαμε τα κύρια ονόματα (peter, jimmy, helen) με μικρά γράμματα. Ο λόγος είναι ότι **κάθε λέξη με κεφαλαίο πρώτο γράμμα, η Prolog τη θεωρεί μεταβλητή**. Ετσι, το **peter** είναι μια σταθερά, ενώ το **Peter** θα ήταν μεταβλητή. Στην ερώτηση που κάναμε, το X στο man(X) είναι επίσης μεταβλητή και η ερώτηση έχει την εξής έννοια: "*βρές τις κατάλληλες τιμές για το X, έτσι ώστε το man(X) να είναι TRUE*". Χαρακτηριστικό είναι ότι η Prolog βρίσκει **όλες** τις τιμές της μεταβλητής (ή των μεταβλητών, αν υπάρχουν πολλές) που ικανοποιούν την ερώτησή μας. Σημειώστε ότι η Prolog τυπώνει τις τιμές μια-μια. Γράφει μόνο την πρώτη, και στη συνέχεια περιμένει να πατήσουμε ένα πλήκτρο. Αν πατήσουμε το πλήκτρο ';' θα μας τυπώσει την επόμενη τιμή που ικανοποιεί την ερώτηση που κάναμε (ή θα γράψει 'No', αν δεν υπάρχει άλλη). Αν πατήσουμε το 'Space' δεν θα ψάξει για άλλη τιμή. Έτσι (πατώντας το ;), το X παίρνει πρώτα την τιμή 'peter', μετά την τιμή 'jimmy' και στο τέλος η Prolog απαντά 'No', εννοώντας ότι δεν υπάρχουν άλλες τιμές για τη μεταβλητή X που ικανοποιούν την ερώτηση.

Παρατηρήστε ότι η Prolog βρήκε πρώτα τη λύση X=peter και μετά την X=jimmy. Αυτό οφείλεται στη σειρά με την οποία είναι γραμμένα τα γεγονότα στο πρόγραμμά μας. Η σειρά λοιπόν έχει σημασία. Και η σειρά στο πρόγραμμα, αλλά και η σειρά που κάνουμε τις ερωτήσεις[1]:

```
?- man(X); woman(X).
X = peter ;
X = jimmy ;
X = helen ;
No

?- woman(X); man(X).
X = helen ;
X = peter ;
X = jimmy ;
No
```

Στην πρώτη ερώτηση ζητήσαμε από την Prolog να βρεί πρώτα τους άντρες, ενώ στη δεύτερη ζητήσαμε πρώτα τις γυναίκες. Βλέπουμε λοιπόν ότι οι λογικοί τελεστές στην Prolog έχουν μια τάση να μη συμπεριφέρονται απόλυτα αντιμεταθετικά, όπως έχουμε συνηθίσει στη λογική. A AND B στην Prolog σημαίνει να εξεταστεί πρώτα το A και μετά το B, ενώ στη λογική δεν υπήρχε τέτοια υπόνοια. Και μπορεί να πεί κανείς: εντάξει, τί μας πειράζει η σειρά αφού το σύνολο των λύσεων είναι αυτό που έχει τελικά σημασία; Δυστυχώς η σειρά μερικές φορές επηρρεάζει και το σύνολο των λύσεων! Προσέξτε το ακόλουθο αξιομνημόνευτο παράδειγμα:

1 Αλλά δεν έχει σημασία η σχετική θέση των συναρτήσεων, πχ. αν στον ορισμό των γεγονότων γράφαμε πρώτα το **woman(helen)** και μετά τα **man(peter)** και **man(jimmy)**, οι απαντήσεις θα ήταν ίδιες.

```
?- man(X), \+ woman(X).
X = peter ;
X = jimmy ;
No

?- \+ woman(X), man(X).
No
```

Την πρώτη φορά που συναντά κανείς αυτή την περίπτωση, απορεί (και με το δίκιο του) γιατί έχει την εντύπωση ότι η Prolog είναι μια γλώσσα που (θα έπρεπε να) συμπεριφέρεται λογικά. Όμως η Prolog είναι στην πραγματικότητα μια διαδικαστική γλώσσα που **προσποιείται** ότι συμπεριφέρεται λογικά, χωρίς να τα καταφέρνει πάντα (όπως φάνηκε στο παράδειγμα). Μένει τώρα να κατανοήσουμε γιατί η Prolog αντέδρασε τόσο αρνητικά στη δεύτερη ερώτησή μας, ενώ υπήρχαν τιμές για το X που θα την ικανοποιούσαν. Στην προσπάθειά μας να καταλάβουμε το συλλογισμό της Prolog, τής απευθύνουμε τη μισή μόνο από την επίμαχη ερώτηση:

```
?- \+ woman(X).
No
```

Δεν υπάρχει τιμή για το X η οποία κάνει το woman(X) FALSE, ώστε να κάνει το \+ woman(X) TRUE; Βεβαίως και υπάρχει, το X=peter. Αλλά η Prolog ποτέ δε βρίσκει αυτή την τιμή, γιατί "σκέφτεται" ως εξής:

- Θέλω να ικανοποιήσω το \+ **woman(X)**.
- Ελέγχω πρώτα το **woman(X)**.
- Ψάχνω λοιπόν όλα τα γεγονότα του προγράμματος που έχουν για όνομα συνάρτησης (κατηγόρημα) τη λέξη 'woman'.
- Βρίσκω μόνο ένα, το **woman(helen)**.
- Επομένως, το **woman(X)** γίνεται TRUE με X=helen (μοναδική τιμή).
- Αρα το \+ **woman(X)** γίνεται FALSE για X=helen και αφού το X στο **woman(X)** δε μπορεί να πάρει άλλη τιμή, ούτε το \+ **woman(X)** μπορεί να πάρει άλλη τιμή, επομένως είναι πάντα FALSE και γράφω "No".

Βλέπουμε λοιπόν ότι υπάρχουν εκφράσεις όπως η **woman(X)** που μπορούν να επιστρέψουν κάποιες τιμές για τα ορίσματά τους (***γεννήτριες τιμών*** ή ***εκφράσεις-γεννήτριες***) και εκφράσεις όπως η \+ **woman(X)** που δεν επιστρέφουν τιμές για τα ορίσματά τους. Αυτές (οι δεύτερες) μπορούν να χρησιμοποιηθούν μόνο για να ελέγξουν αν οι τιμές των ορισμάτων τους είναι αποδεκτές ή όχι (***ελεγκτές τιμών*** ή ***εκφράσεις-ελεγκτές***).

Έτσι, στην ερώτηση:

```
?- man(X), \+ woman(X).
X = peter ;
X = jimmy ;
No
```

η **man(X)** είναι έκφραση-γεννήτρια ενώ η \+ **woman(X)** είναι έκφραση-ελεγκτής. Η **man(X)**

επιστρέφει τις τιμές 'peter' και 'jimmy' για το X και η \+ **woman(X)** ελέγχει αν οι τιμές αυτές την ικανοποιούν. Αντίθετα, στην ερώτηση:

```
?- \+ woman(X), man(X).
No
```

η \+ **woman(X)** ως έκφραση-ελεγκτής που είναι, δε μπορεί να δώσει τιμές στο X. Θα μπορούσε μόνο να ελέγξει αν μια τιμή που ήδη έχει πάρει η μεταβλητή X είναι ικανοποιητική ή όχι. Όμως η X δεν έχει ακόμα καμμιά τιμή. Επομένως η \+ **woman(X)** γίνεται FALSE, και η Prolog ούτε καν ελέγχει τη **man(X)** αφού συνδέεται με τη \+ **woman(X)** με λογικό AND (και TRUE να γίνει η **man(X)** δεν αλλάζει τίποτα, αφού ολόκληρη η ερώτηση έχει πια αποτύχει).

Χαρακτηριστικό είναι ότι παρόμοια ερώτηση με OR:

```
?- \+ woman(X); man(X).
X = peter ;
X = jimmy ;
No
```

προχωράει πέρα από το ***fail***[1] της \+ **woman(X)** και βρίσκει τιμές που ικανοποιούν τη **man(X)**.

Και ερχόμαστε με ένα τελευταίο παράδειγμα να απορρίψουμε την αρχή εκείνη της λογικής που λέει ότι NOT(NOT(A))=A, ή αλλιώς: **δυο αρνήσεις ισοδυναμούν με μια κατάφαση**.

```
?- \+ \+ woman(helen).
Yes
```

Απόλυτα λογικό, η απάντηση είναι ίδια όπως και στο:

```
?- woman(helen).
Yes
```

Ας γενικεύσουμε την ερώτηση:

```
?- woman(X).
X = helen ;
No
```

...και τώρα αν βάλουμε και τη διπλή άρνηση, η απάντηση θα πρέπει να είναι η ίδια:

```
?- \+ \+ woman(X).
X = _G278 ;
No
```

...ή όχι; Και πρώτα-πρώτα να πούμε ότι εκείνο το περίεργο "_G278" (καθώς και οποιαδήποτε

1 fail=αποτυχία. Οι έκφράσεις: "η συνάρτηση έκανε fail", "χτύπησε fail" ή "έγινε FALSE" χρησιμοποιούνται πιο συχνά μεταξύ των προγραμματιστών της Prolog από την πιο άμεση: "η συνάρτηση είναι FALSE", επειδή για τις περισσότερες συναρτήσεις δεν είναι εμφανές από την αρχή αν θα καταλήξουν σε TRUE ή FALSE. Το "έγινε fail" υπονοεί μια διαδικασία ελέγχου: αρχικά δεν ξέραμε τί είναι, στη συνέχεια έκανε fail, άρα είναι FALSE. Ενώ το "είναι FALSE" σημαίνει ότι είναι ολοφάνερα FALSE εξ' αρχής.

έκφραση που αποτελείται από ένα ***underscore***[1] ακολουθούμενο από έναν τετραψήφιο κωδικό) είναι μια "***εσωτερική μεταβλητή***"[2] της Prolog, ένας καταχωρητής του οποίου την ύπαρξη θα έπρεπε να αγνοούμε, κάτω από κανονικές συνθήκες. Εδώ τί ακριβώς έγινε και εμφανίστηκε ως τιμή στη μεταβλητή X; Ας δούμε πάλι πώς "σκέφτηκε" η Prolog:

Κατ' αρχήν, δεν επιχείρησε την ενέργεια που θα έκανε αμέσως ένας άνθρωπος: δυο αρνήσεις στη σειρά διαγράφονται και μένει μόνο το **woman(X)**. Και σωστά δεν το έπραξε επειδή το \+ δεν είναι ακριβώς άρνηση αλλά σημαίνει "μη αποδείξιμο". Έτσι προσπάθησε να εξαντλήσει τα περιθώρια απόδειξης της ερώτησης που κάναμε.

Ξεκίνησε από το **woman(X)** και βρήκε κάποια τιμή για το X (X=helen). Την τιμή αυτή την αποθήκευσε προσωρινά στην εσωτερική μεταβλητή "_G278". Κατόπιν προσπάθησε να δεί αν η έκφραση \+ **woman(X)** είναι TRUE για X=helen. Απέτυχε, επομένως η _G278 αδειάζει για να δεχτεί την επόμενη τιμή του X που κάνει TRUE την \+ woman(X). Τέτοια τιμή δε βρέθηκε, και η \+ **woman(X)** γίνεται FALSE. Στη συνέχεια εξετάζεται η έκφραση \+ \+ **woman(X)** η οποία είναι TRUE (αφού η \+ **woman(X)** βρέθηκε FALSE) για την τιμή του X που ήδη υπάρχει στην _G278. Αλλά η _G278 έχει αδειάσει και η Prolog αφού δε βρίσκει τιμή για το X να επιστρέψει, αρκείται να μας δώσει το όνομα της εσωτερικής μεταβλητής που θά 'πρεπε να έχει την τιμή του X.

Σημείωση: Αν ο παραπάνω συλλογισμός σας φάνηκε περίπλοκος ή δυσνόητος μην ανησυχείτε γιατί οι προγραμματιστές της Prolog ΣΕ ΚΑΜΜΙΑ ΠΕΡΙΠΤΩΣΗ δεν αναλύουν με αυτόν τον τρόπο τη λογική των προγραμμάτων τους. Ο συλλογισμός δόθηκε μόνο για λόγους πληρότητας και για να δείξει ότι η Prolog μερικές φορές δε συμπεριφέρεται όσο "λογικά" θα περιμέναμε, αλλά πάντοτε υπάρχει κάποιος λόγος για αυτό.

Εδώ και επτά περίπου σελίδες ασχολούμαστε με ένα πρόγραμμα τριών γραμμών το οποίο δεν έχει **ούτε μια εντολή της Prolog**. Πραγματικά, κοιτάξτε ξανά τον κώδικα:

```
man(peter).
man(jimmy).
woman(helen).
```

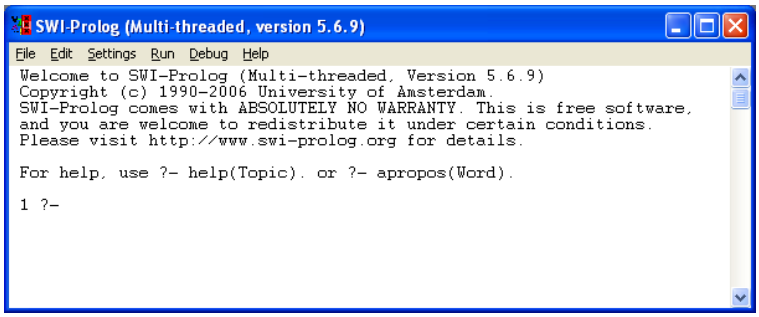
Αυτό είναι όλο κι όλο. Δεν έχει εντολές που τυπώνουν ή εισάγουν δεδομένα, δεν έχει μεταβλητές που παίρνουν τιμές, ακόμα και τα ονόματα στα κατηγορήματα και στα ορίσματα ήταν δικές μας επιλογές. Παρ' όλα αυτά είναι ένα πρόγραμμα Prolog. Και πώς φαίνεται ότι ένα πρόγραμμα είναι πρόγραμμα Prolog, αν όχι από τις εντολές του; Μα φυσικά από τη σύνταξη. Μπορεί να μη χρησιμοποιήσαμε καμμιά από τις πολυάριθμες built-in συναρτήσεις της Prolog, όμως δεν αποφύγαμε να χρησιμοποιήσουμε τις παρενθέσεις και τις τελείες. Ακόμα, φροντίσαμε τα ονόματα να είναι γραμμένα με μικρά γράμματα και όχι με κεφαλαία. Όλα αυτά τα στοιχεία αποτελούν το συντακτικό της Prolog το οποίο είναι τελικά αυτό που χαρακτηρίζει τη γλώσσα. Η Prolog λοιπόν είναι μια γλώσσα που χαρακτηρίζεται από το συντακτικό της, σε αντίθεση με άλλες γλώσσες προγραμματισμού που χαρακτηρίζονται από το λεξιλόγιό τους. Οχι πως έχει καμμία ιδιαίτερη σημασία αυτό από μόνο του, αλλά μας προετοιμάζει να γνωρίσουμε μια νέα φιλοσοφία στον τρόπο προγραμματισμού, μια φιλοσοφία που θα στηρίζεται περισσότερο στη μορφή παρά στις λέξεις, περισσότερο στο "πνεύμα" παρά στο "γράμμα" του κώδικα.

1 Το σύμβολο '_'.
2 Βλ.ενότητα 5.β.

3. Εξοικείωση με το Περιβάλλον της Prolog

Στη συνέχεια θα εξετάσουμε πώς τρέχει κανείς προγράμματα στο περιβάλλον της Prolog, πώς χρησιμοποιεί editors, και άλλες παρόμοιες τεχνικές πληροφορίες. Τα στοιχεία που θα δώσουμε αφορούν την SWI-Prolog, αλλά πολλά από αυτά είναι κοινά για τις περισσότερες εκδόσεις Prolog που ακολουθούν το ISO Standard.

Μόλις τρέξουμε την εφαρμογή της SWI-Prolog εμφανίζεται ένα παράθυρο όπως αυτό:



...και βρισκόμαστε στο γνωστό prompt[1]:

```
?-
```

Τώρα αν δώσουμε:

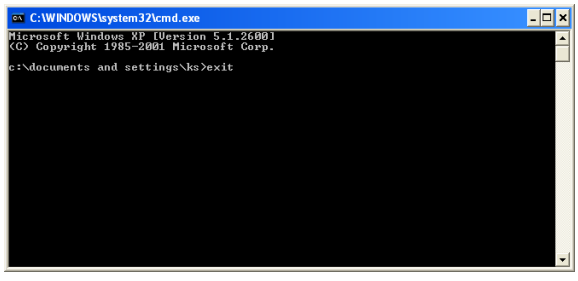
```
?- halt.
```

...τερματίζουμε την εφαρμογή και κλείνουμε το παράθυρο.

Αν θέλουμε να έχουμε πρόσβαση στο λειτουργικό σύστημα, γράφουμε:

```
?- shell.
```

...και ανοίγει ένα παράθυρο Command line:



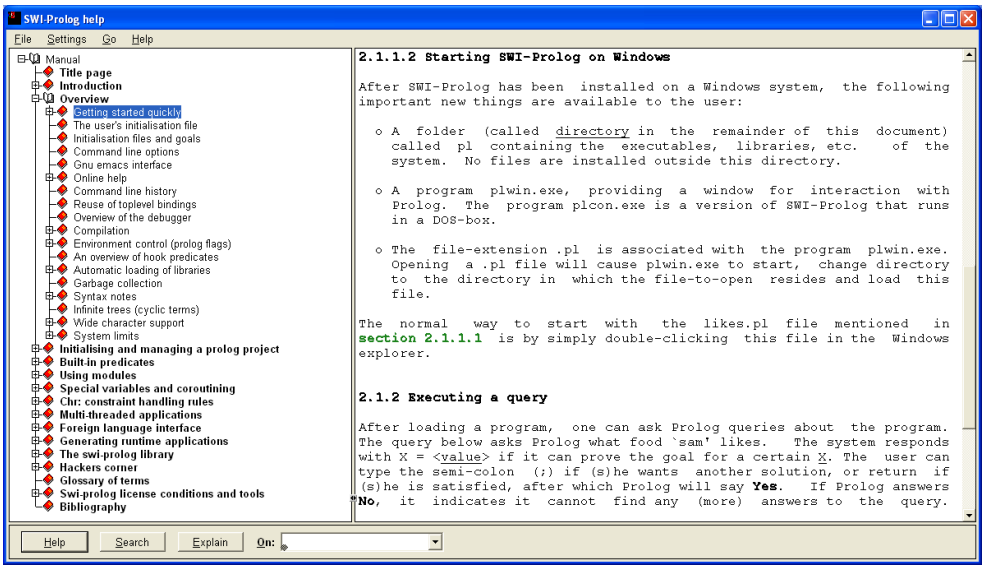
1 Η SWI-Prolog πριν το prompt βάζει έναν αριθμό (π.χ. "4 ?-"). Αυτός είναι ένας μετρητής που αυξάνεται κάθε φορά που δίνουμε μια νέα εντολή και χρησιμεύει για να έχουμε εύκολη πρόσβαση σε προηγούμενες εντολές μέσω του αρχείου ιστορικού (history file) που κρατάει η γλώσσα. Αν γράψετε στο prompt την εντολή '?- h.' θα δείτε όλες τις εντολές που έχετε δώσει στην Prolog από την στιγμή που την ξεκινήσατε, μαζί με τον αύξοντα αριθμό της κάθε μίας. Μπορείτε να επιλέξετε να ξανατρέξετε κάποια από αυτές είτε χρησιμοποιώντας τα βελάκια "πάνω"-"κάτω" του πληκτρολογίου, είτε γράφοντας στο prompt '?- !n.' (όπου n είναι ο αριθμός της εντολής).

Από εκεί μπορούμε να δώσουμε οποιεσδήποτε εντολές θέλουμε στο λειτουργικό σύστημα και όταν θέλουμε να το κλείσουμε αρκεί να γράψουμε **exit**.

Η SWI-Prolog έχει λεπτομερέστατο σύστημα on-line help. Γράφοντας στο prompt:

```
?- help.
```

...εμφανίζεται ένα νέο παράθυρο με το πλήρες manual της γλώσσας:



Το ίδιο συμβαίνει και επιλέγοντας “Help > Online Manual” από το μενού του παραθύρου της SWI-Prolog. Το μενού του Help μας δίνει ακόμα περισσότερες πηγές βοήθειας.

Τα αρχεία (source files) με προγράμματα Prolog έχουν συνήθως extension *.pl ή *.pro ή *.swi. Τα αρχεία είναι ένα απλά ASCII files και η γλώσσα τα διαβάζει ανεξάρτητα από το extension που μπορεί να έχουν. Για να γράψουμε ένα πρόγραμμα Prolog μπορούμε να χρησιμοποιήσουμε έναν οποιονδήποτε editor.

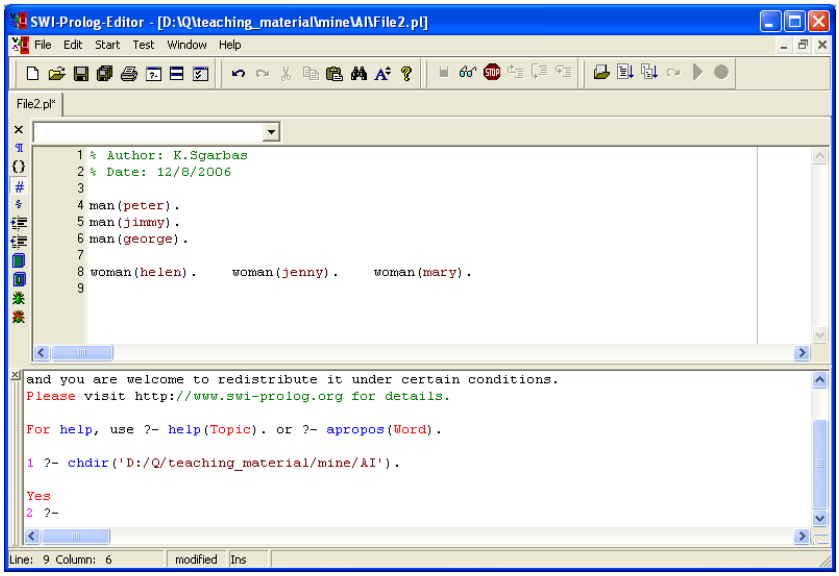
Σε όλες τις εκδόσεις Prolog υπάρχει ένας (όχι και τόσο εύχρηστος) built-in editor που ενεργοποιείται δίνοντας την εντολή consult/1 με argument τη λέξη **user**. Δεν ανοίγει καινούργιο παράθυρο, παρά μας δίνει ένα νέο prompt στο οποίο γράφουμε μια-μια τις εντολές του προγράμματός μας. Μόλις τελειώσουμε φεύγουμε από τον editor πατώντας Ctrl-Z:

```
?- consult(user).
|:    :
|:    :
|:    :
|: πρόγραμμα
|:    :
|:    :
|:    :
```

```
|:  ^Z
% user://1 compiled 0.01 sec, 516 bytes
Yes
```

Με τον τρόπο αυτόν το πρόγραμμά μας φορτώνεται κατ' ευθείαν στη μνήμη του υπολογιστή, χωρίς να σωθεί σε κάποιο file. Αυτή η μέθοδος είναι χρήσιμη μόνο για πολύ μικρά προγράμματα, συνήθως αν θέλουμε τα ελέγξουμε κάτι στη γλώσσα που δεν χρειάζεται στη συνέχεια να σώσουμε σε κάποιο αρχείο.

Για μεγαλύτερα προγράμματα, μπορούμε να επιλέξουμε “File>New” ή “File>Edit” από το παράθυρο της γλώσσας και να ενεργοποιήσουμε τον προεπιλεγμένο editor (για τα Windows συνήθως τον notepad). Όμως ειδικά για τη SWI-Prolog υπάρχει ελεύθερα διαθέσιμος ο SWI-Prolog-Editor (τον οποίο μπορείτε να κατεβάσετε από το site http://www.bildung.hessen.de/abereich/inform/skii/material/swing/indexe.htm) που δημιουργεί ένα περιβάλλον εργασίας για τη γλώσσα, όπως φαίνεται στο παρακάτω παράθυρο:



Από τη στιγμή που έχουμε γράψει ένα πρόγραμμα και το έχουμε σώσει στον δίσκο (έστω program.pl), για να το φορτώσουμε στην Prolog είτε επιλέγουμε από το μενού “File>Consult”, ή δίνουμε την εντολή:

```
?- consult(program).
% program compiled 0.01 sec, 896 bytes
Yes
```

Η Prolog μας ενημερώνει πόσο χρόνο χρειάστηκε να διαβάσει το πρόγραμμα και πόσο χώρο κατέλαβε στη μνήμη. Το extension εννοείται αν είναι *.pl. Αν όμως το source file είχε διαφορετικό extension, τότε το argument της consult/1 θα έπρεπε να είναι το πλήρες όνομα του file περικλειόμενο από απλά εισαγωγικά:

```
?- consult('program.ext').
% program.ext compiled 0.01 sec, 896 bytes
Yes
```

Εναλλακτικά, για το φόρτωμα των source files, μπορούμε να χρησιμοποιήσουμε το παρακάτω format:

```
?- [program].
% program compiled 0.02 sec, 572 bytes
Yes
```

...το οποίο μας παρέχει τη δυνατότητα να φορτώσουμε πολλά files με μια εντολή:

```
?- [program1,program2,program3,program4].
% program1 compiled 0.02 sec, 572 bytes
% program2 compiled 0.05 sec, 932 bytes
% program3 compiled 0.03 sec, 692 bytes
% program4 compiled 0.01 sec, 322 bytes
Yes
```

Κάτι που θα πρέπει να διευκρινίσουμε είναι ότι αφού φορτωθεί το πρόγραμμα, δε γίνεται καμμιά διαδικασία για να τρέξει. Αντίθετα με τις κλασσικές γλώσσες προγραμματισμού, δεν υπάρχει καμμιά εντολή που "εξαναγκάζει" το πρόγραμμα που βρίσκεται στη μνήμη να αρχίσει να εκτελείται. Στην Prolog, με την ολοκλήρωση του φορτώματος, το πρόγραμμα απλώς παραμένει στη μνήμη. Εκτελείται μόνο όταν ο χρήστης κάνει την κατάλληλη ερώτηση[1]. Έτσι, είναι δυνατόν με τη μια ερώτηση να εκτελεστεί ένα τμήμα μόνο του προγράμματος, με μια άλλη ερώτηση ένα άλλο τμήμα κ.ο.κ.

Για να δούμε το πρόγραμμα που υπάρχει κάθε στιγμή στη μνήμη, γράφουμε:

```
?- listing.
man(peter).
man(jimmy).
woman(helen).
Yes
```

...ενώ αν ενδιαφερόμαστε μόνο για μια συγκεκριμένη συνάρτηση:

```
?- listing(man).
man(peter).
man(jimmy).
Yes
```

Η σειρά με την οποία εμφανίζονται τα γεγονότα είναι η ίδια με την οποία εμφανίζονται στο source file για κάθε συνάρτηση, αλλά όχι απαραίτητα η ίδια μεταξύ των συναρτήσεων. Δηλαδή το man(peter) θα εμφανίζεται σίγουρα πριν το man(jimmy), αλλά οι συναρτήσεις man/1 δεν είναι βέβαιο αν θα εμφανίζονται πριν την woman/1 ή μετά (εξαρτάται από την έκδοση της Prolog, και δεν έχει σημασία για την εκτέλεση του προγράμματος).

1 Υπάρχει όμως και η δυνατότητα της auto-executable εντολής (βλ. ενότητα 11)

4. Facts με Περισσότερα Arguments

Ας εξετάσουμε το ακόλουθο πρόγραμμα:

```
mother_of(helen,peter).
mother_of(helen,jimmy).
mother_of(jenny,george).
```

Αυτή τη φορά έχουμε γεγονότα με δυο ορίσματα. Παρατηρήστε ότι τα ορίσματα χωρίζονται με κόμμα που εδώ δεν έχει την έννοια του λογικού-AND. Το παραπάνω πρόγραμμα θα μπορούσε να δηλώνει σχέσεις μητέρας-παιδιών. Το mother_of είναι το όνομα της συνάρτησης (functor) και φυσικά κανείς δε θα μας εμπόδιζε να είχαμε ορίσει τα ίδια γεγονότα με κάποια άλλη συνάρτηση (πχ. son_of) ή να είχαμε διαλέξει στα ορίσματα να βάζουμε πρώτα τα παιδιά και μετά τις μητέρες. Ολα αυτά είναι προσωπικές μας επιλογές και μπορούμε να τις κάνουμε ελεύθερα. Το μόνο που έχει σημασία είναι να διατηρούμε αυτές τις επιλογές σε ολόκληρο το πρόγραμμά μας.

Οι ερωτήσεις που κάνουμε εδώ θα μπορούν να είναι ερωτήσεις επιβεβαίωσης:

```
?- mother_of(helen,peter).
Yes
```

ή ανάκτησης πληροφοριών, πχ. "ποιά είναι τα παιδιά της helen;"

```
?- mother_of(helen,X).
X = jimmy ;
X = peter ;
No
```

"ποιά είναι η μητέρα του peter;"

```
?- mother_of(X,peter).
X = helen ;
No
```

Φυσικά η Prolog δεν καταλαβαίνει τις έννοιες "μητέρα" και "παιδί". Εξετάζει μόνο αν η μεταβλητή είναι στο πρώτο ή στο δεύτερο argument, ή και στα δύο:

```
?- mother_of(X,Y).
X = helen
Y = peter ;

X = helen
Y = jimmy ;

X = jenny
Y = george;
No
```

Η παραπάνω σχέση μάς δίνει όλα τα ζεύγη "μητέρα-παιδί" που γνωρίζει το πρόγραμμά μας. Το γεγονός ότι χρησιμοποιήσαμε δυο διαφορετικές μεταβλητές ήταν αποφασιστικής σημασίας γιατί αν διατυπώναμε την ερώτηση ως:

```
?- mother_of(X,X).
No
```

...η αντίρρηση της Prolog θα ήταν δίκαια: δεν έχουμε κάποια σχέση στην οποία το όνομα της μητέρας είναι το ίδιο με το όνομα του παιδιού.

Ας υποθέσουμε τώρα ότι θέλουμε να μάθουμε μόνο τις μητέρες και δεν θέλουμε από την Prolog να μας δείξει τα ονόματα των παιδιών. Η ερώτηση θα είναι:

```
?- mother_of(X,_).
X = helen ;
X = helen ;
X = jenny ;
No
```

Εδώ συναντάμε για πρώτη φορά την ***ανώνυμη μεταβλητή*** που συμβολίζεται με ένα underscore ("_"). Βάζοντας underscore σε κάποια arguments, η Prolog δεν επιστρέφει τιμές για αυτά. Πχ. αν θέλαμε να ρωτήσουμε απλώς αν υπάρχουν στο πρόγραμμά μας σχέσεις "μητέρας-παιδιού", χωρίς να ενδιαφερόμαστε για τα ονόματα των προσώπων, θα γράφαμε:

```
?- mother_of(_,_).
Yes
```

Προσέξτε ότι όσες φορές κι αν χρησιμοποιήσουμε την ανώνυμη μεταβλητή σε μια ερώτηση, αυτή θα έχει την έννοια **διαφορετικής κάθε φορά μεταβλητής**. Στο παράδειγμά μας δηλαδή, το mother_of(_,_) δεν σημαίνει ότι τα δυο arguments πρέπει να έχουν την ίδια τιμή (όπως υπαινισσόταν η mother_of(X,X) που είδαμε προηγουμένως). Στην Prolog **μόνο η ανώνυμη μεταβλητή** έχει αυτή την ιδιότητα.

Φυσικά, η ερώτηση:

```
?- mother_of(_).
No
```

...αποτυγχάνει γιατί η mother_of έχει δυο ορίσματα κι όχι ένα.

Ας προσπαθήσουμε να ανακαλύψουμε ποιά από τα πρόσωπα του προγράμματός μας είναι αδέρφια. Με βάση την ως τώρα εμπειρία μας θα κάναμε την ερώτηση κάπως έτσι:

```
?- mother_of(X,Y1), mother_of(X,Y2).
```

"*Βρές δυο παιδιά (Y1 και Y2) που έχουν την ίδια μητέρα (X)*"

Και ενώ περιμέναμε να βρούμε μόνο τον peter και τον jimmy, το σύνολο των λύσεων παρουσιάζεται τελικά πολυπληθέστερο:

```
?- mother_of(X,Y1), mother_of(X,Y2).
X = helen
Y1 = peter
Y2 = peter ;

X = helen
Y1 = peter
Y2 = jimmy ;

X = helen
Y1 = jimmy
Y2 = peter ;

X = helen
Y1 = jimmy
Y2 = jimmy ;

X = jenny
Y1 = george
Y2 = george ;
No
```

Κλασσικό σφάλμα. Κατά την επικοινωνία μας με την Prolog πρέπει να συνηθίσουμε να είμαστε όσο πιο σαφείς γίνεται. Εδώ τίποτα δεν εμπόδισε τις μεταβλητές Y1 και Y2 να πάρουν την ίδια τιμή. Ετσι βγήκε ο καθένας αδερφός του εαυτού του! Διορθώνουμε, λοιπόν:

```
?- mother_of(X,Y1), mother_of(X,Y2), \+ Y1==Y2.
X = helen
Y1 = peter
Y2 = jimmy ;

X = helen
Y1 = jimmy
Y2 = peter ;
No
```

Το σύμβολο == δηλώνει ισότητα (ένα από τα τέσσερα είδη ισότητας που υπάρχουν στην Prolog και θα μας απασχολήσουν στην ενότητα 10).

Και επειδή το όνομα της helen δε μας ενδιαφέρει, ίσως προσπαθούσαμε να χρησιμοποιήσουμε την ανώνυμη μεταβλητή στη θέση της X:

```
?- mother_of(_,Y1), mother_of(_,Y2), \+ Y1==Y2.
Y1 = peter
Y2 = jimmy ;
```

```
Y1 = peter
Y2 = george ;

Y1 = jimmy
Y2 = peter ;

Y1 = jimmy
Y2 = george ;

Y1 = george
Y2 = peter ;

Y1 = george
Y2 = jimmy ;
No
```

Αλλά το αποτέλεσμα για μια ακόμα φορά δε συμφωνεί με τις προσδοκίες μας γιατί η ύπαρξη του X εξασφάλιζε ότι η μητέρα του Y1 είναι η ίδια με τη μητέρα του Y2 ενώ **οι ανώνυμες μεταβλητές είναι πάντα ανεξάρτητες**.

5. Prolog Objects

Οι σταθερές, οι μεταβλητές και οι αριθμοί που χρησιμοποιούμε στην Prolog λέγονται με ένα όνομα ***prolog objects*** και διακρίνονται σε:

α) CONSTANTS (Σταθερές)

Διακρίνονται με τη σειρά τους σε ***atoms*** και ***integers***. Τα atoms είναι ονόματα που αρχίζουν με μικρό πρώτο γράμμα ή περικλείονται σε απλά εισαγωγικά. Δεν είναι strings[1] γιατί δε μπορούν να αποσυντεθούν (κάτω από κανονικές συνθήκες) σε sub-atoms.

Οι ακόλουθες λέξεις **είναι** atoms:

```
john   jOHN   'JOHN'   b019   tell_mE   'c   d'
```

Οι ακόλουθες λέξεις **δεν είναι** atoms:

```
18ab   John   _alpha   tell-me
```

Για τους integers δε χρειάζονται ειδικές εξηγήσεις:

```
0   1   2   -41   6221
```

Η "καθαρή" Prolog δεν έχει πραγματικούς αριθμούς (reals). Δεν τους χρειάζεται. Οι μόνοι αριθμοί που έχουν έννοια στη λογική είναι οι ακέραιοι (integers). Επειδή όμως οι προγραμματιστές είναι εξαιρετικά ευαίσθητοι σε τέτοιου είδους "καινοτομίες", όλες σχεδόν οι εκδόσεις της Prolog που κυκλοφορούν, διαθέτουν κάποιες βιβλιοθήκες για χειρισμό πραγματικών αριθμών (εξαγωγή ριζών, τριγωνομετρικές συναρτήσεις, λογάριθμοι κτλ.)

β) VARIABLES (Μεταβλητές)

Μεταβλητή είναι οποιαδήποτε λέξη αρχίζει με κεφαλαίο λατινικό γράμμα ή με underscore και δεν περικλείεται σε εισαγωγικά. Παραδείγματα μεταβλητών:

```
Answer     Tell_me     WHAT     _get     _     _45E8
```

Τη σημασία της ανώνυμης μεταβλητής την έχουμε συζητήσει ήδη. Μεταβλητές που **αρχίζουν** με underscore **δεν** είναι ανώνυμες και **δεν** τυγχάνουν "ειδικού χειρισμού" από την Prolog. Προσοχή μόνο στις μεταβλητές που μετά το underscore ακολουθεί τετραψήφιος κωδικός. Έτσι συμβολίζει η Prolog τις εσωτερικές της μεταβλητές (εσωτερικοί καταχωρητές) και αν τύχει να χρησιμοποιούμε το ίδιο όνομα για μεταβλητή του προγράμματός μας, πολλά "διασκεδαστικά" γεγονότα μπορούν να συμβούν κατά την εκτέλεσή του.

1 Τα strings η Prolog τα θεωρεί ειδική περίπτωση λιστών γιαυτό και θα τα εξετάσουμε στην ενότητα 14 με τις λίστες.

γ) COMPOUND OBJECTS (Σύνθετα Αντικείμενα)

Μια συνάρτηση με ένα ή περισσότερα ορίσματα αποτελεί compound object, πχ:

```
time(23,15,20)
```

Τα ορίσματα ενός compound object μπορούν να είναι διαφορετικών τύπων:

```
date(wednesday,16,8,2006)
```

ή ακόμα και άλλα compound objects:

```
now(date(wednesday,16,8,2006),time(23,15,20))
```

Η Prolog χειρίζεται τα compound objects ως αυτόνομες οντότητες κι έτσι μπορεί μια μεταβλητή να πάρει τιμή ένα ολόκληρο compound object:

```
X = now(date(wednesday,16,8,2006),time(23,15,20))
```

6. Μερικές Εντολές

Η Prolog έχει μια πλούσια βιβλιοθήκη με συναρτήσεις-εντολές αλλά θα ξεφεύγαμε από το σκοπό μας αν ασχολούμασταν λεπτομερώς με όλες αυτές. Αντίθετα, θα εξηγήσουμε τις πιό συνηθισμένες (και χρήσιμες), αυτές που θα χρησιμοποιούμε στα παραδείγματά μας. Ηδη στην ενότητα 3 ασχοληθήκαμε με εντολές που αφορούσαν την είσοδο-έξοδο από το περιβάλλον της γλώσσας και την εκτέλεση προγραμμάτων. Εδώ θα περιγράψουμε μερικές βασικές εντολές για απεικόνιση πληροφοριών στην οθόνη.

Και πρώτα-πρώτα η εντολή write/1:

```
?- write(hello).
hello
Yes

?- write('hello there').
hello there
Yes

?- write('και Ελληνικά').
και Ελληνικά
Yes

?- X=hello, write(X).
hello
X = hello ;
Yes
```

Απλώς γράφει την τιμή που είναι καταχωρημένη στο μοναδικό της argument, χωρίς να αλλάζει γραμμή για την επόμενη εκτύπωση (το "Yes" της Prolog είναι αυτό που αλλάζει τη γραμμή στα παραπάνω παραδείγματα). Αυτό φαίνεται καθαρότερα αν γράψουμε διαδοχικές εντολές write/1:

```
?- write(one), write(two), write(three).
onetwothree
Yes
```

Βέβαια, μπορούμε να προσθέσουμε κενά με τη write/1:

```
?- write(one), write('    '), write(two), write('    three').
one    two    three
Yes
```

ή να χρησιμοποιήσουμε τη συνάρτηση tab/1: tab(X) σημαίνει "*γράψε X κενά*":

```
?- write(one), tab(4), write(two), tab(4), write(three).
one    two    three
Yes
```

Αλλαγή γραμμής εκτελεί η συνάρτηση nl/0:

```
?- write(one), nl, write(two), nl, write(three).
one
two
three
Yes
```

Και ακολουθούν δυο συναρτήσεις που η χρησιμότητά τους δε φαίνεται από την πρώτη στιγμή, η true/0 που επιτυγχάνει πάντα:

```
?- true.
Yes
```

...και η fail/0 που αποτυγχάνει πάντα:

```
?- fail.
No
```

Οι λίγες αυτές συναρτήσεις είναι αρκετές για να κατανοήσουμε τα παραδείγματα της επόμενης ενότητας.

7. Κανόνες Prolog

Τα προγράμματα που εξετάσαμε ως τώρα αποτελούνταν αποκλειστικά από γεγονότα (facts). Η δύναμη όμως της Prolog έγκειται στο συνδυασμό των γεγονότων με λογικούς κανόνες για τη δημιουργία νέων γεγονότων. Σε αυτήν την ενότητα θα ασχοληθούμε με τον τρόπο έκφρασης τέτοιων κανόνων στην Prolog.

Γενικά, ένας κανόνας στην Prolog έχει τη μορφή:

Head :- Body.

Ενα αριστερό μέλος (Head) συνδέεται με ένα δεξί μέλος (Body) με ένα ειδικό σύμβολο (:-) που ακριβώς επειδή συνδέει ένα Head με ένα Body λέγεται **neck**. Ο κανόνας τελειώνει με τελεία. Η σημασία του κανόνα είναι: "το Head ισχύει, αν ισχύει το Body". Παρατηρούμε λοιπόν ότι το neck ισοδυναμεί με λογικό-IF. Το Body είναι η υπόθεση και το Head είναι το συμπέρασμα.

Το Body μπορεί να είναι μια ακολουθία συναρτήσεων συνδεδεμένων με λογικούς τελεστές, ενώ το Head επιτρέπεται να είναι μόνο μία συνάρτηση. Διακρίνουμε εδώ μια αντιστοιχία των κανόνων Prolog με τα Horn clauses που γνωρίζουμε από τη Λογική. Τα γεγονότα της Prolog μπορούν να θεωρηθούν ως ειδική περίπτωση κανόνων χωρίς Body (το Head ισχύει, χωρίς καμμία υπόθεση) ή ακόμα και ως:

fact :- true.

Στην ενότητα 11 θα δούμε ότι επιτρέπεται ακόμα και να έχουμε "κανόνες" χωρίς Head, οι οποίοι έχουν μια ξεχωριστή ιδιότητα: το Body τους εκτελείται αυτόματα (autoexecutable goals). Ετσι ολοκληρώνουμε τη γενίκευση λέγοντας ότι κάθε εντολή ενός προγράμματος Prolog είναι της μορφής "Head :- Body." όπου ένα από τα Head, Body μπορεί να λείπει.

Αρκετά με τη θεωρία, ας δούμε ένα παράδειγμα:

```
mother_of(helen,peter).
mother_of(helen,jimmy).
mother_of(jenny,george).
brothers(X,Y):- mother_of(Z,X), mother_of(Z,Y), \+ X==Y.
brothers2:- brothers(X,Y).
brothers3:- brothers(X,Y), write(X), nl, write(Y), nl.
brothers4:- brothers3.
```

Το παραπάνω πρόγραμμα αποτελεί επέκταση του παραδείγματος που εξετάσαμε στην ενότητα 5. Εδώ εκτός από τα γεγονότα που δηλώνουν τις σχέσεις "μητέρα-παιδί", έχουμε ορίσει τη συνάρτηση brothers/2 με έναν κανόνα του οποίου το Body μάς είναι ήδη γνώριμο από την ενότητα 5. Ο κανόνας λέει ότι "ο X και ο Y είναι αδέρφια αν έχουν την ίδια μητέρα και είναι διαφορετικά πρόσωπα". Ορίσαμε ακόμα τις συναρτήσεις brothers2/0, brothers3/0 και brothers4/0 οι που βασίζονται πάνω στην brothers/2 και τη λειτουργία τους θα την εξετάσουμε αμέσως παρακάτω.

Και πρώτα-πρώτα να δούμε τι τιμές επιστρέφει η brothers/2:

```
?- brothers(X,Y).
X = peter
Y = jimmy ;
```

```
X = jimmy
Y = peter ;
No
```

Αν θυμάστε, στην ενότητα 5 είχαμε ρωτήσει την Prolog για το Body της brothers/2 και είχε απαντήσει:

```
?- mother_of(Z,X), mother_of(Z,Y), \+ X==Y.
Z = helen
X = peter
Y = jimmy ;

Z = helen
X = jimmy
Y = peter ;
No
```

Η helen δεν τυπώθηκε στην ερώτηση brothers(X,Y). Ο λόγος είναι ότι **η Prolog απαντάει μόνο για όσες μεταβλητές υπάρχουν στην ερώτηση που κάνουμε**. Για να υπολογίσει τις τιμές αυτές, ενδεχομένως να χρησιμοποιήσει πολύ περισσότερες μεταβλητές τις οποίες θα κρατήσει κρυφές από εμάς. Αυτό φαίνεται καλύτερα στην brothers2/0. Κοιτάξτε τον κανόνα: "η brothers2 είναι TRUE όταν η brothers(X,Y) είναι TRUE". Θα περιμέναμε δηλαδή τα ίδια αποτελέσματα αν ρωτούσαμε:

```
?- brothers2.
Yes
```

Αλλά η απάντηση είναι ένα σκέτο "Yes", παρόλο που η Prolog έχει ανακαλύψει τα αδέρφια. Απλώς στην ερώτησή μας δεν είχαμε καμμιά μεταβλητή και η Prolog μάς ενημέρωσε μόνο για τη λογική τιμή της. Βέβαια, αυτό δε σημαίνει ότι αν ρωτούσαμε:

```
?- brothers2(X,Y).
no
```

...θα παίρναμε απάντηση, γιατί η brothers2 έχει οριστεί ως συνάρτηση με μηδέν ορίσματα ενώ εδώ ρωτάμε για κάποια brothers2 με δυο ορίσματα. Μόνο αν είχαμε ορίσει στο πρόγραμμά μας την brothers2 ως:

```
brothers2(X,Y):- brothers(X,Y).
```

...θα παίρναμε τιμές για τα X και Y με την προηγούμενη ερώτησή μας.

Κοιτάξτε τώρα αυτό:

```
?- brothers3.
peter
jimmy
Yes
```

Ούτε η brothers3 έχει ορίσματα. Όμως η ίδια η συνάρτηση φροντίζει να τυπώσει τις τιμές των εσωτερικών της μεταβλητών με εντολές write/1. Το μόνο που γράφει η Prolog από μόνη της είναι το "Yes" (όπως και στην brothers2) δηλώνοντας ότι η συνάρτηση έχει γίνει TRUE. Παρατηρούμε ακόμα ότι το X πήρε την τιμή **peter** και το Y την τιμή **jimmy** αλλά όχι και αντίστροφα, όπως έγινε στην κλήση της brothers(X,Y), επειδή **η Prolog μας δίνει όλες τις δυνατές τιμές μόνο για τα ορίσματα της συνάρτησης που ρωτάμε**. Εδώ η συνάρτηση δεν είχε ορίσματα και η Prolog σταμάτησε μόλις η brothers3 έγινε TRUE.

Τέλος, η brothers4:

```
?- brothers4.
peter
jimmy
Yes
```

...καλεί απλώς την brothers3 γιαυτό και έχει την ίδια συμπεριφορά με αυτήν.

Οι κανόνες που είδαμε μας έκαναν να γνωρίσουμε μια άλλη πτυχή της Prolog: την προσπάθεια ικανοποίησης της ερώτησης που της υποβάλλουμε. Πράγματι, όταν κάνουμε μια ερώτηση, η Prolog προσπαθεί να καταλήξει σε TRUE (να ικανοποιήσει τη συνάρτηση). Βρίσκει λοιπόν έναν κανόνα του οποίου το Head ταιριάζει με την ερώτησή μας και προσπαθεί να ικανοποιήσει το Body του. Για κάθε συνάρτηση στο Body επαναλαμβάνει την ίδια διαδικασία, κ.ο.κ. ώσπου να καταλήξει στα γεγονότα. Βλέπουμε έτσι ότι κάθε κανόνας έχει δυό σημασίες, τη ***λογική*** (ορίζει τις συνθήκες κάτω από τις οποίες μια συνάρτηση θα γίνει TRUE) και τη ***διαδικαστική*** (ορίζει μια ακολουθία συναρτήσεων-διαδικασιών οι οποίες θα εκτελεστούν όταν κληθεί το Head του). Κάθε πρόγραμμα Prolog μπορεί να το εξηγήσει κανείς έτσι ή αλλιώς (λογικά ή διαδικαστικά) με όποιον τρόπο κάθε φορά γίνεται πιο κατανοητό.

Και για να εξοικειωθούμε λίγο ακόμα με τους κανόνες ας δούμε άλλο ένα "γενεαλογικό" παράδειγμα. Εστω ότι έχουμε γεγονότα της μορφής:

```
man(peter).
man(jimmy).
    :
    :
woman(helen).
woman(jenny).
    :
    :
parent_of(peter,jenny).
parent_of(helen,peter).
    :
    :
```

Θέλουμε με βάση τα παραπάνω γεγονότα να ορίσουμε κανόνες που θα δηλώνουν συγγένειες μεταξύ αυτών των προσώπων. Συμφωνούμε ότι οι συναρτήσεις που θα ορίσουμε θα έχουν τη μορφή **relation(X,Y)** και τη σημασία: "*ο X έχει τη συγγένεια* relation *ως προς τον Y*".

```
father_of(X,Y):- parent_of(X,Y), man(X).
mother_of(X,Y):- parent_of(X,Y), woman(X).
```

"*Ο Χ είναι πατέρας του Υ, αν είναι γονέας του Υ και είναι και άντρας*". Εντελώς αντίστοιχα ορίζεται και η mother_of/2.

```
son_of(X,Y):- parent_of(Y,X), man(X).
daughter_of(X,Y):- parent_of(Y,X), woman(X).
```

"*Ο Χ είναι γιός του Υ, αν ο Υ είναι γονέας του Χ και ο Χ είναι άντρας*". Προσέξτε πώς αλλάζει θέση το Χ με το Υ στη σχέση parent_of/2 επειδή στη son_of/2 θέλουμε το πρώτο argument να είναι το παιδί. Αντίστοιχα ορίζεται και η daughter_of/2.

```
grandfather_of(X,Y):- parent_of(X,Z), parent_of(Z,Y), man(X).
grandmother_of(X,Y):- mother_of(X,Z), parent_of(Z,Y).
```

"*Ο Χ είναι παππούς του Υ, αν ο Χ είναι γονέας κάποιου Ζ ο οποίος είναι γονέας του Υ και ο Χ είναι άντρας*". Η σχέση grandmother/2 ορίστηκε διαφορετικά: "*η Χ είναι γιαγιά του Υ, αν η Χ είναι μητέρα κάποιου Ζ ο οποίος είναι γονέας του Υ*". Παρατηρήστε ότι εδώ το **mother_of(X,Z)** αντικαθιστά το **parent_of(X,Z), woman(X)**, οπότε οι συναρτήσεις grandfather_of/2 και grandmother_of/2 είναι εντελώς όμοιες.

```
brother_of(X,Y):- parent_of(Z,X),parent_of(Z,Y),\+ X==Y,man(X).
sister_of(X,Y):- daughter_of(X,Z), parent_of(Z,Y), \+ X==Y.
```

Και εδώ χρησιμοποιήσαμε δυο είδη ορισμών. Η brother_of/2 ορίστηκε μόνο με facts: "*ο Χ είναι αδερφός του Υ, αν οι Χ και Υ έχουν κοινό γονέα, δεν είναι το ίδιο πρόσωπο και ο Χ είναι άντρας*". Η sister_of/2 ορίστηκε με τη βοήθεια ενός άλλου κανόνα, αυτού που ορίζει τη σχέση daughter_of/2: "*η Χ είναι αδερφή του Υ, αν η Χ είναι κόρη κάποιου Ζ ο οποίος είναι γονέας του Υ και οι Χ και Υ είναι διαφορετικά πρόσωπα*".

```
uncle_of(X,Y):- parent_of(Z,Y), brother_of(X,Z).
aunt_of(X,Y):- sister_of(X,Z), (father_of(Z,Y); mother_of(Z,Y)).
```

"*Ο Χ είναι θείος του Υ, αν είναι αδερφός του γονέα (Ζ) του Υ*". "*Η Χ είναι θεία του Υ, αν είναι αδερφή κάποιου (Ζ) ο οποίος είναι ή πατέρας ή μητέρα του Υ*".

Κάτι που ίσως παρατηρήσατε ήδη είναι η ***τοπικότητα*** (***locality***) των μεταβλητών του προγράμματός μας. Χρησιμοποιήσαμε τις ίδιες μεταβλητές για να ορίσουμε τα παιδιά, τους γονείς, τους παππούδες και τις θείες, χωρίς να φοβόμαστε μήπως μπλεχτούν οι τιμές τους κατά τη διάρκεια της εκτέλεσης. **Οι μεταβλητές στην Prolog είναι local μέχρι την τελεία**. Αυτό σημαίνει ότι το Χ στο Head του κανόνα θα πάρει την ίδια τιμή με το Χ στο Body του ίδιου κανόνα αλλά δεν έχει καμμία σχέση με το Χ που βρίσκεται σε άλλο κανόνα ή οπουδήποτε αλλού στο πρόγραμμα. **Δυο μεταβλητές είναι ίδιες όταν ανήκουν στον ίδιο κανόνα και έχουν το ίδιο όνομα**. Και για μια ακόμα φορά, **εξαίρεση αποτελεί η ανώνυμη μεταβλητή που ΠΟΤΕ δεν είναι ίδια με τον εαυτό της**.

Η τοπικότητα των μεταβλητών, η συμμετρία των συναρτήσεων (εξηγείται στην ενότητα 14) και η

ανεξαρτησία της εκτέλεσης του προγράμματος από τη σειρά δήλωσης των συναρτήσεων (την είδαμε ήδη στην ενότητα 2) αποτελούν τις σημαντικότερες διαφορές της Prolog με τις συνηθισμένες γλώσσες προγραμματισμού.

8. Backtracking (Οπισθοδρόμηση)

Ας εξετάσουμε τώρα το εξής πρόγραμμα:

```
man(peter).     man(jimmy).     man(george).
woman(helen).   woman(jenny).   woman(mary).
goal0:- man(X).
goal1:- man(X), write(X), nl.
goal2:- man(X), write(X), nl, fail.
goal3:- man(X), write(X), nl, woman(X).
goal4:- man(X), woman(X), write(X), nl.
```

Το πρώτο πράγμα που παρατηρούμε είναι ότι τα γεγονότα δεν τα έχουμε γράψει το ένα κάτω από το άλλο, αλλά στη σειρά. Η Prolog ενδιαφέρεται μόνο για τις τελείες στο τέλος της κάθε εντολής και όχι για new line. Αν και θα μπορούσαμε να γράφουμε και τους κανόνες του προγράμματός μας σε αυτό το στυλ, συνήθως γράφουμε έτσι μόνο τα γεγονότα για να διαβάζονται πιο εύκολα.

Και αρχίζουμε τις ερωτήσεις:

```
?- man(X).
X = peter ;
X = jimmy ;
X = george ;
No
```

..τίποτα το ιδιαίτερο, οι απαντήσεις είναι αυτές που περιμέναμε.

```
?- goal0.
Yes
```

Οπως έχουμε ξαναπεί, μια και η goal0/0 δεν έχει arguments, η Prolog περιορίζεται στο να μας πληροφορήσει μόνο για τη λογική τιμή της. Προσέξτε ότι ενώ η goal0/0 καλεί τη man/1 η οποία γίνεται TRUE με τρείς διαφορετικούς τρόπους, η απάντηση είναι σκέτο **Yes** κι όχι "**Yes Yes Yes**".

```
?- goal1.
peter
Yes
```

Εδώ η goal1/0 συμπεριφέρεται από λογική άποψη όπως και η goal0/0 που είδαμε προηγουμένως. Αυτό που προσπαθεί είναι να καταλήξει στο TRUE και αφού το πετυχαίνει με την πρώτη τιμή για το X (X=george) δεν προχωράει παρακάτω.

```
?- goal2.
peter
jimmy
george
No
```

Την goal2/0 την αναγκάζουμε να προχωρήσει στην ανίχνευση επιπλέον λύσεων με την αυθαίρετη προσθήκη ενός "fail" στο τέλος του ορισμού της. Ετσι πάντοτε θα αποτυγχάνει. Αλλά αυτό δεν το ξέρει η ίδια! Θα πρέπει κάθε φορά να δοκιμάζει όλες τις δυνατότητες για να καταλάβει ότι απέτυχε. Και φυσικά στο τέλος γράφει **No**. Αναλυτικότερα, εδώ η Prolog ενήργησε ως εξής:

> Προσπάθησε να ικανοποιήσει την goal2/0. Κάλεσε την man/1 που επέστρεψε με την τιμή **X=peter**, στη συνέχεια εκτέλεσε την write(peter) και την nl/0 (αυτές οι δύο επιτυγχάνουν πάντα) και τέλος έφτασε στη fail, η οποία φυσικά απέτυχε. Απέτυχε μήπως και η goal/2; Όχι ακόμα. Η Prolog υποθέτει ότι την αποτυχία την προκαλεί η επιλογή της μεταβλητής X=peter και θα προσπαθήσει να δεί μήπως με άλλη τιμή καταλήξει σε TRUE. Ετσι από το σημείο που βρίσκεται (τώρα βρίσκεται στο fail/0) γυρίζει μια κλήση πίσω (στην nl/0) και ελέγχει αν η κλήση αυτή μπορεί να γίνει TRUE και με άλλο τρόπο. Αυτό για την nl/0 δεν ισχύει και η Prolog συνεχίζει να προχωράει προς τα πίσω βρίσκοντας την αμέσως προηγούμενη κλήση που ήταν η write/1. Ούτε κι αυτή μπορεί να ικανοποιηθεί με δεύτερο τρόπο (ό,τι της είπαμε το έγραψε, τί άλλο να κάνει;) και η ανάποδη πορεία συνεχίζεται μέχρι την κλήση man/1. Αυτή προηγουμένως έδωσε το **X=peter** και μπορεί να γίνει TRUE και με άλλο τρόπο: **X=jimmy**. Η X λοιπόν αλλάζει τιμή και η πορεία της εκτέλεσης παίρνει ξανά την κανονική της φορά καλώντας την write(jimmy) και την nl/0 για να καταλήξει και πάλι στο fail. Και πάλι αντίστροφη πορεία μέχρι τη man/1 για να γίνει αυτή τη φορά **X=george** και να επαναληφθεί η διαδικασία. Οταν τελικά η Prolog φτάσει να έχει χρησιμοποιήσει όλες τις δυνατές τιμές χωρίς να έχει ικανοποιηθεί, τότε καταλαβαίνει ότι απέτυχε η goal2/0.

Το χαρακτηριστικό του "προς τα πίσω ελέγχου των κλήσεων" που περιγράψαμε εδώ, είναι αποκλειστική ιδιότητα της γλώσσας Prolog και ονομάζεται ***backtracking***.

Στην περιγραφή που παρακολουθήσαμε, είδαμε ότι υπάρχουν συναρτήσεις που μπορούν να ικανοποιηθούν με πολλούς τρόπους (όπως η man/1) και άλλες που ικανοποιούνται μόνο με έναν ή με κανέναν (πχ. write/1, nl/0, fail/0). Οι πρώτες λέγονται ***μη ντετερμινιστικές*** (***non-deterministic***) και οι δεύτερες ***ντετερμινιστικές*** (***deterministic***).

Για να ενεργοποιήσουμε το μηχανισμό του backtracking, πρέπει να προκαλέσουμε σε κάποια δεδομένη στιγμή μια αποτυχία. Η αποτυχία αυτή δεν είναι απαραίτητο να πραγματοποιείται με την κλήση της fail/0 αλλά θα μπορούσε να προκληθεί και με μια οποιαδήποτε άλλη συνάρτηση, όπως η woman/1 στην goal3/0. Το αποτέλεσμα είναι το ίδιο:

```
?- goal3.
peter
jimmy
george
No
```

Φυσικά, για τον ίδιο λόγο θα μπορούσε να προκληθεί backtracking και όταν εμείς δεν το θελήσουμε, πχ. σε κάποιον από τους περίπλοκους ορισμούς μας μια παράμετρος παρασύρει μια κλήση σε αποτυχία και τότε ενεργοποιείται το backtracking που μας επιστρέφει αρκετές κλήσεις πίσω, στην πρώτη μη-ντετερμινιστική σχέση που θα συναντήσει, δίνοντας καινούργια τιμή σε κάποια μεταβλητή

και εκτελώντας ξανά κάποιες λειτουργίες. Για να αποφύγουμε ανεπιθύμητα backtracking η Prolog διαθέτει ένα είδικό εργαλείο, το cut, τη λειτουργία του οποίου θα μελετήσουμε λεπτομερώς στην Ενότητα 13.

Τέλος, στην goal4/0 το fail γίνεται πολύ νωρίς και το backtracking περιορίζεται μόνο μεταξύ των συναρτήσεων man/1 και woman/1 χωρίς ποτέ να εκτελείται η write/1. Γιαυτό η απάντηση είναι απλώς:

```
?- goal4.
No
```

9. Matching (Ταίριασμα)

Το επόμενο πρόγραμμα που θα μας απασχολήσει αποτελείται μόνο από ένα fact:

```
now(date(wednesday,16,august,2006),time(23,15,20)).
```

Το παραπάνω γεγονός σημαίνει ότι το συγκεκριμένο compound object είναι TRUE. Ως γνωστόν, μπορούμε να κάνουμε διάφορες ερωτήσεις:

```
?- now(X,time(Y,15,20)).
X = date(wednesday,16,august,2006)
Y = 23
No
```

Η μεταβλητή Y πήρε την τιμή 23 ενώ η X πήρε όλη την παράσταση: date(wednesday,16,august,2006). Η Prolog απάντησε στην ερώτησή μας συμβουλευόμενη το πρόγραμμα που μόλις δώσαμε. Η διαδικασία με την οποία η Prolog διαλέγει τα κατάλληλα facts του προγράμματος για να απαντήσει στις ερωτήσεις μας, λέγεται ***matching***. Το matching (ταίριασμα, αντιστοίχιση) δεν είναι τίποτε άλλο παρά μια ένα-προς-ένα αντιστοίχιση δύο όρων. Στην προκειμένη περίπτωση, του όρου της ερώτησης και του γεγονότος. Για να αντιστοιχιστούν (να γίνουν matched) δύο όροι, πρέπει κατ' αρχήν να έχουν το ίδιο όνομα functor και το ίδιο arity. Το αυτό θα πρέπει να συμβαίνει και για κάθε ένα από τα ορίσματά τους. Στο παραπάνω παράδειγμα, ο functor στην ερώτηση και στο γεγονός είναι ο ίδιος (now) καθώς και το arity (2). Το πρώτο argument στην ερώτηση είναι μεταβλητή που άνετα μπορεί να πάρει την τιμή **date(wednesday,16,august,2006)** και να γίνει matched με το πρώτο argument του fact, ενώ τα δεύτερα arguments έχουν το ίδιο όνομα functor (time), ίδιο arity (3) και τα arguments τους αντιστοιχούν ένα-προς-ένα αν το Y πάρει την τιμή 23. Αλλο ένα παράδειγμα:

```
?- now(date(X,16,_,2006),_).
X = wednesday ;
No
```

Παρόμοια περίπτωση, μόνο που χρησιμοποιήσαμε ανώνυμες μεταβλητές οι οποίες γίνονται matched με οποιονδήποτε όρο. Προσοχή μόνο, να διατηρείται το arity:

```
?- now(date(X,16,_,2006),_,_).
No
```

Ενώ εδώ:

```
?- now(date(X,9,_,_),_).
No
```

...το 9 δε μπορεί να γίνει matched με το 16...

```
?- now(gate(X,16,_,_),_).
No
```

...ούτε το **gate** με το **date**.

Η διαδικασία του matching ενεργοποιείται όχι μόνο όταν πρόκειται να συγκρίνουμε ένα goal με τα facts του προγράμματός μας, αλλά και σε κάθε περίπτωση που μια μεταβλητή πρόκειται να πάρει τιμή ή δυο όροι συγκρίνονται με οποιονδήποτε τρόπο.

Και επειδή η διαδικασία του matching φαίνεται λίγο-πολύ προφανής, αξίζει να δούμε μερικά διδακτικά παραδείγματα που θα μπορούσαν να μας πείσουν για το αντίθετο:

Εστω το πρόγραμμα:

```
sum(5).
```

...και η ερώτηση:

```
?- sum(5).
Yes
```

...φυσικά, Yes...αλλά:

```
?- sum(2+3).
No
```

Δίδαγμα: οι πράξεις δεν γίνονται μόνες τους! Το 5 γίνεται matched με το 5 αλλά όχι με το 2+3.

Αντίστοιχα, αν το πρόγραμμά μας είναι το:

```
sum(2+3).
```

...τότε το:

```
?- sum(5).
No
```

...δικαίως αποτυγχάνει, ενώ η ερώτηση:

```
?- sum(X).
X = 2+3 ;
No
```

...μας δίνει την τιμή που θα έπρεπε να είχε ο όρος για να γίνει matched με το fact μας. Παραδόξως, το matching μπορεί να χρησιοποιηθεί για να δώσει απάντηση σε πιο "εξωτικές" ερωτήσεις, όπως:

```
?- sum(2+X).
X = 3 ;
No
```

...αλλά (επαναλαμβάνουμε) δεν κάνει πράξεις:

```
?- sum(3+X).
No
```

10. Είδη Ισότητας

Οι περισσότερες γλώσσες προγραμματισμού χρησιμοποιούν το σύμβολο = για να ελέγξουν την ισότητα δυο σχέσεων, για να προκαλέσουν την καταχώρηση τιμής σε μεταβλητή κλπ. Στην Prolog όπου η σαφήνεια και η λογική συνέπεια είναι στοιχεία αποφασιστικής σημασίας, πρέπει κάθε φορά να δηλώνουμε επακριβώς τι είδους "ισότητα" είναι αυτή που περιγράφουμε. Γιαυτό η Prolog διαθέτει 4 σύμβολα ισότητας (=, ==, **=:=**, **is**), η σημασία και η χρήση των οποίων περιγράφεται στα παραδείγματα που ακολουθούν:

```
?- 5=5.
Yes
```

Το σύμβολο = ελέγχει αν δυο όροι είναι ίδιοι (αν γίνονται matched).

```
?- hello = hello.
Yes
```

Μπορεί ακόμα να εκτελέσει καταχώρηση τιμής:

```
?- X = hello.
X = hello ;
No
```

Η καταχώρηση γίνεται με matching:

```
?- X=5.
X = 5 ;
No
```

...και, όπως είναι φυσικό, το matching δεν έχει φορά:

```
?- 5=X.
X = 5 ;
No
```

...αλλά έχει τη συνήθεια να εκτελεί **κατά γράμμα** τις δηλώσεις μας:

```
?- X=4+1.
X = 4+1
No
```

Το **is** κάνει καταχώρηση αφού υπολογίσει την τιμή του δεύτερου μέλους:

```
?- X is 4+1.
X = 5 ;
```

```
No
```

...αλλά μόνο του δεύτερου:

```
?- 4+1 is X.
No
```

Μπορεί να ελέγξει ισότητα αριθμών:

```
?- 5 is 5.
Yes
```

...αλλά μόνο αριθμών...

```
?- hello is hello.
No
```

...γιατί συνηθίζει να κάνει πράξεις στο δεύτερο μέλος.

Αντίθετα από ότι συμβαίνει σε άλλες γλώσσες προγραμματισμού, μια μεταβλητή στην Prolog δε μπορεί να αλλάξει τιμή με διαδοχικές καταχωρήσεις:

```
?- X is 5, X is 6.
No

?- X=5, X=6.
No
```

...επομένως εκείνο το **X=X+1** που χρησιμοποιούσαμε κατά κόρον σε διαδικαστικές γλώσσες θα πρέπει να το ξεχάσουμε, μια που θα γίνεται μονίμως FALSE.

Το =**:**= κάνει υπολογισμό και στα δυο μέλη:

```
?- 4+5 =:= 3*3.
Yes
```

...αλλά δεν κάνει καταχώρηση:

```
?- X =:= 4+5.
ERROR: =:=/2: Arguments are not sufficiently instantiated
```

...και συγκρίνει μόνο αριθμητικές παραστάσεις:

```
?- hello =:= hello.
No
```

Το σύμβολο == κάνει έλεγχο ισότητας:

```
?- 5 == 5.
Yes
```

...και σε μη αριθμητικά:

```
?- hello == hello.
Yes
```

...αλλά όχι υπολογισμό:

```
?- 5 == 4+1.
No
```

...ούτε καταχώρηση:

```
?- X == 5.
No
```

Το **X==5** θα μπορούσε να γίνει TRUE μόνο στην περίπτωση που το X είχε πάρει προηγουμένως τιμή, πχ:

```
?- X=5, X == 5.
X = 5 ;
No
```

Το == είναι πολύ ευαίσθητο στις μεταβλητές, ακόμα κι όταν αυτές δεν έχουν πάρει τιμή (***unbound variables***). Μια unbound μεταβλητή μπορεί να συγκριθεί με τον εαυτό της:

```
?- X == X.
X = _G224 ;
No
```

...αλλά όχι με άλλη:

```
?- X == Y.
No
```

Σημειώστε ότι το σύμβολο = ήταν πιο ελαστικό στη σύγκριση unbound μεταβλητών. Τις ενοποιούσε στον ίδιο εσωτερικό καταχωρητή:

```
?- X = Y.
X = _G206
Y = _G206 ;
No
```

Συνοψίζοντας, τα σύμβολα ισότητας στην Prolog εκτελούν τις λειτουργίες που δείχνει ο πίνακας:

=	Κάνει σύγκριση και καταχώρηση με matching. Χειρίζεται τα μέλη ως μη αριθμητικά, γιαυτό και δεν κάνει πράξεις.
is	Μόνο για αριθμητικές παραστάσεις. Υπολογίζει το δεξί μέλος και κάνει καταχώρηση ή σύγκριση.
=:=	Μόνο για αριθμητικές παραστάσεις. Υπολογίζει και τα δυο μέλη. Κάνει σύκγριση αλλά όχι καταχώρηση.
==	Χειρίζεται τα μέλη ως μη αριθμητικά (δεν εκτελεί υπολογισμούς). Δεν κάνει καταχώρηση, μόνο έλεγχο.

11. Auto-Executable Goals και Σχόλια

Κάθε εντολή που μπορούμε να γράψουμε στην Prolog μπορεί να εκφραστεί στην παρακάτω γενικευμένη μορφή:

```
Head :- Body.
```

όπου το Head είναι μία συνάρτηση και το Body μια ακολουθία συναρτήσεων συνδεδεμένες με τελεστές. Μια εντολή της Prolog με την παραπάνω μορφή λέγεται **κανόνας** όπως έχουμε ήδη πεί. Ενα γεγονός είναι μια ειδική περίπτωση κανόνα, αφού μπορεί να γραφτεί ως:

```
Head :- true.
```

Συνήθως όμως παραλείπουμε τελείως το **true** και γράφουμε:

```
Head.
```

Οπως λοιπόν από τη γενικευμένη έκφραση **Head:-Body.** μπορεί να λείπει το Body, με την ίδια ευχέρεια μπορεί να λείπει το Head. Σε αυτή την περίπτωση η εντολή παίρνει τη μορφή:

```
:- Body.
```

καί έχει την έννοια της ***άμεσα εκτελέσιμης εντολής***: οι συναρτήσεις που βρίσκονται στο Body εκτελούνται αυτόματα κατά τη διαδικασία της διαδικασίας **consult**. Παράδειγμα:

```
:- write('NOW LOADING PROGRAM'), nl.
          :
          :
          :     (εντολές προγράμματος)
          :
          :
:- write('Ready to answer your questions'), nl.
```

Ο κώδικας του προγράμματος παρεμβάλλεται μεταξύ δυο άμεσα εκτελέσιμων εντολών (auto-executable goals). Οταν η Prolog διαβάζει το file, βλέπει το πρώτο goal και γράφει στην οθόνη το μήνυμα:

```
NOW LOADING PROGRAM
```

Στη συνέχεια, διαβάζει μια-μια τις εντολές του προγράμματος και τις καταχωρεί στη μνήμη (στον Prolog Workspace, όπως συνηθίζεται να τον λέμε) και τέλος βρίσκει το δεύτερο auto-executable goal που γράφει στην οθόνη:

```
Ready to answer your questions
```

Μετά εμφανίζεται (σα να μη συμβαίνει τίποτα) το γνωστό prompt:

```
?-
```

Φυσικά, η χρήση των auto-executable goals δεν δεν περιορίζεται στην απεικόνιση μηνυμάτων. Οποιαδήποτε built-it εντολή της Prolog ή συνάρτηση που έχουμε ορίσει μπορεί να κληθεί με αυτό τον τρόπο. Τυπική περίπτωση είναι η παρουσία εντολών:

```
:- consult(filename).
```

μέσα σε ένα file, που προκαλεί το άμεσο φόρτωμα άλλων Prolog files (πχ. βιβλιοθήκες).

Όπως σε όλες τις γλώσσες προγραμματισμού, έτσι και στην Prolog μπορούμε να ενσωματώσουμε σχόλια στον κώδικα. Ο συμβολισμός των σχολίων μοιάζει και με άλλες γλώσσες προγραμματισμού και κάνει χρήση των συμβόλων /*, */ και %:

```
/*
  Ο,τι περικλείεται μεταξύ των συμβόλων /* και */
  είναι σχόλιο, όσες γραμμές κι αν καταλαμβάνει.
*/

%  αυτό είναι επίσης σχόλιο, μέχρι το end-of-line
```

12. Recursion (Αναδρομικότητα)

Βασικό χαρακτηριστικό αρκετών γλωσσών προγραμματισμού είναι η δυνατότητα που παρέχουν στις συναρτήσεις (ή υπορουτίνες) τους να καλούν τον εαυτό τους. Η τεχνική αυτή λέγεται ***recursion*** (***αναδρομικότητα***) και αποτελεί ένα από τα βασικά χαρακτηριστικά της Prolog.

Ας δούμε το εξής πρόγραμμα:

```
parent(alex,bill).
parent(bill,charlie).
parent(charlie,don).
     :
     :
     :      % (n γεγονότα)
     :
     :
parent(simon,timothy).
```

Εχουμε μια σειρά από γεγονότα που εκφράζουν σχέσεις γονέα-παιδιού. Αυτό που θέλουμε είναι να φτάξουμε μια συνάρτηση που να ορίζει τη σχέση προγόνου-απογόνου. Εστω λοιπόν η συνάρτηση ancestor(X,Y) με την έννοια "*ο Χ είναι πρόγονος του Υ*". Θα μπορούσε να ορίζεται κάπως έτσι:

```
ancestor(X,Y):- parent(X,Y);
          ( parent(X,Z), parent(Z,Y) );
          ( parent(X,Z), parent(Z,W), parent(W,Y) );
               :
               :
               :     % (n όροι)
```

"*Ο Χ λοιπόν είναι πρόγονος του Υ αν είναι ή πατέρας του, ή παππούς του, ή προπάππος του κτλ.*" Είναι φανερό ότι για να εκφράσουμε με αυτόν τον τρόπο ολόκληρη την αλυσίδα, θα χρειαζόμασταν n όρους. Εναλλακτικά, θα μπορούσαμε να είχαμε γράψει:

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- parent(X,Z), parent(Z,Y).
ancestor(X,Y):- parent(X,Z), parent(Z,W), parent(W,Y).
               :
               :
               :     % (n κανόνες)
```

...αλλά πάλι, θα χρειαζόμασταν n κανόνες.

Η πιο κομψή λύση είναι η αναδρομική:

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

...με μόνο δυο κανόνες: "*ο X είναι πρόγονος του Y αν είναι πατέρας του ή αν είναι πατέρας κάποιου Z ο οποίος είναι πρόγονος του Y*". Βλέπουμε έτσι και στην πράξη την έννοια της αναδρομικότητας τώρα που η ancestor/2 καλεί τον εαυτό της. Αλλά τη στιγμή που αυτοκαλείται η ancestor/2, δεν έχει ολοκληρωθεί ο ορισμός της! Πώς γίνεται λοιπόν να χρησιμοποιήσουμε μια έννοια χωρίς να την έχουμε ορίσει πλήρως; Η απάντηση είναι ότι ο "μερικός" ορισμός της ancestor/2 που έχει πραγματοποιηθεί ως εκείνη τη στιγμή, είναι αρκετός για να δώσει μια πλήρη απάντηση. Συγκεκριμένα, ο "μισός" δεύτερος κανόνας χρησιμοποιείται για τις αναδρομές και ο πλήρης πρώτος κανόνας καθορίζει πότε θα σταματήσει η αναδρομική διαδικασία. Το συνεργαζόμενο δίδυμο αποτελεί έναν ολοκληρωμένο ορισμό της συνάρτησης ancestor/2. Βλέπουμε λοιπόν ότι οι αναδρομικοί ορισμοί είναι "δυναμικοί": πρέπει να εκτελεστούν για να ολοκληρωθούν.

Για ευνόητους λόγους, ο πρώτος από τους δυο κανόνες ενός αναδρομικού ορισμού λέγεται ***τερματική σχέση*** ή ***τερματικός κανόνας***, ενώ ο δεύτερος ονομάζεται ***αναδρομική σχέση*** ή ***αναδρομικός κανόνας***. Συνήθως σε έναν αναδρομικό ορισμό, η τερματική σχέση προηγείται της αναδρομικής (όπως και στην περίπτωση της ancestor/2) έτσι ώστε να γίνεται πρώτα ο έλεγχος του τερματισμού της διαδικασίας πρίν εκτελεστεί νέα αναδρομή.

Προσοχή, η αναδρομική σχέση:

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

δεν είναι το ίδιο να γραφτεί:

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

ούτε:

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

γιατί ναι μεν η συνάρτηση θα έδινε σωστές απαντήσεις για ονόματα που υπάρχουν στα γεγονότα, αλλά αν ρωτούσαμε:

```
?- ancestor(stranger,bill).
```

...η απάντηση δεν θα ήταν **no**, αλλά τίποτα, μια και η Prolog θα έμπαινε σε ατέρμονο loop καλώντας συνέχεια την ancestor/2 για να ανακαλύψει τη σχέση του stranger με κάποιον από τους υπόλοιπους[1]. Για άλλη μια φορά διαπιστώνουμε ότι η λογική της Prolog διαφέρει από τη μαθηματική Λογική. Για να δουλεύουν σωστά οι αναδρομικοί ορισμοί που γράφουμε, το μυστικό είναι να **βάζουμε πρώτα τις ειδικότερες κλήσεις και μετά τις γενικότερες, ώστε να οδηγούμε την Prolog γρήγορα στις τερματικές σχέσεις**.

Στο παράδειγμά μας, η συνάρτηση parent/2 είναι πιο ειδική από την ancestor/2 γιαυτό και καλείται πρώτα. Ακόμα και στους κανόνες, βάζουμε πρώτα την τερματική σχέση σαν πιο ειδική και μετά την αναδρομική σαν πιο γενική.

Και πώς συγκρίνουμε δυο συναρτήσεις ως προς τη ειδικότητά τους; Γενικά, **μια ντετερμινιστική συνάρτηση είναι πιο ειδική από μια μη ντετερμινιστική και μια μη αναδρομική είναι πιο ειδική**

1 Στη συνέχεια αυτής της ενότητας παρουσιάζονται λεπτομερέστερα παρόμοιες περιπτώσεις "λογικού κολλήματος".

από μια αναδρομική[1].

Ας δούμε τώρα ένα παράδειγμα μιας κλασσικής αναδρομικής συνάρτησης:

```
% Υπολογισμός Παραγοντικού
paragontiko(1,1).
paragontiko(X,Y):-  X2 is X-1,
                    paragontiko(X2,Y2),
                    Y is Y2*X.
```

"*Το παραγοντικό του 1 είναι 1 και το παραγοντικό ενός οποιουδήποτε άλλου αριθμού X είναι το παραγοντικό του προηγούμενού του πολλαπλασιασμένο επί X.*"

Προσέξτε την ομοιότητα του παραπάνω ορισμού με τη διαδικασία της ***τέλειας επαγωγής***: στην επαγωγή ορίζει κανείς μια σχέση για κάποιον συγκεκριμένο ακέραιο (τερματική σχέση) και μετά θεωρώντας ότι η σχέση ισχύει για κάποιον ακέραιο N επεκτείνει τον ορισμό για N+1. Πολλές φορές θα διακρίνουμε μια "κρυμμένη επαγωγή" στις αναδρομικές συναρτήσεις.

Καιρός να δοκιμάσουμε τον ορισμό μας:

```
?- paragontiko(4,X).
X = 24
```

Μια συνήθεια που πρέπει να αποκτά από νωρίς ένας προγραμματιστής Prolog είναι να ελέγχει την ορθότητα των ορισμών του βασιζόμενος στη λογική και όχι στην αλγοριθμική διαδικασία όπως θα επιχειρήσουμε να κάνουμε εδώ. Αυτή η αντιμετώπιση χρησιμεύει μόνο κατά τη διαδικασία του step-by-step debugging[2] (βήμα-προς-βήμα αποσφαλμάτωση) και μάλιστα τότε γίνεται αυτόματα από την Prolog. Ακολουθεί η "δαιδαλώδης" αλγοριθμική εξήγηση της απάντησης στην ερώτησή μας "ποιό είναι το παραγοντικό του 4;"

1 Η Prolog λοιπόν προσπαθεί να οδηγήσει το goal **paragontiko(4,X)** σε TRUE. Γνωρίζει μόνο δυο σχέσεις που αναφέρονται σε αυτή τη συνάρτηση. Η πρώτη που βρίσκει είναι η τερματική σχέση αλλά την απορρίπτει αμέσως γιατί δε μπορεί να κάνει match το 4 της ερώτησης με το 1 που είναι το πρώτο argument του γεγονότος.

2 Συναντά στη συνέχεια την αναδρομική σχέση και βλέπει ότι της ταιριάζει αφού μπορεί να αντιστοιχίσει το 4 της ερώτησης με το X του κανόνα και το X της ερώτησης με το Y του κανόνα. Σημειώστε ότι τα δυο X δεν έχουν καμμία σχέση! Οι μεταβλητές στην Prolog είναι local μόνο μέσα στον κανόνα που αναφέρονται. Μετά την τελεία, οποιαδήποτε αναφορά στην ίδια μεταβλητή είναι απλή συνωνυμία (πρόκειται για άλλη μεταβλητή).

3 Τώρα η Prolog πρέπει να ελέγξει αν ο κανόνας που διάλεξε καταλήγει σε TRUE η FALSE, ώστε να δώσει την κατάλληλη λογική τιμή στην αρχική ερώτηση. Υπολογίζει τον

1 Αν θέλουμε να είμαστε πιο ακριβείς, μπορούμε να χρησιμοποιήσουμε τον ορισμό της πολυπλοκότητας κατά Kowalski: "*όσο πιο πολλές λύσεις δίνει μιά συνάρτηση, τόσο πιο πολύπλοκη (γενική) είναι*". Ο κανόνας αυτός χρησιμοποιείται και για το optimization ενός προγράμματος Prolog. Οταν καλούμε πρώτα τις λιγώτερο πολύπλοκες συναρτήσεις, το πρόγραμμά μας γίνεται πιο αποτελεσματικό.

2 Βλ. Ενότητα 16.

προηγούμενο ακέραιο (3) και κάνει μια **πρώτη** αναδρομική κλήση στην **paragontiko(3,Y2)**. Η **Y is Y2*X** δεν εκτελείται ακόμα, θα εκτελεστεί μόλις η **πρώτη** αναδρομική κλήση καταλήξει σε TRUE.

4 Η καινούρια κλήση επανεξετάζει όλους τους κανόνες. Η τερματική σχέση πάλι απορρίπτεται αφού το 3 δεν ταιριάζει με το 1 και επιλέγεται πάλι η αναδρομική σχέση αυτή τη φορά όμως για να δώσει απάντηση όχι στην αρχική ερώτηση αλλά στην **πρώτη** αναδρομική κλήση.

5 Πάλι υπολογίζεται ο προηγούμενος ακέραιος (2) και εκτελείται μια **δεύτερη** αναδρομική κλήση στην **paragontiko(2,Y2)**, χωρίς ακόμα να έχει εκτιμηθεί η λογική τιμή της **πρώτης** αναδρομικής σχέσης.

6 Και ξανά η ίδια διαδικασία, επιλογή της αναδρομικής σχέσης, υπολογισμός του αμέσως προηγούμενου ακεραίου (1) και ακολουθεί **τρίτη** αναδρομική κλήση, **paragontiko(1,Y2)**.

7 Αυτή τη φορά το goal **paragontiko(1,Y2)** ταιριάζει με την τερματική σχέση, το Y2 παίρνει την τιμή 1 και η **τρίτη** αναδρομική σχέση γίνεται TRUE. Είναι η πρώτη φορά που μια κλήση της συνάρτησης paragontiko/2 γίνεται TRUE.

8 Η **τρίτη** αναδρομική κλήση κλήθηκε από τη **δεύτερη** αναδρομική κλήση **paragontiko(2,Y2)** και τώρα που η **τρίτη** επαληθεύτηκε (επιστρέφοντας την τιμή 1), συνεχίζεται η εκτέλεση της **δεύτερης**. Εκτελείται λοιπόν η **Y is Y2*X** (με X=2, Y2=1) και το αποτέλεσμα (2) είναι η τιμή για το 2! που επιστρέφει η **δεύτερη** αναδρομική κλήση.

9 Αφού λοιπόν και η **δεύτερη** αναδρομική κλήση έγινε TRUE, εκτελείται η υπολειπόμενη εντολή της **πρώτης** αναδρομικής κλήσης: **Y is Y2*X** με X=3 και Y2=2. Το αποτέλεσμα (6) είναι το 3! και η **πρώτη** αναδρομική κλήση έχει γίνει TRUE.

10 Μένει τέλος να εκτελεστεί η τελευταία κλήση του αρχικού μας goal: **Y is Y2*X**, αλλά αυτή τη φορά με X=4 και Y2=6. Το Y παίρνει την τιμή 24 (η τελική απάντηση) και το goal επιτυγχάνει.

Δυστυχώς δε μπορούμε να κάνουμε αντίστροφες ερωτήσεις:

```
?- paragontiko(X,24).
ERROR: is/2: Arguments are not sufficiently instantiated
```

...γιατί οι συναρτήσεις is/2 που χρησιμοποιήσαμε στον ορισμό δεν είναι συμμετρικές[1].

Θα προσέξατε ίσως ότι η απάντηση της Prolog στην προηγούμενη ερώτηση:

```
?- paragontiko(4,X).
X = 24
```

...δεν περιέχει **No** μετά το **X = 24**. Αυτό συμβαίνει επειδή η Prolog "κολλάει" στην προσπάθειά της

1 **Συμμετρικές** λέγονται οι συναρτήσεις που δεν έχουν καθορισμένες εισόδους και εξόδους. Βλ. Ενότητα 14 για πλήρη ορισμό των συμμετρικών συναρτήσεων.

να βρεί επιπλέον απαντήσεις στην ερώτησή μας. Το "κόλημμα" ξεκινάει από το βήμα 7 που είδαμε προηγουμένως. Η τερματική σχέση ταίριαζε και η Prolog την διάλεξε μια που εμφανίστηκε πρώτη. Αλλά και η αναδρομική σχέση θα την ικανοποιούσε, μια και το 1 μπορεί να γίνει matched με το X και το Y2 με το Y. Επειδή όμως επέλεξε την πρώτη, σύμφωνα με όσα είπαμε στο backtracking, η Prolog θα σημειώσει τον εναλλακτικό κανόνα και θα τον χρησιμοποιήσει αν της ζητήσουμε επιπλέον λύσεις. Ο κανόνας όμως αυτός την ρίχνει σε loop, αφού την αναγκάζει να υπολογίσει το παραγοντικό του 0 και μετά του -1, του -2, του -3,... κοκ, χωρίς ποτέ να ξαναφτάνει στην τερματική σχέση, αφού οι εξεταζόμενοι ακέραιοι δεν ταιριάζουν με το 1.

Ίδιου είδους κόλλημα συμβαίνει και στην παρακάτω περίπτωση:

```
%  Ένα ατέρμονο loop στην Prolog
loop:- loop.
```

Το πρόγραμμα αποτελείται μόνο από έναν κανόνα ο οποίος καλεί απλώς τον εαυτό του. Δίνοντας:

```
?- loop.
```

...η Prolog πέφτει σε ατέρμονο "λογικό" loop και με το δίκιο της γιατί προσπαθεί να ελέγξει αν η συνάτρηση loop είναι TRUE. Βρίσκει τον κανόνα που έχει το loop στο head του και εκτελεί το body. Ο,τι λογική τιμή πάρει το body, την ίδια θα έχει και η συνάρτηση loop. Το body όμως λέει loop, της οποίας η λογική τιμή είναι ακόμη απροσδιόριστη. Ξανακαλείται λοιπόν ο κανόνας που έχει το loop στο head του και εκτελείται το body του, που είναι πάλι loop! Ο φαύλος κύκλος είναι φανερός σε μας αλλά η Prolog είναι προγραμματισμένη να ακολουθεί το νήμα ώσπου να καταλήξει σε κάποια λογική τιμή. Οσο αυτή δε βρίσκεται, η γλώσσα θα συνεχίζει να ψάχνει...

Ας δούμε ένα χρήσιμο μικρό πρόγραμμα:

```
%  Ενας μερτητής
count(X):- write(X), nl,
           NewX is X+1,
           count(NewX).
```

Οπως φαίνεται από τον ορισμό, αν καλέσουμε την count/1 με κάποιον ακέραιο για argument, τότε ο ακέραιος θα τυπωθεί, θα γίνει αλλαγή γραμμής, θα υπολογιστεί ο επόμενος ακέραιος και θα ξανακληθεί η count/1 με τον καινούριο ακέραιο. Παράδειγμα:

```
?- count(15).
15
16
17
18
19
20
:
```

```
:
```

Ο μετρητής συνεχίζει χωρίς να σταματήσει πουθενά, αφού δεν υπάρχει τερματική σχέση. Για να τερματίσει ένα goal, πρέπει να καταλήξει κάποτε σε κάποιο γεγονός. Ας βάλουμε λοιπόν στον μετρητή μας ένα γεγονός (τερματική σχέση) που θα τον κάνει να σταματήσει:

```
count(18).
count(X):-  write(X), nl,
          NewX is X+1,
          count(NewX).

?- count(14).
14
15
16
17
Yes
```

Τώρα σταματάει στο 17, εκτός αν αρχίσουμε το μέτρημα από ψηλότερα:

```
?- count(19).
19
20
21
22
:
:
```

...οπότε δεν τη σταματάει η τερματική σχέση.

Ο μετρητής μας μπορεί να μετράει (θεωρητικά) επ'άπειρον χωρίς ποτέ να μείνει από μνήμη. Οι αναδρομικές κλήσεις που έχουν αυτή την ιδιότητα λέγονται ***tail recursive***. Αν φτιάχναμε τον μετρητή κάπως πιο άτσαλα, πιθανόν να μην είχε αυτή την ιδιότητα:

```
%  Ενας "κακός" μερτητής
badcount1(X):- write(X), nl,
           NewX is X+1,
           badcount1(NewX),
           nl.
```

Η μόνη διαφορά από τον προηγούμενο είναι ότι στον ορισμό υπάρχει και ένα **nl** μετά την αναδρομική κλήση. Κοιτάξτε όμως αλλαγή:

```
?- badcount1(1).
1
2
3
:
:
```

```
1514
:
:
50778
ERROR: (user://1:47):
        Out of local stack
```

Γιατί "Out of local stack"; Πού καταναλώθηκε τόση μνήμη; Μα στο nl/0 φυσικά! Η συνάρτηση καλεί τον εαυτό της πριν εκτελεστεί η nl/0. Αλλά η nl/0 ανήκει στο body του κανόνα. Επομένως θα πρέπει κάποτε να εκτελεστεί. Για το σκοπό αυτό η Prolog βάζει έναν pointer να δείχνει στην nl/0 έτσι ώστε να την εκτελέσει όταν επιστρέψει από την αναδρομική κλήση. Ομως ο pointer αυτός δεν είναι μόνο ένας! Είναι ένας για κάθε κύκλο της αναδρομικής κλήσης. Και αφού η Prolog δεν επιστρέφει ποτέ από αυτές τις κλήσεις, όλα εκείνα τα bytes παραμένουν και γεμίζουν το χώρο του stack[1].

Βασική λοιπόν προϋπόθεση για να έχουμε tail recursion είναι **στην αναδρομική σχέση, μετά από την αναδρομική κλήση να υπάρχει τελεία**. Ο,τιδήποτε άλλο υπάρχει εκτός του ότι δεν πρόκειται να εκτελεστεί, θα καταναλώνει και μνήμη...

Βέβαια, αυτό δεν είναι πάντα δυνατό. Μερικές συναρτήσεις δε μπορούν να γίνουν tail recursive. Στο παράδειγμα με το παραγοντικό η αναδρομική κλήση δε θα μπορούσε να μπεί τελευταία. Γιαυτό, για μεγάλο πρώτο argument η paragontiko/2 θα βγάλει "Out of local stack"[2].

```
%  Αλλος ένας "κακός" μερτητής
badcount2(X):- write(X), nl,
           NewX is X+1,
           badcount2(NewX).
badcount2(X):- X<0,
           write('X is negative'),nl.

?- badcount2(1).
1
2
3
:
```

1 Η Prolog χρησιμοποιεί ένα χώρο της μνήμης ως stack (σωρό) στον οποίο αποθηκεύει προσωρινές τιμές διευθύνσεων για το backtracking.

2 Ειδικά όμως για την paragontiko/2 υπάρχει ένα διδακτικό "κολπάκι" που την κάνει tail recursive. Δείτε:

```
parag(X,1):- 1 =:= X.
parag(X,Y):- X2 = X-1, Y = Y2*X, parag(X2,Y2).

paragontiko(X,Y):- parag(X,Y1), Y is Y1.

?- paragontiko(4,X).
X = 24

?- parag(4,X).
X = 1 * (4 - 1 - 1) * (4 - 1) * 4
```

```
:
1704
:
:
29870
ERROR: (user://2:50905):
        Out of local stack
```

Ακόμα κι αν η αναδρομική κλήση είναι η τελευταία του κανόνα, **δεν πρέπει να υπάρχει εναλλακτική σχέση που δεν δοκιμάζεται ποτέ**. Εδώ υπήρχε κάποιος έλεγχος για αρνητικές τιμές. Σε κάθε αναδρομική κλήση όμως επιλεγόταν πάντα ο πρώτος κανόνας. Και σε κάθε αναδρομή η Prolog έβαζε κάποιον pointer να δείχνει ότι υπάρχει και εναλλακτικός κανόνας, για την περίπτωση που ο πρώτος κατέληγε σε fail. Οι pointers που μαζεύτηκαν μετά από αρκετούς κύκλους έκαναν τον stack να γεμίσει.

```
%  Ενας εξίσου "κακός" μερτητής
badcount3(X):- write(X), nl,
           NewX is X+1,
           check(NewX),
           badcount3(NewX).
check(Z):- Z>0.
check(Z):- Z<0, write('X is negative'),nl.

?- badcount3(1).
1
2
3
:
:
:
19530
ERROR: (user://3:80820):
        Out of local stack
```

Δεν πρέπει ακόμα η αναδρομική σχέση να περιέχει μια κλήση η οποία έχει μια εναλλακτική που δεν δοκιμάζεται ποτέ. Εδώ η check/1 ικανοποιείται πάντα από τον πρώτο κανόνα και ο δεύτερος δεν επιλέγεται ποτέ. Οι pointers όμως μπαίνουν...

Γενικά είναι ωφέλιμο να προσπαθούμε να κάνουμε τις αναδρομικές μας συναρτήσεις tail recursive, όπου είναι δυνατόν. Όχι γιατί θέλουμε να "τρέχουν επ' άπειρον", αλλά γιατί γίνονται πιο αποτελεσματικές, καταναλώνοντας λιγώτερη μνήμη.

13. Cut (!)

Το cut είναι μια ειδική συνάρτηση-εντολή της Prolog που διακόπτει το backtracking. Συμβολίζεται με θαυμαστικό "**!**" και η λειτουργία του εξηγείται κατ'αρχήν με ένα παράδειγμα:

```
man(peter).    man(jimmy).    man(george).
a:- man(X), write(X), nl, fail.
b:- man(X), !, write(X), nl, fail.
c:- man(X), write(X), nl, !, fail.
d:- !, man(X), write(X), nl,  fail.
e:- man(X), write(X), nl, fail, !.
```

```
?- a.
peter ;
jimmy ;
george ;
No
```

Το γνωστό πλέον κόλπο του backtracking με το τελικό fail, χρησιμοποιήθηκε εδώ για να μας δώσει όλες τις τιμές που ικανοποιούν τη συνάρτηση man/1.

```
?- b.
peter ;
No
```

Η b/0 διαφέρει από την a/0 στο cut που ακολουθεί την man/1. Το ! λειτούργησε ως εξής: Η man(X) επιστρέφει την πρώτη τιμή που βρίσκει (X=peter). Μετά ακολουθεί το cut, (το οποίο γίνεται πάντα TRUE) και η write(X) γράφει τη λέξη **peter**. Εκτελείται η nl/0 και μετά εμφανίζεται το fail. Η Prolog τώρα μπαίνει στη διαδικασία του backtracking, ψάχνοντας για μια μη-ντετερμινιστική συνάρτηση που θα ικανοποιήσει ενδεχομένως ολόκληρη την παράσταση. Η nl/0 δε μας κάνει, ούτε η write/1 και το backtracking φτάνει αισίως στο cut. Αν δεν υπήρχε το cut θα έβρισκε τη μη-ντετερμινιστική man/1, τώρα όμως **το cut απαγορεύει περαιτέρω backtracking** και το goal b/0 αποτυγχάνει. Το cut λοιπόν λειτουργεί σαν δίοδος μονής κατεύθυνσης. Οταν η ροή είναι φυσιολογική (ορθή φορά εκτέλεσης = η κλήση των συναρτήσεων γίνεται από αριστερά προς τα δεξιά) το cut γίνεται TRUE και δεν εμποδίζει καθόλου (είναι σα να μην υπάρχει). Στο backtracking, όμως, το cut απαγορεύει τη "διέλευση" προς τα αριστερά του.

Ετσι, στην a/0 το backtracking γίνεται από το fail μέχρι το man(X), στην b/0 από το fail μέχρι το ! και στην c/0 πάλι από το fail ως το ! που τώρα απέχουν μεν ελάχιστα, αλλά το αποτέλεσμα είναι το ίδιο:

```
?- c.
peter ;
No
```

Στην d/0 το backtracking συναντάει τη μη-ντετερμινιστική συνάρτηση man/1 πριν φτάσει στο !,

γιαυτό και η συμπεριφορά της μοιάζει με της a/0:

```
?- d.
peter ;
jimmy ;
george ;
No
```

Τέλος, στην e/0, το ! βρίσκεται μετά το fail, οπότε είναι σα να μην υπάρχει[1].

```
?- e.
peter ;
jimmy ;
george ;
No
```

Και αφού πήραμε μια πρώτη γεύση για το πώς δουλεύει το !, ας δώσουμε επακριβώς τον ορισμό της λειτουργίας του:

Το cut (!) στο body ενός κανόνα, προκαλεί τα εξής φαινόμενα:

(α) Απαγορεύει το backtracking προς τα αριστερά του, με όλες τις επακόλουθες συνέπειες: οι τιμές που έχουν έχουν οι μεταβλητές τη στιγμή που εκτελείται (κατά την ορθή φορά) το ! δεν πρόκειται να αλλάξουν αφού αποκλείεται το backtracking. Η λογική τιμή του κανόνα εξαρτάται από τη λογική τιμή του υπόλοιπου κανόνα και μόνον, αφού μέχρι το ! ο κανόνας είναι TRUE.

(β) Αποτρέπει την Prolog να εξετάσει εναλλακτικό κανόνα με τον ίδιο functor στο head. Από τη στιγμή που στον επιλεγμένο κανόνα έχει συναντηθεί cut, οποιοιδήποτε pointers δημιουργήθηκαν τη στιγμή της επιλογής (pointers που έδειχναν σε εναλλακτικούς κανόνες) σβήνονται και ο κανόνας λειτουργεί ως ντετερμινιστικός.

Προηγουμένως εξετάσαμε το (α). Στο παράδειγμα που ακολουθεί παρουσιάζεται το (β):

Ορίζεται η συνάρτηση not1/1 που γίνεται TRUE όταν το όρισμά της είναι FALSE, και FALSE όταν το όρισμα είναι TRUE[2].

```
not1(X):- X, !, fail.
not1(_).
```

Προσέξτε λογική: "*Για να εκτελέσεις το not1(X), εκτέλεσε πρώτα το X*". Αν αυτό είναι FALSE ο πρώτος κανόνας αποτυγχάνει αμέσως και η Prolog επιλέγει τον δεύτερο, ο οποίος γίνεται TRUE πάντα. Αν όμως το X είναι TRUE, τότε στον πρώτο κανόνα εκτελείται το ! που σημαίνει: "*αγνόησε οποιουσδήποτε εναλλακτικούς κανόνες, η not1(X) είναι πια ντετερμινιστική*" και fail. Ακολουθεί ένα

1 Γενικά, η fail/0 χρησιμοποιείται μόνο ως τελευταία κλήση σε ένα body. Δεν έχει νόημα να γράψει κανείς κάτι μετά από το fail αφού ποτέ δε φτάνει η εκτέλεση του προγράμματος ως εκεί.

2 Στην πραγματικότητα η not1/1 έχει την ίδια λειτουργία με τον τελεστή \+, καθώς και με την built-in συνάρτηση not/1.

πολύ κοντινό backtracking μέχρι το ! και η not1(X) επιστρέφει τιμή FALSE.

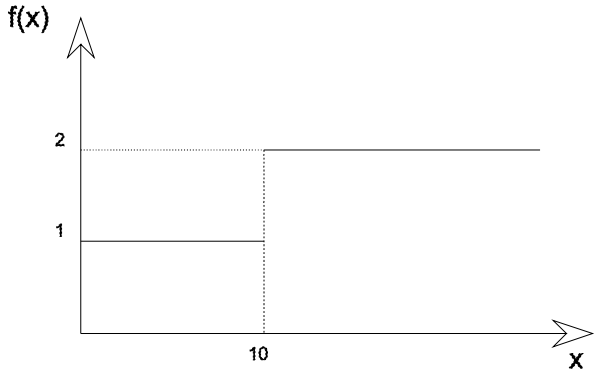
```
?- not1(1=2).
Yes

?- not1(2=2).
No
```

Αν ο παραπάνω ορισμός δεν περιείχε το !, το τελευταίο αποτέλεσμα θα ήταν:

```
?- not1(2=2).
Yes
```

Η "ντετερμινιστικοποίηση" ενός ορισμού με τη βοήθεια του ! είναι πολύ κοινή και έχει χρήση παντού: από το tail recursion μέχρι τον ορισμό μονότιμων μαθηματικών συναρτήσεων. Για παράδειγμα, έστω η συνάρτηση:

$$f(x) = \begin{cases} 1 \Leftarrow x < 10 \\ 2 \Leftarrow x \geq 10 \end{cases}$$



Για να την ορίσουμε στην Prolog χρειαζόμαστε κατ'αρχήν 2 arguments: ένα για είσοδο τιμής και ένα για έξοδο. Αυτό συμβαίνει γιατί στην Prolog (αντίθετα απ'ότι συμβαίνει σε άλλες γλώσσες προγραμματισμού) η ίδια η συνάρτηση δεν επιστρέφει κάποια τιμή (πχ. f(15)=2), παρά μονάχα λογικό TRUE ή FALSE. Ακολουθεί ο ορισμός:

```
f(X,Y):- X<10, !, Y=1.
f(_,2).
```

"*Αν το X είναι <10, τότε αγνόησε τον άλλο κανόνα. Το Y είναι 1. Για οποιαδήποτε άλλη τιμή του X, το Y είναι 2*".

```
?- f(4,X).
X = 1 ;
No

?- f(14,X).
X = 2 ;
No
```

Αν το ! έλειπε τότε η Prolog δε θα αγνοούσε το δεύτερο κανόνα και η συνάρτηση για X<10 θα είχε δυο τιμές:

```
?- f(4,X).
X = 1 ;
X = 2 ;
No
```

Αξιοσημείωτη είναι και η παρακάτω περίπτωση:

```
a:- (b;c), !, fail.
b:- write(b),nl.
c:- write(c),nl.

?- a.
b ;
No
```

Το cut εμποδίζει το backtracking και έτσι το **c** στο **(b;c)** δεν εκτελείται αν το **b** γίνει true.

Σε περιπτώσεις όπως οι προηγούμενες όπου το cut είναι αποφασιστικής σημασίας για τη λογική του οριζόμενου κανόνα, λέμε ότι **το cut είναι *λογικό*** ή **το cut είναι *red***. Αν το cut χρησιμεύει μόνο για να κάνει το πρόγραμμα πιο efficient και η απουσία του δεν θα είχε λογική αλλά λειτουργική διαφορά (πχ. μια καθυστέρηση στην εκτέλεση του προγράμματος), λέμε ότι **το cut είναι *διαδικαστικό*** ή ***λειτουργικό*** ή ότι **το cut είναι *green***. Green cuts χρησιμοποιούνται συνήθως όταν ξέρουμε εκ των προτέρων ότι ένας κανόνας είναι ντετερμινιστικός. Τότε μπορούμε να βάλουμε ! αμέσως πριν την τελεία, για να αποφύγουμε έτσι το χρονοβόρο backtracking. Πολύ περισσότερο, αν το backtracking μας οδηγεί σε ατέρμονο loop, όπως στο παράδειγμα με τις εναλλακτικές λύσεις του παραγοντικού (βλ. Ενότητα 12). Εκεί, αν διορθώναμε την τερματική σχέση και αντί:

```
paragontiko(1,1).
```

γράφαμε:

```
paragontiko(1,1):- !.
```

τότε η επιλογή του τερματικού κανόνα θα απέκλειε την εναλλακτική δυνατότητα της αναδρομικής σχέσης και θα εμπόδιζε το backtracking.

14. Λίστες

Η ***λίστα*** (***list***) είναι μια δομή δεδομένων εξαιρετικά χρήσιμη τόσο στην Prolog όσο και στη LISP[1]. Λίστα είναι μια διατεταγμένη ν-άδα, που τα στοιχεία της περικλείονται σε αγκύλες (**[]**). Παραδείγματα λιστών:

```
[1,2,3]      [abc,'JOHN',4711,date(10,june,2006)]
```

Όπως βλέπετε, δεν είναι απαραίτητο τα στοιχεία μιας λίστας να είναι ομοειδή. Μπορούν ακόμα να είναι και άλλες λίστες:

```
[4,[yet,f(2,a(1,4))],[4,3,f,2,[0]],end]
```

Φυσικά, η λίστα με ένα μόνο στοιχείο έχει διαφορετική έννοια από το ίδιο το στοιχείο:

```
[bill] ≠ bill
```

Και βέβαια υπάρχει και η έννοια της ***κενής λίστας*** (empty list) που συμβολίζεται με άδειες αγκύλες:

```
[]
```

Στη λίστα επιτρέπεται μόνο μια πράξη: ο "αποκεφαλισμός" της, δηλαδή ο διαχωρισμός του πρώτου στοιχείου της από τα υπόλοιπα. Το πρώτο στοιχείο λέγεται ***head*** και η λίστα που περιέχει τα υπόλοιπα ***tail***. Πχ. για την [1,2,3] το head είναι το στοιχείο 1 και το tail η λίστα [2,3]. Στην Prolog δεν επιτυγχάνεται πραγματικός αποκεφαλισμός μιας λίστας, δηλαδή η λίστα που διαχωρίστηκε δεν αλλάζει. Επιτρέπεται μόνο η καταχώρηση των τιμών του head και του tail σε unbound[2] μεταβλητές. Η καταχώρηση αυτή επιτυγχάνεται με matching της λίστας που θέλουμε να τεμαχίσουμε με μια έκφραση της μορφής **[Head|Tail]**, όπου τα Head, Tail είναι μεταβλητές και το | είναι το ειδικό σύμβολο του αποκεφαλισμού (ο τελεστής της "πράξης").

Σύμφωνα με τα όσα είπαμε παραπάνω, κάθε λίστα (με μοναδική εξαίρεση την κενή) έχει ένα head και ένα tail. Το tail είναι επίσης λίστα, ενώ το head είναι στοιχείο της λίστας. Η παρακάτω συνάρτηση μας βοηθά να δούμε αυτά τα τμήματα:

```
chop(LIST,HEAD,TAIL):- LIST = [HEAD|TAIL].

?- chop([1,2,3],H,T).
H = 1
T = [2,3] ;
No

?- chop([2,3],H,T).
H = 2
T = [3] ;
```

1 Η LISP μάλιστα οφείλει το όνομά της στις λίστες: LISP = LISt Processing

2 μεταβλητή που δεν έχει πάρει ακόμα τιμή

```
No

?- chop([3],H,T).
H = 3
T = [] ;
No

?- chop([],H,T).
No
```

Προσπαθώντας να διαχωρίσει το head τής [] απέτυχε, αφού η κενή λίστα δεν έχει κανένα στοιχείο (άρα δεν έχει head).

```
?- chop([[1,2,3],4,5],H,T).
H = [1,2,3]
T = [4,5] ;
No
```

Εδώ το head ήταν λίστα; Ναι, γιατί το πρώτο στοιχείο της λίστας που διαχωρίσαμε ήταν λίστα.

Η chop/3 θα μπορούσε να γραφτεί πιο σύντομα με άμεσο matching (χωρίς τη χρησιμοποίηση του =) ως:

```
chop([H|T],H,T).
```

Το πιο ενδιαφέρον όμως είναι ότι η chop/3 δουλεύει και ανάποδα!!

```
?- chop(L,5,[10,20]).
L = [5,10,20] ;
No
```

Εμείς ορίσαμε την chop/3 ως διαχωρίστρια λιστών και τώρα βλέπουμε ότι βάζοντας τιμές στα arguments που κανονικά χρησιμοποιούνται για έξοδο, η chop/3 συνθέτει λίστες! Μια τέτοια συμπεριφορά δεν πρέπει να μας εκπλήσσει. Οταν παρουσιάσαμε τη συνάρτηση mother_of/2 είχαμε δεχτεί με απόλυτη φυσικότητα το γεγονός ότι μπορούσαμε να τη χρησιμοποιήσουμε για να βρούμε όχι μόνο τις μητέρες αλλά και τα παιδιά. Αρκετά συχνά συναντάμε στην Prolog τέτοιου είδους συναρτήσεις που δεν έχουν καθορισμένες "εισόδους" και "εξόδους" στα arguments. Οι συναρτήσεις αυτές ονομάζονται ***συμμετρικές***, δηλώνοντας έτσι ότι παρουσιάζουν ***συμμετρίες*** ως προς τον τύπο των ορισμάτων τους (αν δηλαδή είναι είσοδοι ή έξοδοι).

Η εξήγηση στο "φαινόμενο" είναι ότι μια συμμετρική συνάρτηση δεν χρησιμοποιεί κάποια arguments για να υπολογίσει κάποια άλλα. Εκφράζει απλώς τις σχέσεις μεταξύ τους. Και όταν οι σχέσεις είναι συμμετρικές (όπως στην περίπτωση γονέα-παιδιού), είναι συμμετρική και η συνάρτηση. Κατά κανόνα, μια συνάρτηση είναι συμμετρική όταν στο body της δεν περιέχει cut και όλες οι συναρτήσεις που καλεί είναι επίσης συμμετρικές. Η πρώτη προϋπόθεση (να μην περιέχει cut) είναι πιο χαλαρή από τη δεύτερη, μια και υπάρχουν περιπτώσεις που ένα green cut δε χαλάει τη συμμετρία. Αυτού του είδους οι συναρτήσεις είναι όπως καταλαβαίνετε συμμετρικές από τη φύση τους (τις χαρακτηρίζουμε ως ***φυσικά συμμετρικές***). Είναι όμως δυνατό με τον κατάλληλο

προγραμματισμό να αναγκάσουμε σχεδόν οποιαδήποτε συνάρτηση να συμπεριφερθεί συμμετρικά. Το μυστικό είναι να φτιάχνουμε ξεχωριστά τις συμμετρίες[1] της και μετά να προσθέτουμε κλήσεις που θα ελέγχουν τους τύπους των ορισμάτων. Πχ. για τη συνάρτηση paragontiko(X,Y) θα φτιάχναμε μια συμμετρία που θα ενεργοποιόταν όταν το Χ ήταν είσοδος και το Υ έξοδος και μια δεύτερη συμμετρία με τον αντίστροφο αλγόριθμο που θα ενεργοποιόταν όταν το Χ ήταν έξοδος και το Υ είσοδος. Συναρτήσεις που κάνουν έλεγχο για τα ορίσματα θα συναντήσουμε στην ενότητα 15.

Πολύ συχνά για να δηλώσουμε αν κάποιο όρισμα είναι είσοδος ή έξοδος βάζουμε ένα από τα σύμβολα +, -, ? πριν από αυτό. Π.χ.:

```
function(+Arg1,-Arg2,?Arg3)
```

Τα τρία αυτά σύμβολα έχουν τις εξής σημασίες:

Το σύμβολο **+** σημαίνει ότι το εν λόγω argument είναι μόνο είσοδος στη συνάρτηση. Δε θα μπορούσαμε δηλαδή εκεί να βάλουμε μια unbound μεταβλητή και να περιμένουμε να πάρει τιμή.

Το σύμβολο **-** σημαίνει ότι το argument είναι μόνο έξοδος. Μπορούμε δηλαδή να βάλουμε εκεί μια unbound μεταβλητή αλλά όχι μια bound ή μια προκαθορισμένη τιμή.

Το σύμβολο **?** σημαίνει ότι το argument που ακολουθεί μπορεί να χρησιμοποιηθεί είτε ως είσοδος είτε ως έξοδος στη συνάρτηση.

Ο παραπάνω συμβολισμός χρησιμοποιείται κατά κόρον στα manuals της Prolog και θα τον χρησιμοποιούμε κι εμείς από εδώ και πέρα. Αν θέλουμε για παράδειγμα να περιγράψουμε πώς καλείται η συνάρτηση chop/3 μπορούμε να γράψουμε:

```
chop(?List,?Head,?Tail)
```

...που σημαίνει ότι κάθε όρισμά της μπορεί να είναι είσοδος ή έξοδος. Όταν όλα τα ορίσματα σε μια συνάρτηση έχουν το σύμβολο ?, σημαίνει ότι η συνάρτηση είναι πλήρως συμμετρική. Αντίθετα για τη συνάρτηση paragontiko/2 θα γράφαμε:

```
paragontiko(+Number,-Result)
```

...και βλέπουμε με μια ματιά ότι δεν είναι συμμετρική. Το πρώτο της όρισμα χρησιμοποιείται για αποκλειστικά ως είσοδος και το δεύτερο αποκλειστικά ως έξοδος.

Ας δούμε τώρα μερικά παραδείγματα χρήσιμων συναρτήσεων για λίστες:

Κατ'αρχήν η member/2. Είναι μια συνάρτηση που μας λέει αν κάποια έκφραση είναι στοιχείο μιας λίστας ή όχι.

1 Μια συνάρτηση μπορεί να έχει παραπάνω από μια συμμετρία. Πχ. μια συνάρτηση με 3 ορίσματα μπορεί να κληθεί ως **(i,i,o)** (input-input-output, δηλαδή τα 2 πρώτα ορίσματα έχουν ήδη κάποιες τιμές ενώ στο τρίτο υπάρχει μια μεταβλητή που θα αποκτήσει τιμή), ή ως **(i,o,i)**, ή ως **(o,o,i)**, ... κλπ, θεωρητικά 9 συμμετρίες.

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

Η τερματική σχέση αυτού του ορισμού λέει ότι "*το X είναι στοιχείο μιας λίστας αν το X είναι το head της λίστας, δηλαδή το πρώτο στοιχείο της*". Η αναδρομική σχέση συμπληρώνει τον ορισμό: "*το X είναι στοιχείο μιας λίστας, αν είναι στοιχείο του tail της, δηλαδή αν είναι ένα από τα υπόλοιπα (μετά το head) στοιχεία της λίστας*".

```
?- member(2,[1,2,3,4]).
Yes

?- member(5,[1,2,3,4]).
No
```

Όπως η chop/3, έτσι και αυτή η συνάρτηση μας εκπλήσσει γιατί αποδεικνύεται πιο "έξυπνη" από ότι περιμέναμε:

```
?- member(X,[1,2,3,4]).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
No
```

Εμείς απλώς της ορίσαμε τι σημαίνει να είναι ένα στοιχείο μέλος μιας λίστας και τώρα βλέπουμε ότι αυτή μπορεί όχι μόνο να χαρακτηρίσει μια έκφραση ως "μέλος" ή "μη-μέλος", αλλά και να βρεί όλα τα στοιχεία που αποτελούν μια λίστα. Φαίνεται εδώ η δύναμη του πλήρους ορισμού. Αν ορίσουμε το πρόβλημά μας σωστά και με πληρότητα, η ανταπόκριση της Prolog θα είναι κάτι παραπάνω από ικανοποιητική γιατί θα δώσει στους ορισμούς μας διαστάσεις που εμείς ίσως δεν είχαμε φανταστεί, όταν τους γράφαμε.

Αλλη μια ενδιαφέρουσα συνάρτηση είναι η append/3. Η append ενώνει δυο λίστες για να δημιουργήσει μια τρίτη, σύμφωνα με το σχήμα:

[1,2,3] + [4,5] = [1,2,3,4,5]

Ορίζεται ως εξής (το τρίτο argument είναι το αποτέλεσμα):

```
append([],L,L).
append([H|T],L1,[H|L2]):- append(T,L1,L2).
```

"*Η ένωση της κενής λίστας με οποιαδήποτε λίστα, δίνει την ίδια τη λίστα*". Προσέξτε ότι γράψαμε **append([],L,L)** και όχι **append(L,[],L)**, που λογικά στέκει εξίσου, γιατί στην αναδρομική σχέση έχουμε σκοπό να τεμαχίσουμε την πρώτη λίστα (που θα καταλήξει κάποτε να γίνει κενή) και όχι τη δεύτερη.

Η αναδρομική σχέση λέει: "*για να ενώσουμε δυο λίστες, κρατάμε το head της πρώτης (H) { με το [H|*

T] }, ενώνουμε το tail (T) της πρώτης λίστας με τη δεύτερη λίστα (L1) { με την append(T,L1,L2) } και στο αποτέλεσμα (L2) συνδέουμε το head που κρατήσαμε, παίρνοντας έτσι το [H|L2]".

```
?- append([a,b,c],[d,e],X).
X = [a,b,c,d,e] ;
No
```

Είναι και συμμετρική:

```
?- append(X,[d,e],[a,b,c,d,e]).
X = [a,b,c] ;
No

?- append([a,b,c],X,[a,b,c,d,e]).
X = [d,e] ;
No
```

Στον ορισμό της παρακάτω συνάρτησης χρησιμοποιούνται και οι δυο "συμμετρίες" της append/3:

```
take(L,X,L2):- append(B,[X|A],L),
               append(B,A,L2).
```

Δεδομένης μιας λίστας L και ενός στοιχείου της X, η πρώτη append χωρίζει τη λίστα σε δυο κομμάτια: στο [X|A] πού έχει για πρώτο στοιχείο το X και στο B που τελειώνει στο στοιχείο πριν το X. Η δεύτερη append ενώνει τις B και A σε μια νέα λίστα, που περνά ως τρίτο argument στην take/3. Και επειδή η ανάλυση ελάχιστα διαφωτίζει τη λειτουργία μιας συνάρτησης, δείτε ένα παράδειγμα:

```
?- take([a,b,c,d,e],c,L).
L = [a,b,d,e] ;
No
```

Εδώ φαίνεται ότι η take/3 αφαιρεί ένα στοιχείο από μια λίστα. Για άλλη μια φορά βλέπουμε πόσο δυνατή δομή είναι η λίστα. Παρόλο που η μόνη πράξη που ορίζεται πάνω της είναι ο αποκεφαλισμός, τελικά είναι αρκετή για να χειριστούμε τη λίστα με όποιον τρόπο φανταστούμε.

Ως λίστες θεωρεί η Prolog και τα strings (και ως λίστες τα χειρίζεται). String στην Prolog είναι μια έκφραση που εγκλείεται μέσα σε διπλά εισαγωγικά, πχ: "ABCD". Κοιτάξτε τι καταλαβαίνει η Prolog αν της δείξουμε ένα string:

```
?- X = "ABCD".
X = [65, 66, 67, 68] ;
No
```

Για την Prolog λοιπόν δεν υπάρχουν strings. Στη θέση τους υπάρχουν λίστες ακεραίων. Οι ακέραιοι αντιστοιχούν στις ASCII τιμές των στοιχείων του string.

15. Μερικές Χρήσιμες Built-In Συναρτήσεις

Εστω το πρόγραμμα:

```
man(peter).    man(jimmy).    man(george).
```

Ως γνωστόν, αν ρωτήσουμε:

```
?- man(X).
X = peter ;
X = jimmy ;
X = george ;
No
```

η Prolog δίνει όλες τις δυνατές απαντήσεις. Αν όμως πούμε:

```
?- once(man(X)).
X = peter ;
No
```

θα δώσει μόνο μια απάντηση, την πρώτη που θα βρεί, και καμμία άλλη. Η once/1 είναι μια εντολή που παίρνει ως όρισμα μια άλλη συνάρτηση και την υποχρεώνει να φερθεί ντετερμινιστικά. Έχει οριστεί εσωτερικά στην Prolog ως:

```
once(Goal):- Goal, !.
```

Ακριβώς αντίθετη είναι η λειτουργία της repeat/0. Η repeat/0 είναι μια μη ντετερμινιστική αναδρομική συνάρτηση που μπορεί να οριστεί ως:

```
repeat.
repeat:- repeat.
```

Οπως βλέπουμε, δημιουργεί ένα ατέρμονο loop χωρίς να έχει κάποια μεταβλητή για να της δώσει τιμή. Επιτυγχάνει πάντα, τόσο κατά την ορθή φορά εκτέλεσης όσο και κατά το backtracking. Χρησιμοποιείται ως εξής:

```
?- repeat, write('HELLO'), nl, fail.
HELLO
HELLO
HELLO
HELLO
  :
  :
  :
```

Περικλείουμε ό,τι θέλουμε να επαναληφθεί μεταξύ της repeat/0 και της fail/0. Η fail ενεργοποιεί το backtracking και η repeat δίνει συνέχεια νέες εναλλακτικές τιμές TRUE στον εαυτό της.

Επίσης, τις συναρτήσεις member/2 και append/3 που είδαμε στην προηγούμενη ενότητα, στις περισσότερες εκδόσεις Prolog τις βρίσκουμε έτοιμες, ως built-in συναρτήσεις1.

Η Prolog, σαν γλώσσα με έμφαση στη λογική, έχει την ανάγκη να εκφράσει τους λογικούς ποσοδείκτες (υπαρξιακό και καθολικό). Και ενώ ο μεν υπαρξιακός μπορεί εύκολα να εκφραστεί με απλή ακολουθία όρων, πχ. "*υπάρχει γυναίκα που είναι νταλικέρισσα*":

```
woman(X), truck_driver(X)
```

...για τον δε καθολικό υπάρχει η συνάρτηση forall/2 που στο πρώτο argument παίρνει μια έκφραση γεννήτρια (που δίνει τιμές σε μια μεταβλητή) και στο δεύτερο argument περιέχει ελέγχους ή εντολές που θα εκτελεστούν για όλες τις τιμές που μπορεί να δώσει η γεννήτρια. Δείτε το παράδειγμα:

```
?- forall(member(X,[1,2,3]),(write(X),tab(3))).
1   2   3
X=_G455 ;
No
```

Αγνοείστε το side-effect του X=_G455, όταν χρησιμοποιούμε τη forall/2 σε πρόγραμμα δεν περνάμε το X σε κανένα argument. Σημειώστε όμως ότι για να "στριμώξουμε" πολλές εντολές σε ένα όρισμα τις περιβάλλουμε με παρενθέσεις: **(write(X),tab3))**. Το παράξενο αυτό σύνολο (που μπορεί να θεωρηθεί συνάρτηση που έχει ορίσματα αλλά όχι functor) ονομάζεται ***tuple***.

Στη συνέχεια θα εξετάσουμε μια εντολή που συγκεντρώνει όλες τις λύσεις μιας συνάρτησης σε μία λίστα:

```
woman(helen).    woman(jenny).    woman(mary).

?- findall(X,woman(X),L).
X = _G320
L = [helen, jenny, mary] ;
No
```

Το πρώτο argument της findall/3 είναι το όνομα της μεταβλητής (εδώ X). Το δεύτερο argument είναι μια έκφραση-γεννήτρια τιμών για τη μεταβλητή του πρώτου ορίσματος. Τέλος, το τρίτο όρισμα περιέχει το όνομα της λίστας που θα συγκεντρώσει τις τιμές της μεταβλητής.

Είναι γνωστή η χρήση της εντολής write/1 (output στην οθόνη). Η αντίστοιχη εντολή για input από το πληκτρολόγιο είναι η read/1. Οταν το argument της read/1 είναι unbound μεταβλητή τότε παίρνει την τιμή που θα γράψουμε στο πληκτρολόγιο:

```
?- read(Term).
|: test.
Term = test ;
No
```

1 Τη συνάρτηση member/2 σε κάποιες εκδόσεις της Prolog μπορεί να την συναντήσουμε με το όνομα "on", αλλά η λειτουργία είναι η ίδια.

Η τελεία είναι απαραίτητη στο τέλος του input. Μπορούμε να καλέσουμε τη read/1 με προκαθορισμένη τιμή στο argument. Τότε περιμένει από το χρήστη να γράψει αυτή την τιμή:

```
?- read(password).
|: hello.
No

?- read(password).
|: password.
Yes
```

Input χαρακτήρα-προς-χαρακτήρα μπορεί να επιτευχθεί με τις get/1 και get_code/1. Η get_code/1 επιστρέφει ή ελέγχει την ASCII τιμή του χαρακτήρα που θα πληκτρολογήσουμε.

```
?- get_code(X).
|: P{ENTER}
X = 80
Yes

?- get0(81).
|: Q{ENTER}
Yes

?- get0(81).
|: P{ENTER}
No
```

Η get/1 έχει παρόμοια λειτουργία, μόνο που αγνοεί τους blank χαρακτήρες (χαρακτήρες με ASCII 32 ή μικότερο).

```
?- get(80).
|:        P{ENTER}
Yes
```

Πολλές φορές χρειάζεται να ελέγξουμε την ταυτότητα ενός όρου, αν είναι δηλαδή μεταβλητή ή λίστα ή σταθερά ή compound object κλπ. Για το σκοπό αυτό η Prolog διαθέτει μια σειρά από συναρτήσεις ελέγχου που έχουν όλες arity 1. Στο argument βάζουμε τον όρο που θέλουμε να ελέγξουμε και παίρνουμε TRUE ή FALSE ανάλογα.

var(+X**)**

Γίνεται TRUE αν το X είναι unbound μεταβλητή.

nonvar(+X**)**

Δίνει TRUE όταν το X είναι ο,τιδήποτε άλλο εκτός από unbound μεταβλητή.

atom(+X**)**

TRUE, όταν το X είναι σταθερά, ή κενή λίστα.

integer(+X**)**
TRUE, για X ακέραιο.

atomic(+X**)**
Η ένωση των atom/1 και integer/1. Γίνεται TRUE αν το X είναι σταθερά, κενή λίστα ή ακέραιος.

compound(+X**)**
TRUE, αν το X είναι compound object (ακόμα και με ένα argument).

Από τη στιγμή που αναγνωρίσουμε την ταυτότητα του όρου, ίσως θελήσουμε να απομονώσουμε μερικά τμήματά του και να φτιάξουμε άλλους όρους από αυτόν. Οι εντολές arg/3, functor/3 και =../2 εκτελούν τέτοιες μετατροπές.

arg(?N, +Term, ?Arg**)**
Επιστρέφει το N-οστό argument του όρου Term. Το Term μπορεί να είναι ή compound object ή λίστα. Η λίστα θεωρείται ως compound object με 2 arguments. Το πρώτο είναι το head της λίστας και το δεύτερο είναι το tail της. Παραδείγματα:

```
?- arg(2,append([1],[2,3],A),X).
A = _G379
X = [2, 3] ;
No

?- arg(1,[a,b,c],X).
X = a ;
No

?- arg(2,[a,b,c],X).
X = [b, c] ;
No
```

functor(?Term, ?Functor, ?Arity**)**
Συνδέει έναν όρο με το όνομα του functor του και το arity του. Η συνάρτηση μπορεί να χρησιμοποιηθεί είτε για την ανάλυση ενός όρου είτε για την κατασκευή του:

```
?- functor(f(x,y,z),A,B).
A = f
B = 3 ;
No

?- functor(X,g,2).
X = g(_G339, _G340) ;
No
```

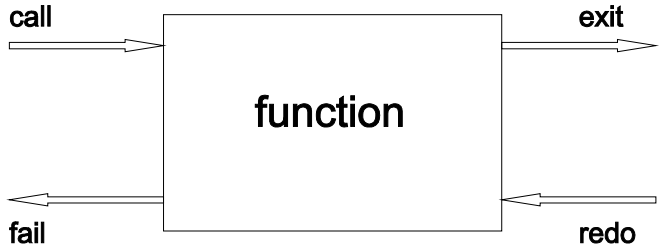
```
?Term =.. ?List
```

Ο τελεστής **=..** λέγεται **univ** (το όνομα που είχε στην πρώτη Γαλλική έκδοση της Prolog), και επιτυγχάνει τη μετατροπή ενός object (compound ή όχι) σε λίστα και αντίστροφα. Το head της λίστας αντιστοιχεί στον functor του object και το tail της στο tuple με τα arguments. Πχ:

```
?- X =.. [likes,paul,prolog].
X = likes(paul,prolog) ;
No

?- X =.. [12].
X = 12 ;
No

?- f(a,b) =.. X.
X = [f,a,b] ;
No

?- f(a,g(b)) =.. X.
X = [f,a,g(b)] ;
No

?- likes(a,b) =.. [X|Y].
X = likes
Y = [a,b] ;
No

?- likes =.. [likes].
Yes

?- likes =.. likes.
ERROR: =../2: Type error: `list' expected, found `likes'
```

16. Debugging (Αποσφαλμάτωση)

Σπάνια ένα πρόγραμμα τρέχει σωστά εξαρχής χωρίς να χρειαστούν διορθώσεις. Ένα ισχυρό εργαλείο αποσφαλμάτωσης (debugging) χρειάζεται σε κάθε γλώσσα προγραμματισμού. Στην Prolog το debugging είναι συνυφασμένο με μια διαδικασία που λέγεται tracing και δίνει στον προγραμματιστή τη δυνατότητα να παρακολουθεί βήμα-βήμα την εκτέλεση του προγράμματος: ποιά συνάρτηση καλείται, αν πέτυχε το matching, τι τιμές έχουν οι μεταβλητές, πότε αρχίζει το backtracking κλπ. Για να μπορέσουμε όμως να ερμηνεύσουμε σωστά τα μηνύματα του tracing πρέπει να καταλάβουμε με ποιό τρόπο "βλέπει" τις συναρτήσεις. Η άποψη του tracing φαίνεται στο ακόλουθο σχήμα:



Φανταστείτε κάθε συνάρτηση ως ένα κουτί με δυο εισόδους και δυο εξόδους. Οι είσοδοι έχουν τα ονόματα **call** και **redo** και οι έξοδοι τα **exit** και **fail**. Με ένα όνομα οι τέσσερεις αυτές είσοδοι/έξοδοι λέγονται ***ports***[1]. Ας τα δούμε ένα-ένα:

exit: Η έξοδος αυτή ενεργοποιείται όταν η συνάρτηση πετύχει. Ανεξάρτητα αν πρόκειται για ντετερμινιστική συνάρτηση ή όχι, αδιάφορα αν είναι η πρώτη ή η τελευταία λύση, αν η συνάρτηση κάποια στιγμή γίνει TRUE, θα ενεργοποιηθεί η exit.

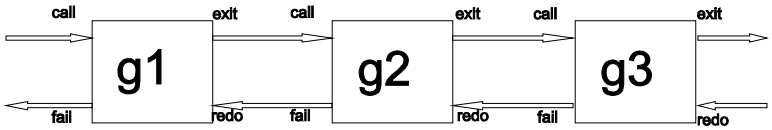
fail: Η έξοδος αυτή ενεργοποιείται όταν η συνάρτηση αποτύχει (γίνει FALSE). Φυσικά δεν είναι ποτέ δυνατόν η fail και η exit να ενεργοποιούνται συγχρόνως.

call: Η είσοδος αυτή προκαλεί την συνηθισμένη κλήση μιας συνάρτησης. Κάτι όμως που πρέπει να τονίσουμε (για να δούμε τη διαφορά από την redo) είναι ότι η κλήση αυτή γίνεται "από την αρχή". Δηλαδή οι μεταβλητές θα πάρουν τις αρχικές τους τιμές, οι κανόνες και τα γεγονότα που θα χρησιμοποιηθούν θα είναι τα πρώτα στη σειρά και γενικά, οποιοιδήποτε pointers για εναλλακτικές κλήσεις δεν έχουν πάρει ακόμα τιμή.

redo: Η είσοδος αυτή προκαλεί την αναζήτηση εναλλακτικών λύσεων από τη συνάρτηση. Φυσικά έχει νόημα μόνο για μη ντετερμινιστικές συναρτήσεις. Το redo σε μια ντετερμινιστική συνάρτηση θα προκαλέσει την ενεργοποίηση της εξόδου fail.

Ακολουθία συναρτήσεων (όπως παρουσιάζεται στο body ενός κανόνα) παριστάνεται σχηματικά ως εξής:

1 Η SWI-Prolog έχει και δυο επιπλέον ports τα οποία ονομάζει **unify** και **exception**. Η ακριβής εξήγηση της χρήσης τους ξεφεύγει από το σκοπό αυτού του εντύπου. Χονδρικά μπορούμε να πούμε ότι το unify είναι μια ειδική περίπτωση call, ενώ το exception είναι μια ειδική περίπτωση fail. Για περισσότερες λεπτομέρειες δείτε το on-line manual της SWI-Prolog.



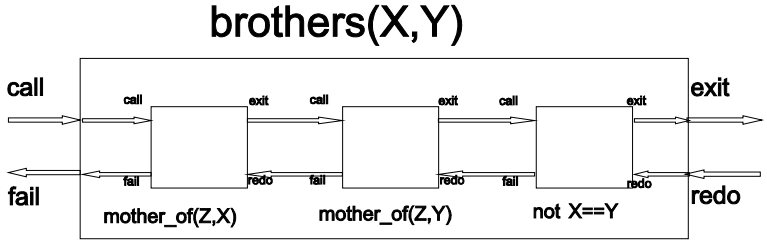

```
g:- g1, g2, g3.
```

Το exit της μιας συνάρτησης ενεργοποιεί το call της επόμενης και το fail μιας συνάρτησης ενεργοποιεί το redo της προηγούμενης. Ετσι, τα βέλη call-exit παριστάνουν την ορθή φορά εκτέλεσης, ενώ τα fail-redo το backtracking. Παρατηρήστε ότι η τελευταία συνάρτηση δεν έχει τρόπο να ενεργοποιήσει το redo της. Ακόμα, αν ενεργοποιηθεί το fail της πρώτης συνάρτησης, γίνεται fail όλος ο κανόνας. Φυσικά κάθε μια από αυτές τις συναρτήσεις μπορεί να καλεί άλλες και το σχήμα περιπλέκεται περισσότερο.

Για παράδειγμα, ο γνωστός ορισμός:

```
brothers(X,Y):- mother_of(Z,X), mother_of(Z,Y), \+ X==Y.
```

...αντιστοιχεί στο διάγραμμα:



Μόλις κληθεί η brothers(X,Y) θα ενεργοποιήσει το call της mother_of(Z,X). Αν αυτή κάνει fail θα ενεργοποιηθεί το fail της brothers(X,Y) και η διαδικασία θα σταματήσει εκεί. Αν όμως η mother_of(Z,X) δώσει κάποια λύση, τότε θα ενεργοποιηθεί το exit της, και αμέσως το call της συνάρτησης mother_of(Z,Y) με την οποία είναι συνδεδεμένο. Αν αυτή κάνει fail, θα ενεργοποιηθεί το redo της mother_of(Z,X) αναγκάζοντάς την να αναζητήσει εναλλακτική λύση. Αν βρεθεί εναλλακτική λύση θα ενεργοποιηθεί πάλι το exit της mother_of(Z,X) και το call της mother_of(Z,Y) και η διαδικασία θα συνεχιστεί μέχρι η να ενεργοποιηθεί ή το exit της brothers(X,Y) ή το fail της. Σημειώστε ότι αν έχουμε καλέσει την brothers(X,Y) από το prompt τότε κάθε φορά που πατάμε το πλήκτρο ';' για να δούμε εναλλακτικές λύσεις στην πραγματικότητα ενεργοποιούμε το redo της brothers(X,Y) και κατά συνέπεια το redo της \+X==Y η οποία ως ντετερμινιστική κάνει fail και ενεργοποιεί το redo της mother_of(Z,Y) για να βρει νέα τιμή για το Y.

Και τώρα που είδαμε τι περίπου γίνεται, ας δούμε πώς τα χρησιμοποιούμε όλα αυτά στο debugging. Έστω ότι έχουμε φορτώσει στη μνήμη το πρόγραμμα του παραγοντικού που εξετάσαμε στις ενότητες 12 και 13:

```
?- listing(paragontiko).
paragontiko(1,1):- !.
paragontiko(X,Y):- X2 is X-1,
                   paragontiko(X2,Y2),
                   Y is Y2*X.
Yes
```

Η λειτουργία του tracing ενεργοποιείται με την εντολή trace/0, ακολουθούμενη από τη συνάρτηση που θέλουμε να εξετάσουμε:

```
?- trace, paragontiko(4,X).
   Call: (9) paragontiko(4, _G336) ?
```

Τώρα η Prolog έχει μπει σε αυτό που λέμε ***debug-mode***. Βλέπουμε ότι έχει ξεκινήσει την κλήση της συνάρτησης αλλά την έχει "παγώσει" και περιμένει από εμάς να δώσουμε κάποια εντολή για το πώς να συνεχίσει.Σε αυτό το mode λειτουργίας η Prolog αναγνωρίζει ένα σύνολο πλήκτρων καθένα από τα οποία αντιστοιχεί σε μια εντολή. Το πλήρες σύνολο πλήκτρων και τη σημασία του καθενός μπορούμε να τα δούμε αν πατήσουμε το '?'. Όμως το συνηθέστερο είναι να πατήσουμε το 'Enter' ή το 'c' που αντιστοιχούν στην εντολή ***creep***, που σημαίνει να προχωρήσει μόλις ένα βήμα στην εκτέλεση του προγράμματος και να περιμένει επόμενη εντολή από εμάς. Έτσι, πατώντας συνεχώς 'Enter' βλέπουμε τις διαδοχικές κλήσεις μέχρι το τελικό αποτέλεσμα:

```
   Call: (9) paragontiko(4, _G336) ? creep
^  Call: (10) _L160 is 4-1 ? creep
^  Exit: (10) 3 is 4-1 ? creep
   Call: (10) paragontiko(3, _L161) ? creep
^  Call: (11) _L179 is 3-1 ? creep
^  Exit: (11) 2 is 3-1 ? creep
   Call: (11) paragontiko(2, _L180) ? creep
^  Call: (12) _L198 is 2-1 ? creep
^  Exit: (12) 1 is 2-1 ? creep
   Call: (12) paragontiko(1, _L199) ? creep
   Exit: (12) paragontiko(1, 1) ? creep
^  Call: (12) _L180 is 1*2 ? creep
^  Exit: (12) 2 is 1*2 ? creep
   Exit: (11) paragontiko(2, 2) ? creep
^  Call: (11) _L161 is 2*3 ? creep
^  Exit: (11) 6 is 2*3 ? creep
   Exit: (10) paragontiko(3, 6) ? creep
^  Call: (10) _G336 is 6*4 ? creep
^  Exit: (10) 24 is 6*4 ? creep
   Exit: (9) paragontiko(4, 24) ? creep

X = 24 ;

No
```

Στην αρχή κάθε γραμμής μας δείχνει το όνομα του port που ενεργοποιείται. Το σύμβολο '^' δηλώνει ότι η συνάρτηση την οποία εξετάζει εκείνη τη στιγμή είναι εσωτερική (built-in) της Prolog και όχι κάποια που έχει φτιάξει ο προγραμματιστής. Η παρένθεση που ακολουθεί δείχνει το βάθος της εκτέλεσης. Στο παράδειγμά μας η πρώτη κλήση ξεκινάει από βάθος 9[1] και καλεί κάποιες άλλες συναρτήσεις με βάθος 10, κ.ο.κ. Το βάθος αυξάνεται κατά 1 κάθε φορά που περνάμε από το neck κάποιου κανόνα. Το βάθος μειώνεται κατά 1 κάθε φορά που συναντάμε την τελεία κάποιου κανόνα.

1 Εκ πρώτης όψεως θα περιμέναμε το πρόγραμμα να ξεκινάει από βάθος 1 κι όχι από κάποιο μεγαλύτερο. Όμως στην πραγματικότητα το shell της Prolog (το περιβάλλον που μας δίνει το prompt) είναι μια διεργασία που τρέχει πάνω στον πυρήνα της Prolog και έχει ήδη κάποιο βάθος μεγαλύτερο του 1.

Έτσι μπορούμε να ακολουθήσουμε βήμα-βήμα την εκτέλεση του προγράμματός μας και να διαπιστώσουμε σε ποιό σημείο δεν λειτουργεί σωστά.

Αλλά ακόμα κι όταν τελειώσει το trace, η Prolog παραμένει σε debug mode. Αυτό φαίνεται από τη λέξη debug που έχει προστεθεί στην αρχή του prompt:

```
[debug] ?-
```

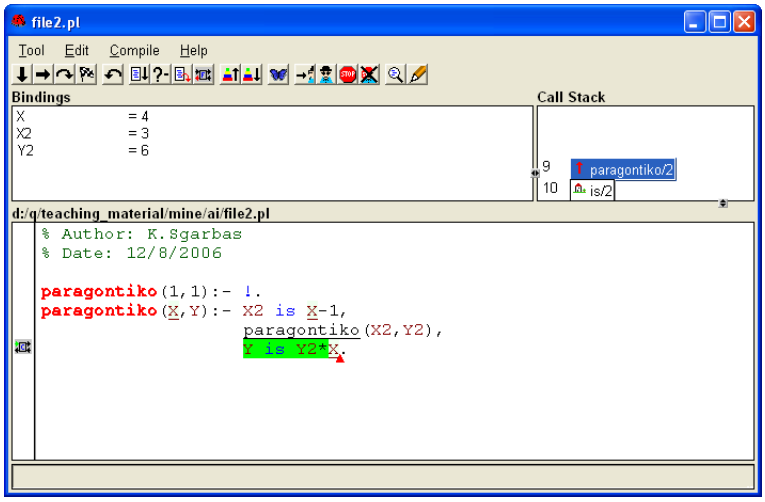
Για να επαναφέρουμε την Prolog στο mode φυσιολογικής λειτουργίας δίνουμε την εντολή nodebug/0:

```
[debug] ?- nodebug.
Yes

?-
```

Εναλλακτικά, η SWI-Prolog διαθέτει και ένα γραφικό περιβάλλον για debugging. Για να το ενεργοποιήσουμε δίνουμε την εντολή gtrace/0 αντί της trace/0. Π.χ.:

```
?- gtrace, paragontiko(4,X).
% The graphical front-end will be used for subsequent tracing
```

Η γλώσσα μάς ενημερώνει ότι από εδώ και πέρα το γραφικό περιβάλλον θα χρησιμοποιείται για όλες τις διαδικασίες tracing[1] και παράλληλα ανοίγει ένα καινούργιο παράθυρο στο οποίο μπορούμε να δούμε με παραστατικό τρόπο την εκτέλεση του προγράμματος πατώντας διαδοχικά το πλήκτρο 'Space':



Βέβαια, η διαδικασία που μόλις είδαμε μπορεί να είναι εφικτή για μικρά προγράμματα, όμως για μεγαλύτερα δεν είναι καθόλου εύκολο να ακολουθούμε βήμα-βήμα την κάθε συνάρτηση που συναντάμε. Γιαυτό το λόγο η Prolog μας επιτρέπει να δηλώσουμε ποιές από τις συναρτήσεις μας είναι πιθανόν να μην λειτουργούν σωστά, ώστε να κάνει trace μόνο σε αυτές. Αυτό το πετυχαίνουμε με τη βοήθεια των συναρτήσεων spy/1 και nospy/1.

Συγκεκριμένα, αν δώσουμε:

1 Αν δεν το θέλουμε αυτό, τότε μόλις κλείσουμε τον γραφικό debugger δίνουμε στο prompt την εντολή '?- **noguitracer.**'

```
?- spy( +Predicate_Name).
```

...τότε το πρόγραμμα θα εκτελείται κανονικά μέχρι να συναντηθεί call προς τη συνάρτηση με όνομα Predicate_Name, οπότε μπαίνουμε σε διαδικασία tracing. Στην ορολογία της Prolog λέμε ότι έχουμε βάλει ένα ***spy-point*** στη συνάρτηση Predicate_Name. Μπορούν να υφίστανται πολλά spy-points ταυτόχρονα.

Το spy-point αίρει η:

```
?- nospy( +Predicate_Name).
```

Για να σβήσουμε όλα τα spy-points δίνουμε:

```
?- nospyall.
Yes
```

17. Assert / Retract

Όπως έχουμε ήδη πεί, το πρόγραμμά μας αρχικά υφίσταται σε μορφή text file. Όταν αυτό το file γίνεται consulted, οι εντολές Prolog που περιέχει φορτώνονται στη μνήμη του υπολογιστή (στον Prolog Workspace). Σε αυτόν τον Workspace μπορούμε να επέμβουμε με τη χρήση των συναρτήσεων assert/1, retract/1 και των παραλλαγών τους. Συγκεκριμένα:

`assert( +Prolog_Clause)`

Προσθέτει το Prolog_Clause (μπορεί να είναι γεγονός ή κανόνας) στον Prolog Workspace. Αν υπάρχουν και άλλα clauses με το ίδιο όνομα, τοποθετεί το καινούργιο μετά από τα υπάρχοντα.

`asserta( +Prolog_Clause)`

Παραλλαγή της assert/1. Έχει ακριβώς την ίδια χρήση με την assert/1, με μόνη διαφορά ότι αν υπάρχουν και άλλα clauses με το ίδιο όνομα στον Prolog Workspace, το καινούργιο θα μπεί πρώτο, ΠΡΙΝ από όλα τα υπάρχοντα. Το **a** στο τέλος του ονόματός της μας υπενθυμίζει ακριβώς αυτό το γεγονός.

`assertz( +Prolog_Clause)`

Απόλυτα ισοδύναμη με την assert/1. Τοποθετεί το νέο clause στο τέλος των υπολοίπων με το ίδιο όνομα. Το **z** στο τέλος του ονόματός της μας υπενθυμίζει ακριβώς αυτό.

`retract( +Prolog_Clause)`

Αφαιρεί από τον Prolog Workspace το συγκεκριμένο Prolog_Clause.

`retractall( +Head_Description)`

Κάνει retract όσα clauses στον Prolog Workspace έχουν heads που γίνονται matched με την Head_Description (που μπορεί να περιέχει unbound ή ανώνυμες μεταβλητές).

Και τί χρησιμότητα έχουν όλα αυτά; Για ποιό λόγο αντί να γράφουμε **father(bill,jack)** να λέμε **assert(father(bill,jack))** και αντί για **brother(X,Y):- father(Z,X), father(Z,Y)** να γράφουμε **assert(brother(X,Y):- father(Z,X), father(Z,Y))**; Τί κερδίζουμε;

Η διαφορά έγκειται στο ότι το Prolog_Clause δεν είναι απαραίτητο να έχει οριστεί τη στιγμή που γράφουμε το πρόγραμμα. Μπορεί να είναι μια μεταβλητή που η τιμή της υπολογίζεται αργότερα, και τότε ένα compound object γίνεται ένα νέο γεγονός που δεν το είχε υπόψη του ο προγραμματιστής την ώρα που έγραφε τον κώδικα. Οι εντολές assert/1 και retract/1 χρησιμοποιούνται σε κάθε **μεταπρογραμματιστική**[1] εφαρμογή και αποτελούν τη βάση του learning για ένα "έξυπνο" πρόγραμμα. Κι ας εξηγήσουμε τα παραπάνω με ένα παράδειγμα. Θα χρησιμοποιήσουμε πάλι τη συνάρτηση paragontiko/2:

```
paragontiko(1,1):- !.
paragontiko(X,Y):- X2 is X-1,
                   paragontiko(X2,Y2),
                   Y is Y2*X.
```

1 Μεταπρογραμματισμός: Σειρά προγραμματιστικών τεχνικών που χρησιμοποιούν τα προγράμματα ως δεδομένα (data). Οι κάθε λογής interpreters και compilers αποτελούν παραδείγματα μεταπρογραμματιστικών εφαρμογών. Αυτομεταπρογραμματισμός καλείται η μέθοδος με την οποία ένα πρόγραμμα αλλάζει τον εαυτό του κατά τη διάρκεια της εκτέλεσής του.

Αν ζητήσουμε το παραγοντικό του 5, η συνάρτηση θα το υπολογίσει:

```
?- paragontiko(5,X).
X = 120 ;
No
```

Αν στη συνέχεια ζητήσουμε το παραγοντικό του 4 θα το ξαναϋπολογίσει (κακώς, γιατί το υπολόγισε και προηγουμένως που ζητήσαμε το 5!) Αλλά η Prolog δε θυμάται τι υπολόγισε προηγουμένως. Εδώ αρχίζει να φαίνεται η χρησιμότητα της assert/1 (για την ακρίβεια θα χρησιμοποιήσουμε την asserta/1). Το βελτιωμένο πρόγραμμα έχει ως εξής:

```
:- dynamic paragontiko/2.
paragontiko(1,1):- !.
paragontiko(X,Y):- X2 is X-1,
                   paragontiko(X2,Y2),
                   Y is Y2*X,
                   asserta(paragontiko(X,Y):- !).
```

Προσέξτε ότι μέσα στην asserta/1 γράψαμε το clause χωρίς τελεία στο τέλος! Δεν μπορεί να υπάρχουν δυο τελείες σε έναν κανόνα Prolog.

Προσέξτε επίσης στην αρχή του προγράμματος την εντολή:

```
:- dynamic paragontiko/2.
```

Το ISO Standard επιβάλλει να ορίσουμε ως dynamic τη συνάρτηση paragontiko/2, επειδή εμφανίζεται και ως όρισμα μέσα στην asserta/1 και απευθείας στον κώδικά μας.

Και τώρα είμασε έτοιμοι να τρέξουμε το πρόγραμμα. Αν γράψουμε:

```
?- paragontiko(5,X).
X = 120 ;
No
```

...εκτός από απάντηση παίρνουμε και ένα βελτιωμένο πρόγραμμα[1]:

```
?- listing.
paragontiko(5,120):- !.
paragontiko(4,24):- !.
paragontiko(3,6):- !.
paragontiko(2,2):- !.
paragontiko(1,1):- !.
paragontiko(A,B):-
     C is A-1,
```

1 Μην σας παραξενεύει το γεγονός ότι η εντολή listing/0. έχει αλλάξει τα ονόματα των μεταβλητών. Τα ονόματα των μεταβλητών όπως τα έχει γράψει ο προγραμματιστής στο source file δεν μεταφέρονται στον Prolog Workspace. Στην πραγματικότητα στον Prolog Workspace οι μεταβλητές φυλάσσονται με κωδικούς όπως _G340. Η εντολή listing/0 αντικαθιστά αυτές τις μεταβλητές με κεφαλαία γράμματα του λατινικού αλφαβήτου για να είναι πιο ευανάγνωστες.

```
        paragontiko(C,D),
        B is D*A,
        asserta((paragontiko(A,B):-!)).
Yes
```

Τώρα οποιοδήποτε παραγοντικό για αριθμό μικρότερο του 5 θα βρεθεί κατ'ευθείαν από fact, χωρίς ούτε μια αναδρομική κλήση. Επιπλέον, για παραγοντικά μεγαλύτερων αριθμών οι αναδρομικές κλήσεις θα σταματήσουν στην τερματική σχέση **paragontiko(5,120)** αντί για την **paragontiko(1,1)**. Βέβαια, αυτό που κερδίζει κανείς σε χρόνο υπολογισμού το χάνει σε μνήμη, αλλά τα 'έξυπνα' προγράμματα κατά κανόνα χρειάζονται επιπλέον μνήμη για να καταχωρήσουν τη γνώση τους. Το πρόγραμμα με το "βελτιωμένο" παραγοντικό μπορούμε να πούμε ότι εμφανίζει συμπτώματα learning, αφού η συμπεριφορά του βελτιώνεται όσο χρησιμοποιείται.

18. Τελεστές

Κοιτάξτε τα παρακάτω γεγονότα:

```
father(alex,bill).
father(bill,charlie).
father(charlie,don).
```

Δεν θα ήταν πιο ευανάγνωστα αν μπορούσαμε να τα εκφράσουμε κάπως έτσι;

```
alex is_father_of bill.
bill is_father_of charlie.
charlie is_father_of don.
```

Η Prolog μας δίνει και αυτή τη δυνατότητα. Το is_father_of μπορεί να οριστεί ως **τελεστής** (**operator**). Η έννοια του τελεστή είναι παρόμοια με την έννοια του functor, μόνο που ο τελεστής μπορεί να έχει ένα ή δύο ορίσματα μονάχα[1]. Σε αντιστάθμισμα αυτού του περιορισμού, οι τελεστές έχουν στοιχεία που δεν έχουν τα κατηγορήματα (τα predicates), όπως η ***προτεραιότητα*** και ο ***τύπος***. Πρίν εξηγήσουμε τι ακριβώς είναι αυτά, αξίζει να πούμε ότι από τη στιγμή που θα ορίσουμε έναν τελεστή, μπορούμε να τον χρησιμοποιούμε εν είδει κατηγορήματος σε οποιοδήποτε σημείο του προγράμματός μας. Στο παράδειγμά μας, για να συνδέσουμε τις έννοιες **father** και **is_father_of** προσθέτουμε έναν κανόνα σαν κι αυτόν:

```
X is_father_of Y :- father(X,Y).
```

Ο ορισμός του τελεστή πραγματοποιείται με τη συνάρτηση op/3 που έχει το format:

```
op( +Priority, +Type, +Symbol)
```

Συνήθως ο ορισμός πραγματοποιείται μέσα στον κώδικα με auto-executable goal. Π.χ.:

```
:- op(700,xfx,is_father_of).
```

Ο ορισμός πρέπει φυσικά να προηγείται της πρώτης χρήσης του τελεστή. Καιρός όμως να δούμε τι σημαίνουν τα arguments της op/3:

Symbol: Είναι το σύμβολο (το όνομα) του τελεστή. Χρησιμοποιούμε τη λέξη "σύμβολο" αντί για "όνομα" επειδή οι τελεστές που ορίζονται συνήθως έχουν σύμβολα του ενός-δυο χαρακτήρων.

Priority: Είναι ένας ακέραιος μεταξύ 1 και 1200 που δηλώνει την προτεραιότητα του τελεστή που ορίζουμε σε σχέση με τους άλλους (1200 είναι η ελάχιστη προτεραιότητα για έναν τελεστή και 1 η μέγιστη). Για παράδειγμα, ο τελεστής του πολλαπλασιασμού έχει μεγαλύτερη προτεραιότητα από τον τελεστή της πρόσθεσης κι έτσι η παράσταση: **3+5*2** σημαίνει **3+(5*2)** και όχι **(3+5)*2**.

1 Ίσως οι πιο γνωστοί μας τελεστές είναι τα σύμβολα των πράξεων: +,-,*,/.

Type: Ο τύπος του τελεστή ορίζει την προτεραιότητα του τελεστή ως προς τον εαυτό του (προσεταιριστικότητα - association). Για παράδειγμα, καθορίζει αν μια έκφραση όπως η **1-2-3** σημαίνει **(1-2)-3**, ή **1-(2-3)** ;

Οι δυνατοί τύποι είναι επτά. Τέσσερεις για τελεστές με ένα όρισμα και τρείς για τελεστές με δυο ορίσματα. Δεν υπάρχουν τελεστές που να συνδέουν τρία ή περισσότερα ορίσματα.

Τύποι για 1 όρισμα:

fx Οι τελεστές που ορίζονται με αυτόν τον τύπο λέγονται **μη-προσεταιριστικοί προθεματικοί τελεστές** (non-associative prefix operators). Ο τελεστής μπαίνει πρίν από το όρισμα (πχ. **man george**) και δεν επιτρέπεται επαναληπτική χρησιμοποίηση του τελεστή (δεν έχει νόημα το **man man george**).

xf Οι τελεστές που ορίζονται με αυτόν τον τύπο λέγονται **μη-προσεταιριστικοί επιθεματικοί τελεστές** (non-associative postfix operators). Ο τελεστής μπαίνει μετά από το όρισμα (πχ. **10%**) και δεν επιτρέπεται επαναληπτική χρησιμοποίηση του τελεστή (δεν έχει νόημα το **10%%**).

fy Οι τελεστές αυτού του τύπου λέγονται **προθεματικοί τελεστές με δεξιά προσεταιριστικότητα** (right-associative prefix operators). Ο τελεστής μπαίνει πρίν από το όρισμα (πχ. **not X**) και η επαναληπτική χρήση του προσεταιρίζει από δεξιά: **not not X** = not(not(X)).

yf Οι τελεστές αυτού του τύπου λέγονται **επιθεματικοί τελεστές με αριστερή προσεταιριστικότητα** (left-associative postfix operators). Ο τελεστής μπαίνει μετά από το όρισμα (πχ. **5!**) και η επαναληπτική χρήση του προσεταιρίζει από αριστερά: **5!!** = (5!)!

Τύποι για 2 ορίσματα:

xfx Οι τελεστές που ορίζονται με αυτόν τον τύπο λέγονται **μη-προσεταιριστικοί ενθεματικοί τελεστές** (non-associative infix operators). Ο τελεστής μπαίνει μεταξύ των ορισμάτων (πχ. **alex is_father_of bill**, **head :- body**) και δεν έχει νόημα η επαναληπτική χρησιμοποίησή του: **head :- body :- body2.**

yfx Οι τελεστές αυτοί λέγονται **ενθεματικοί τελεστές με αριστερή προσεταιριστικότητα** (left-associative infix operators). Ο τελεστής μπαίνει μεταξύ των ορισμάτων (πχ. **A-B**) και η επαναληπτική χρήση του προσεταιρίζει από αριστερά: **3-2-1** = (3-2)-1

xfy Οι τελεστές αυτού του τύπου λέγονται **ενθεματικοί τελεστές με δεξιά προσεταιριστικότητα** (right-associative infix operators). Ο τελεστής μπαίνει μεταξύ των ορισμάτων (πχ. **A^B**) και η επαναληπτική χρήση του προσεταιρίζει από δεξιά: **3^4^5** = 3^(4^5)

Μνημονικός κανόνας: Το f σε σχέση με τα x, y στον τύπο, δηλώνει τη θέση του τελεστή ως προς τα

ορίσματα. fx σημαίνει "*ο τελεστής μπροστά από το όρισμα*", xfy = "*ο τελεστής μεταξύ των ορισμάτων*". Ορισμα y στον τύπο δηλώνει ότι ο τελεστής προσεταιρίζει προς την πλευρά του ορίσματος (σε σχέση με το f). Ορισμα x σημαίνει ότι δεν προσεταιρίζει προς αυτήν την πλευρά. Ετσι, yfx σημαίνει: "*προσεταιρισμός από αριστερά*", ενώ xf = "*καθόλου προσεταιρισμός*".

Για να σβήσουμε τον ορισμό ενός τελεστή, αρκεί να τον επαναορίσουμε με προτεραιότητα μηδέν. Π.χ.:

```
:- op(0,xfx,is_father_of).
```

Η συνάρτηση current_op/3, format: **current_op(?Priority, ?Type, ?Symbol)**, μας δίνει πληροφορίες για τους τελεστές που έχουν οριστεί. Π.χ.:

```
?- current_op(X,Y,*).
X = 400
Y = yfx ;
No

?- current_op(X,Y,:-).
X = 1200
Y = fx ;

X = 1200
Y = xfx ;
No
```

Δώστε:

```
?- current_op(X,Y,Z).
```

...για να δείτε όλους τους τελεστές που έχει η Prolog μαζί με τις προτεραιότητες και τους τύπους τους, ή ακόμα καλύτερα γράψτε κάτι τέτοιο:

```
?- current_op(X,Y,Z), write(Z), tab(2), write(X),
     tab(2), write(Y), nl, fail.
```

Συνοψίζοντας, για να ορίσουμε έναν τελεστή, χρησιμοποιούμε την εντολή op/3 για να ορίσουμε τα χαρακτηριστικά του και μια σειρά κανόνων για να ορίσουμε τη λειτουργία του. Για παράδειγμα, έστω ο τελεστής **greater** που ορίζεται από το πρόγραμμα:

```
:- op(800,xfx,greater).
X greater Y:- X>Y.
```

Μόλις το τρέξουμε θα μπορούμε να γράφουμε:

```
?- 5 greater 2.
Yes

?- 4 greater 7.
```

```
No
```

Μια συχνή συνήθεια των πεπειραμένων προγραμματιστών είναι να χρησιμοποιούν ήδη ορισμένους τελεστές ως απλούς συνδέσμους μεταξύ όρων. Για παράδειγμα, δεδομένου ότι το / είναι το σύμβολο της διαίρεσης, το παρακάτω πρόγραμμα ίσως φανεί ακατανόητο:

```
vowel_list(['α'/'ά','ε'/'έ','η'/'ή','ι'/'ί','ο'/'ό',
            'υ'/'ύ','ω'/'ώ']).
stress(X,Y):- vowel_list(L), member(X/Y,L).
```

Αν σκεφτούμε όμως ότι η πράξη της διαίρεσης δεν εκτελείται (χρειάζεται να εκτελεστεί η εντολή **is** για να γίνει η πράξη) τότε καταλαβαίνουμε ότι το / απλώς συσχετίζει ζεύγη άτονων/τονισμένων φωνηέντων και η stress/2 χρησιμοποιείται για τη μετατροπή:

```
?- stress(α,X).
X = ά ;
No

?- stress(X,έ).
X = ε ;
No
```

19. Streams και Αρχεία

Στην Prolog υπάρχει η έννοια του **stream**. Το stream είναι ένα κανάλι (δίαυλος) εισόδου ή εξόδου. Μπορεί να είναι ο,τιδήποτε: το πληκτρολόγιο, ένας buffer, ένα αρχείο, ένας εκτυπωτής κλπ. Η ύπαρξη των streams κάνει πιο εύκολη την προσπέλασή τους. Με τις ίδιες εντολές που γράφουμε σε ένα αρχείο, μπορούμε να στείλουμε δεδομένα στο σειριακό. Αρκεί να αλλάξουμε τον κωδικό του stream στη συνάρτηση. Όλες οι συναρτήσεις εισόδου-εξόδου που έχουμε δεί μέχρι τώρα (write, read, get0, κλπ.) λειτουργούν και για τα streams, αρκεί να προσθέσουμε ένα πρώτο argument με τον κωδικό του stream. Π.χ:.

Ενώ η εντολή:

```
?- write('Message').
```

...γράφει το μήνυμα στην οθόνη, η:

```
?- write('$stream'(34018),'Message').
```

...γράφει το μήνυμα στο stream με κωδικό (stream identifier) '$stream'(34018), που μπορεί να είναι ένα αρχείο ή κάποια συσκευή εξόδου. Στη συνέχεια θα αναφερθούμε αποκλειστικά σε αρχεία, όμως η ίδια διαδικασία αφορά κάθε είδος stream.

Το αρχείο/stream ανοίγει με την εντολή open/3:

```
?- open('myfile.txt',read,X).
X = '$stream'(67892) ;
No
```

Η παραπάνω open/3 ανοίγει ένα file για ανάγνωση. Στο πρώτο όρισμα δέχεται το όνομα του αρχείου. Στο δεύτερο, ένα από τα keywords: **read**, **write**, **append** ή **update**. Τα keywords έχουν την εξής σημασία:

read	ανοίγει ένα υπάρχον αρχείο για ανάγνωση
write	δημιουργεί ένα νέο αρχείο για εγγραφή
append	ανοίγει ένα υπάρχον αρχείο για εγγραφή στο τέλος του (τοποθετεί τον δείκτη αρχείου στο τέλος)
update	ανοίγει ένα υπάρχον αρχείο για εγγραφή τοποθετώντας τον δείκτη αρχείου στην αρχή του

Συνεπώς, για τη δημιουργία ενός νέου αρχείου χρησιμοποιούμε την:

```
?- open('newfile.txt',write,X).
X = '$stream'(60238) ;
```

```
No
```

Το τρίτο όρισμα της open/3 είναι μια μεταβλητή που με την εκτέλεση της open/3 παίρνει ως τιμή έναν κωδικό της μορφής '$stream'(xxxxx). Ο κωδικός αυτός λέγεται ***stream identifier*** και χρησιμοποιείται ως "ταυτότητα" του αρχείου κάθε φορά που θέλουμε να διαβάσουμε από αυτό ή να γράψουμε σε αυτό. Ο κωδικός αυτός είναι επίσης απαραίτητος στην εντολή close/1 που κλείνει το αρχείο. Πχ. για να κλείσουμε το αρχείο newfile.txt που ανοίξαμε πριν λίγο, γράφουμε:

```
?- close('$stream'(60238)).
Yes
```

...και η close/1 το κλείνει.

Φυσικά, δεν χρειάζεται να θυμόμαστε το stream identifier που δίνει η Prolog σε κάθε αρχείο που ανοίγουμε. Αρκεί να περνάμε ως παράμετρο την αντίστοιχη μεταβλητή και η Prolog κάνει τα υπόλοιπα. Για παράδειγμα, αν θέλουμε να ανοίξουμε ένα αρχείο, να το επεξεργαστούμε και στη συνέχεια να το κλείσουμε, μπορούμε να το κάνουμε χωρίς να δούμε καν τον κωδικό του, κάπως έτσι:

```
?- open('myfile.txt',read,StrID), process(StrID), close(StrID).
Yes
```

...όπου η process/1 είναι μια συνάρτηση που έχουμε φτιάξει και κάνει όλες τις απαραίτητες διεργασίες στο αρχείο (στην περίπτωσή μας αναγνώσεις, μια που το έχουμε ανοίξει ως read). Προσέξτε πώς το StrID που παίρνει τιμή στην open/3, περνάει ως παράμετρος στην process/1 και στην close/ για να κλείσει.

Τί θα μπορούσε να περιλαμβάνει η process/1; Κατά πάσα πιθανότητα εντολές ανάγνωσης/εγγραφής για το αρχείο. Ανάγνωση/εγγραφή σε αρχεία πραγματοποιούν οι εντολές read/2 και write/2. Η χρήση τους είναι παρόμοια με τις γνωστές μας read/1 και write/1, μόνο που τώρα βάζουμε στο πρώτο argument το stream identification του αρχείου που θα χρησιμοποιηθεί:

```
read( +StreamID, +Term_to_read)
write(+StreamID, +Term_to_write)
```

Το θέμα είναι σε ποιό ακριβώς σημείο του αρχείου θα διαβάσουμε ή θα γράψουμε; Για το σκοπό αυτό, κάθε file έχει στη διάθεσή του έναν ***file pointer*** που καθορίζει σε ποιό σημείο θα γίνει η επόμενη εγγραφή/ανάγνωση. Ο pointer αυτός παίρνει μια αρχική τιμή με την κλήση της open/3 και αυξάνεται κάθε φορά που εκτελείται μια read/2 ή write/2 στο αρχείο. Η τιμή του pointer μπορεί να βρεθεί ή να αλλάξει με την εντολή seek/4. Το format της seek/4 είναι:

```
seek( +StrID, +Offset, +From, -NewLoc)
```

Το πρώτο όρισμα είναι το stream-id του αρχείου, το δεύτερο μια ποσότητα που ονομάζεται Offset και δηλώνει το πλήθος των bytes που θέλουμε να μετακινηθεί ο file pointer, στο τρίτο όρισμα (From) δηλώνουμε από πού θέλουμε να αρχίσει να μετράει η μετακίνηση (γράφουμε **current** για την τρέχουσα θέση του pointer, **eof** για το τέλος του αρχείου και **bof** για την αρχή του αρχείου), ενώ η έξοδος NewLoc παίρνει ως τιμή τη νέα θέση του file pointer, πάντα εκφρασμένη σε bytes από την αρχή του αρχείου.

Για να τα δούμε όλα αυτά με παραδείγματα. Έστω ότι το αρχείο myfile.txt περιέχει τα εξής δεδομένα:

```
abcd.
efgh.
ijkl.
```

Το ανοίγουμε πρώτα για ανάγνωση:

```
?- open('myfile.txt',read,StrID).
StrID='$stream'(67892) ;
No
```

Και τώρα ας δώσουμε μια εντολή read/2:

```
?- read('$stream'(67892),Data).
Data = abcd ;
No
```

Η read/2 διαβάζει όρους (terms) της Prolog, όπως ακριβώς έκανε και η read/1. Γιαυτό το λόγο διαβάζει μέχρι να συναντήσει τελεία[1]. Κοιτάξτε τί γίνεται αν ξαναδώσουμε την ίδια εντολή:

```
?- read('$stream'(67892),Data).
Data = efgh ;
No
```

Ξανά:

```
?- read('$stream'(67892),Data).
Data = ijkl ;
No
```

Και ξανά:

```
?- read('$stream'(67892),Data).
Data = end_of_file ;
No
```

Κάθε φορά που διαβάζουμε από το αρχείο, ο file pointer μετακινείται κατάλληλα, ώστε η επόμενη read/2 να διαβάσει τον επόμενο όρο. Όταν φτάσει στο τέλος του αρχείου, η σταθερά end_of_file θα εμφανίζεται ως αποτέλεσμα της ανάγνωσης, όσες φορές κι αν προσπαθήσουμε να διαβάσουμε το αρχείο.

Τώρα λοιπόν καταλάβαμε λοιπόν το ρόλο που παίζει ο file pointer, αντιλαμβανόμαστε ότι για να ξαναδιαβάσουμε κάποια δεδομένα από το αρχείο θα πρέπει να τον μετακινήσουμε σε άλλη θέση (ας πούμε στην αρχή) του αρχείου. Ένας τρόπος να το πετύχουμε αυτό είναι να κλείσουμε το αρχείο και να το ξανανοίξουμε. Ένας καλύτερος τρόπος είναι αυτός:

1 Αν θέλουμε να διαβάσουμε ένα αρχείο με άλλο τρόπο, υπάρχουν άλλες εντολές (πχ. η get/2 διαβάζει χαρακτήρα-χαρακτήρα).

```
?- seek('$stream'(67892),0,bof,NewPos).
NewPos = 0 ;
No
```

Λέμε στην seek/4 να μετακινήσει τον file pointer 0 bytes από την αρχή (bof) του αρχείου. Η μεταβλητή NewPos μάς ενημερώνει για τη νέα τιμή που πήρε ο pointer: 0 που σημαίνει ότι η Prolog αρχίζει το μέτρημα των bytes των αρχείων από το 0.

Τώρα μια νέα read/2 θα ξαναδιαβάσει τον πρώτο όρο:

```
?- read('$stream'(67892),Data).
Data = abcd ;
No
```

Καταλαβαίνουμε ότι τώρα ο file pointer είναι σε τέτοια θέση ώστε η επόμενη read/2 να διαβάσει τον όρο efgh. Για να δούμε τί θα γίνει αν τον μετακινήσουμε λίγο:

```
?- seek('$stream'(67892),2,current,NewPos).
NewPos = 8 ;
No

?- read('$stream'(67892),Data).
Data = gh ;
No
```

Η seek/4 μετακίνησε τον pointer 2 bytes από την τρέχουσα θέση. Έτσι, η read/2 αντί να διαβάσει από τον χαρακτήρα e, διάβασε από τον g και μετά.

Τί γίνεται αν θέλουμε να διαβάσουμε τη θέση του pointer χωρίς όμως να τον μετακινήσουμε; Να ένας τρόπος:

```
?- seek('$stream'(67892),0,current,NewPos).
NewPos = 12 ;
No
```

Στην ουσία μετακινήσαμε τον pointer 0 bytes από την τρέχουσα θέση του και διαβάσαμε τη νέα τιμή του: δείχνει στο 13ο byte του αρχείου (θυμηθείτε ότι η αρίθμηση αρχίζει από το 0).

Και πώς μπορούμε να διαβάσουμε έναν όρο χωρίς να μετακινήσουμε τον pointer; Δείτε μια ιδέα:

```
?- S='$stream'(67892), seek(S,0,current,SavePos),
   read(S,Data), seek(S,SavePos,bof,NewPos).
S = '$stream'(67892)
SavePos = 12
Data = ijkl
NewPos = 12 ;
No
```

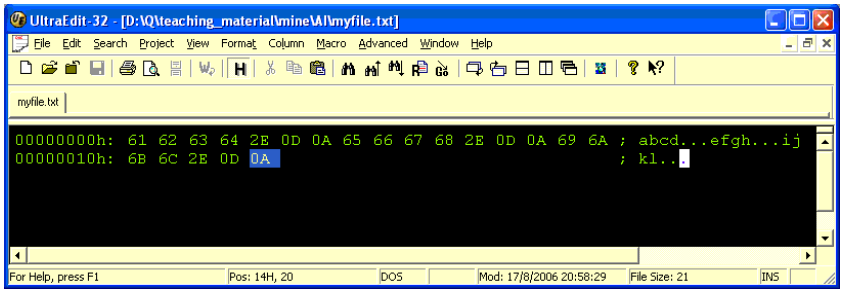
Διαβάσαμε την τρέχουσα θέση και την φυλάξαμε στη μεταβλητή SavePos. Μόλις διαβάσαμε τα Data μετακινήσαμε τον pointer SavePos bytes από την αρχή του αρχείου, θέση που συμπίπτει με την αρχική του, όπως μπορούμε να διαπιστώσουμε από τις τιμές των SavePos και NewPos. Προσέξτε και μια λεπτομέρεια: τη χρήση της μεταβλητής S για να κρατήσει το stream-id ώστε να μην το γράφουμε σε κάθε εντολή.

Δείτε κι έναν τρόπο να βρίσκουμε το μέγεθος του αρχείου που ανοίξαμε:

```
?- seek('$stream'(67892),0,eof,NewPos).
NewPos = 21 ;
No
```

Μετακινήσαμε τον pointer 0 bytes από το end-of-file και η NewPos πήρε την τιμή 21. Αυτό σημαίνει ότι το αρχείο έχει μέγεθος 21 bytes. Όμως για μισό λεπτό! Το αρχείο είχε τρεις όρους με 5 χαρακτήρες τον καθένα. Πώς δικαιολογείται το 21;

Τα αρχεία κειμένου περιέχουν και μη εκτυπώσιμους χαρακτήρες. Οι πιο συνηθισμένοι είναι οι χαρακτήρες με ASCII τιμές 13 (return) και 10 (new line). Για να τους δούμε χρειάζεται να ανοίξουμε το αρχείο με έναν hex-editor:



...και πράγματι διαπιστώνουμε ότι αποτελείται από 21 bytes ακριβώς.

Για να δούμε τί γίνεται αν θέλουμε να γράψουμε στο αρχείο. Η εντολή είναι η write/2:

```
?- write('$stream'(67892),mnop).
ERROR: write/2: No permission to output stream
`$stream(67892)'
```

...όμως δε μπορούμε να την χρησιμοποιήσουμε επειδή έχουμε ανοίξει το αρχείο ως αρχείο ανάγνωσης. Συνεπώς το κλείνουμε και το ξανανοίγουμε για ενημέρωση:

```
?- open('myfile.txt',update,X).
X = '$stream'(67892) ;
No
```

Μετακινούμε τον file pointer στο τέλος του αρχείου[1] για να γράψουμε εκεί:

1 Σημειώστε ότι αν είχαμε ανοίξει το αρχείο ως append, δεν θα χρειαζόταν να μετακινήσουμε τον pointer στο τέλος, όμως δεν θα μας επέτρεπε να γράψουμε σε προηγούμενες θέσεις, όπως σκοπεύουμε να κάνουμε σε λίγο.

```
?- seek('$stream'(67892),0,eof,NP).
NP = 21 ;
No
```

Και τώρα:

```
?- write('$stream'(67892),'mnop.').
Yes
```

Ας έγραψε Yes, η αλλαγή δεν έχει περάσει ακόμα στο αρχείο στον σκληρό δίσκο του υπολογιστή μας. Αν το ανοίξουμε με έναν editor δεν θα δούμε το νέο όρο. Πρέπει να κλείσουμε το αρχείο για να γίνει η αλλαγή. Όμως πρώτα ας κάνουμε και μια δεύτερη αλλαγή:

```
?- seek('$stream'(67892),-19,eof,NP).
NP = 7 ;
No
```

Μετακινηθήκαμε 19 bytes πίσω, και τώρα:

```
?- write('$stream'(67892),'qrst.').
Yes
```

...αντικαθιστούμε τον δεύτερο όρο του αρχείου με τον όρο qrst. Και κλείνουμε το αρχείο για να περάσουν οι αλλαγές στον δίσκο:

```
?- close('$stream'(67892)).
Yes
```

Αν τώρα ανοίξουμε το αρχείο με έναν editor θα δούμε:

```
abcd.
qrst.
ijkl.
mnop.
```

20. DCG-Rules

Με την Prolog είναι πολύ εύκολη η δημιουργία ***parsers***[1] και γενικά συστημάτων γλωσσικής επεξεργασίας (τόσο για φυσικές όσο και για τεχνητές γλώσσες). Αυτό επειδή η Prolog μας δίνει τη δυνατότητα να προγραμματίζουμε απευθείας σε DCG-rules.

Τι είναι τα DCG-rules; Τα αρχικά DCG προέρχονται από τις λέξεις: ***Definite Clause Grammar***. Η Definite Clause Grammar είναι ένα μοντέλο γραμματικής. Γραμματική είναι ένα σύνολο ορισμών που περιγράφουν τη δομή παραστάσεων (προτάσεων) που ανήκουν σε ένα σύνολο γνωστό με το όνομα "***γλώσσα***". ***Μοντέλο γραμματικής*** είναι η μορφή με την οποία θα εκφραστεί η συγκεκριμένη γραμματική. Για παράδειγμα, η ίδια γραμματική μπορεί να εκφραστεί με ένα σύνολο κανόνων ή με έναν πίνακα πιθανοτήτων ή με όποιον τρόπο θεωρήσει κανείς κατάλληλο. Ετσι, η ίδια γραμματική είναι δυνατό να εκφραστεί σε διαφορετικά μοντέλα και το κάθε μοντέλο να εκφράσει γραμματικές πολλών διαφορετικών φυσικών και τεχνητών γλωσσών[2]. Βέβαια, το κάθε μοντέλο θέτει τους δικούς του περιορισμούς ως προς τον πλούτο των γραμματικών φαινομένων που μπορεί να αποδώσει. Ειδικά για τις φυσικές γλώσσες, υπάρχει πάντα ένα χάσμα μεταξύ της γραμματικής του μοντέλου και της γραμματικής της γλώσσας. Πάρτε για παράδειγμα την Ελληνική γλώσσα. Εχει μια γραμματική από μόνη της. Αν εμείς προσπαθήσουμε να φτιάξουμε κάποιο πρόγραμμα που θα καταλαβαίνει Ελληνικά, θα το φτιάξουμε πρώτα να καταλαβαίνει απλές προτάσεις (με ένα ρήμα, ένα υποκείμενο κτλ.) μετά θα το βελτιώσουμε για πιο σύνθετες (με συνδέσμους, δευτερεύουσες), στη συνέχεια θα το βελτιώσουμε κι άλλο για να αναγνωρίζει παθητική φωνή, πλάγιο λόγο κλπ. Σε κάθε στάδιο κατασκευής το πρόγραμμά μας έχει μια δική του γραμματική, υποσύνολο της γραμματικής της Ελληνικής γλώσσας. Οταν οι δυο γραμματικές ταυτιστούν, το πρόγραμμά μας θα είναι τέλειο. Περιττό να πούμε ότι δεν έχει φτιαχτεί ακόμα ο τέλειος parser για καμμία φυσική γλώσσα. Εχει μάλιστα αποδειχτεί ότι οι ορισμένα μοντέλα γραμματικών ποτέ δε θα καταφέρουν να αποδώσουν συγεκριμένα φαινόμενα φυσικών γραμματικών. Ομως πολύ συχνά είμαστε ικανοποιημένοι με ένα σχετικά μικρό υποσύνολο της φυσικής γραμματικής. Αυτός είναι και ο λόγος (μαζί με την ευκολία υλοποίησης) που χρησιμοποιούμε ακόμα απλά μοντέλα σαν την DCG. Σε αυτές τις σημειώσεις δεν θα επεκταθούμε άλλο πάνω στο μεγάλο θέμα των γραμματικών που από μόνες τους αποτελούν ένα ξεχωριστό τομέα στην επιστήμη της ***υπολογιστικής γλωσσολογίας*** (***computational linguistics***). Σκοπός μας εδώ είναι να παρουσιάσουμε το συμβολισμό και τη χρήση του μοντέλου DCG.

Αλλά καιρός να δούμε ένα παράδειγμα προγράμματος με κανόνες DCG:

```
sentence     -->   noun_phrase, verb_phrase.
noun_phrase  -->   determiner, noun.
verb_phrase  -->   verb; verb, noun_phrase.
determiner   -->   [the].
noun         -->   [boy]; [house].
verb         -->   [likes].
```

Το παραπάνω πρόγραμμα αποτελεί μια υποτυπώδη γραμματική που μπορεί να αναγνωρίσει μερικές απλές Αγγλικές προτάσεις. Οι κανόνες διαφέρουν από αυτούς της Prolog μια και ο τελεστής δεν

1 parser = αναλυτής προτάσεων (συνηθέστερα συντακτικός)

2 Φυσικές γλώσσες λέμε τις γλώσσες που χρησιμοποιούνται στην διανθρώπινη επικοινωνία (Ελληνική, Αγγλική, Γαλλική κλπ), ενώ τεχνητές λέμε τις γλώσσες που κατασκευάστηκαν από τον άνθρωπο για την επικοινωνία με διάφορες συσκευές (συνήθως υπολογιστές πχ. FORTRAN, C, Prolog)

είναι πια το "**:-**" αλλά το "**-->**". Αυτοί οι DCG-κανόνες λέγονται και ***rewrite rules*** (***κανόνες επανεγγραφής***) επειδή καθορίζουν πώς θα "ξαναγραφτεί" το αριστερό μέλος του κανόνα (head), χρησιμοποιώντας το δεξί (body). Στο παράδειγμά μας, ο πρώτος rewrite rule μας λέει ότι "*μια πρόταση είναι ένα ονοματικό σύνολο που ακολουθείται από ένα ρηματικό σύνολο*". Ο δεύτερος κανόνας δηλώνει ότι "*ένα ονοματικό σύνολο αποτελείται από έναν προσδιοριστή και ένα ουσιαστικό*". Ο τρίτος DCG-rule προσδιορίζει την έννοια του ρηματικού συνόλου: "*μπορεί να είναι ή σκέτο ρήμα ή ένα ρήμα που ακολουθείται από ένα ονοματικό σύνολο*".

Η προηγούμενη ορολογία ήταν ορολογία του συντακτικού της Αγγλικής γλώσσας. Σε χοντρικές γραμμές μπορούμε να πούμε ότι το **noun_phrase** αντιστοιχεί στο δικό μας **υποκείμενο** ή **αντικείμενο** (κατά περίπτωση), το **verb_phrase** στο **κατηγόρημα** και το **determiner** στο **άρθρο**. Οι λέξεις:

sentence, noun_phrase, verb_phrase, determiner, noun, verb

που χρησιμοποιήσαμε, λέγονται **non-terminals**. Τα non-terminals είναι τα ονόματα των κατηγοριών που έχει η γραμματική μας.

Ο τέταρτος κανόνας επανεγγραφής λέει ότι "*προσδιοριστής είναι το άρθρο 'the'*", ο πέμπτος ότι "*ουσιαστικό μπορεί να είναι ένα από τα 'boy', 'house'*" και τέλος ο έκτος κανόνας επανεγγραφής μάς δίνει το μοναδικό ρήμα που διαθέτει η γραμματική μας, το 'likes'. Οι λέξεις:

the, boy, house, likes

λέγονται **terminals**. Τα terminals είναι οι λέξεις που αποτελούν το λεξιλόγιο της γραμματικής μας, αυτές δηλαδή που θα εμφανίζονται στις προτάσεις της. Παρατηρήστε ότι τα terminals μέσα στους rewrite-rules γράφονται ως μοναδιαίες λίστες.

Από τη στιγμή που η Prolog κάνει consult το παραπάνω DCG-πρόγραμμα, χρησιμοποιούμε τις built-in συναρτήσεις phrase/2 και phrase/3 για να κάνουμε τις ερωτήσεις μας.

Η phrase/2 έχει το format:

```
phrase( +Non_terminal, ?List_of_terminals)
```

όπου στο δεύτερο argument βάζουμε μια λίστα με τις λέξεις (terminals) που αποτελούν τη φράση που θέλουμε να ελέγξουμε και στο πρώτο argument βάζουμε το όνομα του non-terminal ως προς το οποίο θα ελέγξουμε τη φράση. Για παράδειγμα, για να ελέγξουμε αν η φράση "the boy likes the house" είναι sentence, γράφουμε:

```
?- phrase(sentence,[the,boy,likes,the,house]).
Yes
```

Ακόμα,

```
?- phrase(sentence,[the,boy,likes]).
Yes
```

...αλλά:

```
?- phrase(sentence,[the,girl,likes]).
No
```

...επειδή η λέξη "girl" δεν ανήκει στα terminals της γραμματικής μας. Και:

```
?- phrase(sentence,[the,likes,boy]).
No
```

...επειδή η σειρά των λέξεων δεν συμφωνεί με τους κανόνες DCG.

Μπορούμε να ελέγξουμε τις φράσεις ως προς οποιοδήποτε non-terminal:

```
?- phrase(noun_phrase,[the,house]).
Yes

?- phrase(determiner,[the]).
Yes
```

Η phrase/3 είναι παρόμοια, αλλά με ένα argument παραπάνω:

```
phrase( +Non_terminal, ?List_of_terminals, ?Rest_of_terminals)
```

Το τρίτο argument είναι μια λίστα με τις λέξεις που περισσεύουν αν αφαιρεθεί η αρχική ομάδα λέξεων που ταιριάζουν με το Non_terminal. Δηλαδή:

```
?- phrase(noun_phrase,[the,boy,likes,the,house],R).
R = [likes,the,house] ;
No
```

Το noun_phrase ήταν το [the,boy]. Το υπόλοιπο της φράσης μεταφέρθηκε στη μεταβλητή R.

```
?- phrase(verb_phrase,[the,boy,likes,the,house],R).
No
```

..."No", γιατί η φράση δεν αρχίζει από verb_phrase.

```
?- phrase(verb_phrase,[likes,the,house],R).
R = [the,house] ;
R = [] ;
No
```

Όπως διαπιστώνει κανείς από το format των phrase/2 και phrase/3, η γραμματική είναι συμμετρική. Εκτός από ανάλυση (parsing) κάνει και σύνθεση (generation):

```
?- phrase(sentence,S).
S = [the,boy,likes] ;
```

```
S = [the,boy,likes,the,boy] ;
S = [the,boy,likes,the,house] ;
S = [the,house,likes] ;
S = [the,house,likes,the,boy] ;
S = [the,house,likes,the,house] ;
No
```

Αυτές είναι όλες οι προτάσεις που 'καταλαβαίνει' η γραμματική μας. Και οι ακόλουθες είναι όλες οι ρηματικές φράσεις:

```
?- phrase(verb_phrase,V).
V = [likes] ;
V = [likes,the,boy] ;
V = [likes,the,house] ;
No
```

Βέβαια, πρόκειται για μια πολύ απλή γραμματική[1]. Ας την επεκτείνουμε λίγο, ώστε να αναγνωρίζει ενικό και πληθυντικό αριθμό. Αυτό το πετυχαίνουμε βάζοντας στα non-terminals τα κατάλληλα arguments:

```
sentence(N)     -->  noun_phrase(N), verb_phrase(N).
noun_phrase(N)  -->  determiner(N), noun(N).
verb_phrase(N)  -->  verb(N); verb(N), noun_phrase(_).
determiner(_)   -->  [the].
noun(singular)  -->  [boy]; [house].
noun(plural)    -->  [boys]; [houses].
verb(singular)  -->  [likes].
verb(plural)    -->  [like].
```

Για παράδειγμα, ο δεύτερος κανόνας λέει ότι "*Ενα ονοματικό σύνολο αποτελείται από έναν προσδιοριστή με αριθμό N (ενικό ή πληθυντικό), που ακολουθείται από ένα ουσιαστικό με αριθμό επίσης N.*" (Ο αριθμός N είναι μια συνηθισμένη μεταβλητή της Prolog.) Και τώρα ας κάνουμε μερικές ερωτήσεις:

```
?- phrase(sentence(X),[the,boy,likes,the,houses]).
X = singular ;
No

?- phrase(sentence(plural),[the,boys,like]).
Yes

?- phrase(sentence(plural),[the,boy,likes]).
No

?- phrase(sentence(X),[the,boys,likes]).
No
```

1 Είναι εμφανές ότι μερικές από τις προτάσεις που "καταλαβαίνει" ο parser μας δεν στέκουν νοηματικά. Αυτό συμβαίνει γιατί όλοι οι κανόνες που χρησιμοποιήσαμε περιέγραφαν τη συντακτική δομή των προτάσεων και όχι τη σημασία τους. Σημασιολογικούς parsers μπορούμε να φτιάξουμε με πιο περίπλοκες τεχνικές.

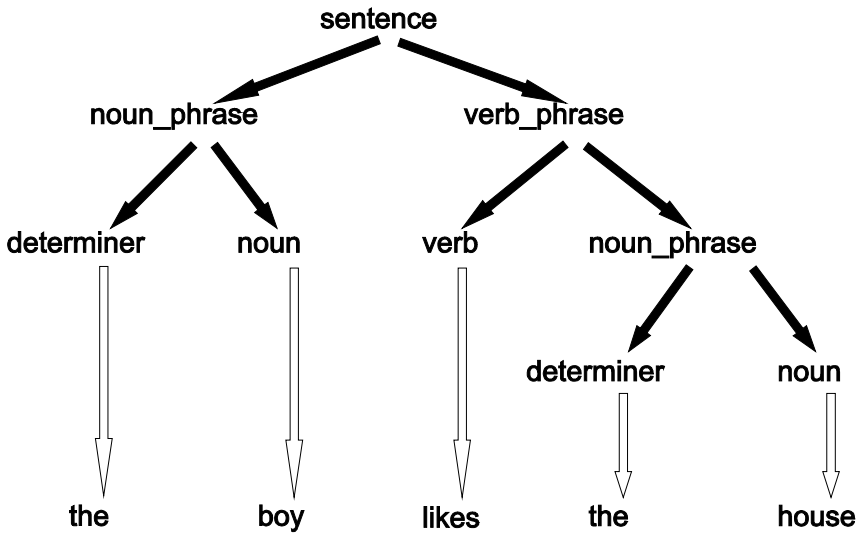
Με τον ίδιο τρόπο μπορούμε να προσθέσουμε και άλλα arguments, για να εκφράσουμε γένος, πτώση, χρόνο κλπ. Όμως υπάρχει και μια πιό εντυπωσιακή χρήση των arguments. Μπορούν να κρατήσουν ολόκληρη τη συντακτική δομή της πρότασης (το ***συντακτικό δέντρο***, όπως το λέμε), με μια τεχνική που λέγεται ***διπλασιασμός των non-terminals***:

```
sentence(sentence(N,V))          --> noun_phrase(N),verb_phrase(V).
noun_phrase(noun_phrase(D,N)) --> determiner(D), noun(N).
verb_phrase(verb_phrase(V))     --> verb(V).
verb_phrase(verb_phrase(V,N)) --> verb(V), noun_phrase(N).
determiner(determiner(the))     --> [the].
noun(noun(boy))                 --> [boy].
noun(noun(house))               --> [house].
verb(verb(likes))               --> [likes].
```

Το non-terminal στο head κάθε κανόνα παίρνει ως argument το όνομά του με παραμέτρους τις συντακτικές δομές καθενός από τους όρους του body του. Το αποτέλεσμα είναι εντυπωσιακό:

```
?- phrase(sentence(X),[the,boy,likes,the,house]).
X = sentence(noun_phrase(determiner(the), noun(boy)),
verb_phrase(verb(likes), noun_phrase(determiner(the),
noun(house)))) ;
No
```

Εκ πρώτης όψεως η τιμή του X είναι ένα compound object αλλά δεν μοιάζει με δέντρο. Όμως αν το αναλύσουμε σχεδιάζοντας βέλη από τον κάθε functor προς τα ορίσματα που περιέχει, τότε καταλήγουμε στο παρακάτω σχήμα που αποτελεί το δέντρο της συντακτικής ανάλυσης της πρότασης:



21. Χρήση και Προγραμματιστικό Στυλ στην Prolog

Τώρα πια μπορούμε να πούμε ότι γνωρίσαμε αρκετά καλά τη γλώσσα Prolog. Είδαμε τις ιδιομορφίες της αλλά και τις αρετές της. Βέβαια χρειάζεται αρκετή εμπειρία για να μάθει κανείς να την χρησιμοποιεί σωστά. Ειδικά αν ο προγραμματιστής έχει συνηθίσει σε άλλες "διαδικαστικές" γλώσσες, είναι κάτι παραπάνω από βέβαιο ότι στα πρώτα του προγράμματα θα προσπαθεί να εξομοιώσει γνώριμές του αλγοριθμικές διαδικασίες, αντί να ορίσει απλώς το πρόβλημα και να αφήσει τη μηχανή αναζήτησης της γλώσσας να κάνει τα υπόλοιπα. Πολύ αργότερα, όταν αποκτήσει την εμπειρία στο σωστό προγραμματισμό στην Prolog, και επιχειρήσει να διορθώσει παλιά του προγράμματα, θα διαπιστώσει ότι η έκταση του κώδικά του τείνει να συρρικνώνεται κατά πολύ. Είναι χαρακτηριστικό της Prolog ότι ο πηγαίος κώδικας που γράφουμε μπορεί να γίνει είναι πολύ πιο "συμπυκνωμένος" και περιεκτικός από οποιαδήποτε άλλη γλώσσα. Κι όσο περισσότερο χρόνο σκέψης επενδύουμε στην ανάπτυξη ενός τμήματος κώδικα, τόσο θα βρίσκουμε τρόπους να γράφουμε λιγώτερα και να εννοούμε περισσότερα. Κυκλοφορεί ως ανέκδοτο μεταξύ των προγραμματιστών της Prolog ότι “οποιοδήποτε πρόγραμμα μπορεί να γραφτεί με 20 γραμμές κώδικα Prolog”. Είναι σίγουρα υπερβολή, όμως έχει μια μεγάλη δόση αλήθειας. Αν σκεφτούμε ότι η δομή της Prolog ευνοεί την top-down ανάπτυξη κώδικα και ότι μπορούμε να εκφράσουμε στην top-level συνάρτηση όλη τη λογική του προγράμματος με τέτοιο τρόπο ώστε να χειρίζεται όλες τις υπόλοιπες συναρτήσεις ως δεδομένα (data) παρά ως κώδικα, τότε ναι, έχει μια μεγάλη δόση αλήθειας.

Ως παραδείγματα περιεκτικού δηλωτικού προγραμματισμού αξίζει να δείτε τα ακόλουθα δυο προγράμματα, παρμένα από το βιβλίο *"Logic Programming and Knowledge Engineering"* του Tore Amble:

```
%  Prolog interpreter written in Prolog
:- op(1199,xfx,'<-').
prove(true).
prove((X,Y)):- prove(X), prove(Y).
prove(H):- (H<-B), prove(B).

logic_programming:- % is
          logic, % as long as we don't use the
          !, % which brings us irrevocably into
          programming. % with no way of backtracking.
```

Το πρώτο δείχνει έναν interpreter της Prolog γραμμένο στην ίδια τη γλώσσα, το δεύτερο εξηγεί τη σχέση μεταξύ λογικής, προγραμματισμού και λογικού προγραμματισμού και δηλώνει το ρόλο του cut σε αυτή τη σχέση.

Τα παραπάνω προγράμματα είναι τόσο λιτά και περιεκτικά που θα μπορούσαμε να πούμε ότι αποτελούν δείγματα "προγραμματιστικής λογοτεχνίας". Μοιάζουν με κείμενα που εκτελούνται, με ποιήματα γραμμένα σε μια τεχνητή γλώσσα. Σύμφωνοι, δεν έχουν καμμία ιδιαίτερη πρακτική χρησιμότητα (εκτός ίσως από το να μας διδάξουν κάποια πράγματα για τη γλώσσα) αλλά αυτό δεν ισχύει κατά κανόνα στα έργα τέχνης;

Βέβαια οφείλουμε να τονίσουμε ότι ο εκτελέσιμος κώδικας που παράγεται από έναν Prolog compiler δεν είναι ο βέλτιστος ούτε από άποψη ταχύτητας, ούτε από άποψη κατανάλωσης μνήμης. Για

εφαρμογές στις οποίες χρειαζεται optimization σε ταχύτητα ή μνήμη υπάρχουν καταλληλότερες γλώσσες προγραμματισμού, όπως η C. Όμως τέτοιες εφαρμογές πολύ σπάνια αγγίζουν το χώρο της Τεχνητής Νοημοσύνης, προβλήματα της οποίας παραδοσιακά καλείται να αντιμετωπίσει η Prolog. Και σχεδόν όλες οι μοντέρνες εκδόσεις Prolog διαθέτουν κάποιο μηχανισμό συνεργασίας με τη C, έτσι ώστε να είναι δυνατή η κλήση υποπρογραμμάτων C μέσα από την Prolog ή υποπρογραμμάτων Prolog μέσα από τη C.

Η Prolog ως γλώσσα έχει εξελιχθεί πάρα πολύ από την δεκαετία του '70 μέχρι σήμερα, χωρίς όμως να χάσει τη φιλοσοφία της. Μοντέρνες εκδόσεις όπως η SWI-Prolog περιλαμβάνουν πολλά νέα χαρακτηριστικά όπως modular programming, multi-threaded εκτελέσιμο κώδικα, constraint programming, γραφικό περιβάλλον επικοινωνίας με το χρήστη, καθώς και πληθώρα βιβλιοθηκών με έτοιμες συναρτήσεις.

Φτάνοντας στο τέλος αυτού του εντύπου έχετε μάθει αρκετά πράγματα για την Prolog ώστε να είστε πλέον σε θέση να φτιάχνετε τα δικά σας προγράμματα με τη βοήθεια του manual του compiler που χρησιμοποιείτε. Για όσουν θα ήθελαν περισσότερες λεπτομέρειες για τον τρόπο λειτουργίας της γλώσσας συνιστάται η ανάγνωση του βιβλίου "*Programming in Prolog*" των Clocksin & Mellish. Για την αντιμετώπιση προβλημάτων Τεχνητής Νοημοσύνης με Prolog, συνιστάται το "*Prolog Programming for A.I.*" του I. Bratko. Δείτε σχετικά την ενότητα με τη Βιβλιογραφία, στο τέλος αυτού του εντύπου. Επίσης, αξίζει να επισκεφθείτε τα sites: http://www.freeprogrammingresources.com/prologbook.html και http://www.amzi.com/articles/prolog_books_tutorials.htm

22. Λυμένες Ασκήσεις

ΑΣΚΗΣΗ -1-
Τι θα τυπώσει το παρακάτω πρόγραμμα;

```
what.
what :- what.

α)   :- what, write('ONE'), nl, fail.
```

Αν αλλάζαμε το goal σε:

```
β)   :- !, what, write('TWO'), nl, fail.
γ)   :- what, !, write('THREE'), nl, fail.
δ)   :- what, write('FOUR'), nl, !, fail.
ε)   :- what, write('FIVE'), nl, fail, !.
```

τι θα έγραφε σε κάθε μια περίπτωση;

```
α)   ONE
     ONE
     ONE
      :
      :

β)   TWO
     TWO
     TWO
      :
      :

γ)   THREE
     No


δ)   FOUR
     No


ε)   FIVE
     FIVE
     FIVE
       :
       :
```

ΑΣΚΗΣΗ -2-

Να φτιαχτεί σε Prolog μια συνάρτηση, έστω addlist(LIST1,LIST2,RESULTS), η οποία θα δέχεται δυο λίστες ακεραίων (LIST1 και LIST2) και θα επιστρέφει σε μια λίστα (RESULTS) τα αθροίσματά τους ανά ζεύγη. Θεωρήστε ότι οι LIST1 και LIST2 είναι ισομεγέθεις.

Πχ: `?- addlist([4,1,3,2],[5,6,1,0],X).`
`X = [9,7,4,2]`

```
addlist([],[],[]).
addlist([H1|T1],[H2|T2],[H3|T3]):- H3 is H1+H2,
                    addlist(T1,T2,T3).
```

ΑΣΚΗΣΗ -3-

Θεωρήστε το παρακάτω πρόγραμμα Prolog:

```
f(X,Y,_):- assert(g(X,Y)), fail.
f(_,_,Y):- retract(g(X,Z)), Y is X+Z.
```

Τι θα απαντήσει η Prolog στην ερώτηση:

```
?- f(11,8,X).
```

Εξηγήστε περιληπτικά τη διαδικασία (5-10 σειρές).

```
X = 19
```

Η κλήση f(11,8,X) προκαλεί τη δημιουργία του γεγονότος g(11,8) μέσω της assert. Οταν ο πρώτος κανόνας αποτύχει, ενεργοποιείται ο δεύτερος που σβήνει το g(11,8) από τον Prolog Workspace. Αλλά τώρα το X έχει την τιμή 11 και το Z την τιμή 8, έτσι Y=19 που είναι το αποτέλεσμα.

ΑΣΚΗΣΗ -4-

Τι θα γράψει το παρακάτω πρόγραμμα;

```
f(1,one).
f(s(1),two).
f(s(s(1)),three).
f(s(s(s(X))),N) :- f(X,N).

:- f(s(1),A), write(A), nl,
   f(s(s(s(s(s(s(1)))))),B), write(B), nl,
   f(C,three), write(C), nl, A=B.
```

Στην κλήση f(s(1),A) το A ενοποιείται με το two.
Η write(A) γράφει one και η nl αλλάζει σειρά.

Για το f(s(s(s(s(s(s(1)))))),B), δεν έχουμε fact με 6 s στη σειρά, αλλά έχουμε κανόνα με πρώτο όρισμα s(s(s(X))). Αρα, το X ενοποιείται με τα υπόλοιπα 3 s: s(s(s(1))) και το N με το B. Αλλά δεν έχει υπολογιστεί ακόμα η λογική του τιμή. Σύμφωνα με τον κανόνα, η λογική τιμή του

f(s(s(s(s(s(1))))),B) είναι ίδια με τη λογική τιμή του f(s(s(s(1))),B) και τα δεύτερα ορίσματα ίδια επίσης. Μένει λοιπόν να υπολογίσουμε το f(s(s(s(1))),B). Πάλι δεν έχουμε κατάλληλο fact και καταφεύγουμε στον ίδιο κανόνα: το f(s(s(s(1))),B) έχει την ίδια λογική τιμή και το ίδιο δεύτερο όρισμα με το f(1,B). Για το f(1,B) έχουμε fact και B=one.
Η write(B) γράφει το one και η nl αλλάζει σειρά.

Για το f(C,three) έχουμε C=s(s(1)) από το fact.
Η write(C) γράφει s(s(1)) και η nl αλλάζει σειρά.

Μετά συναντάμε το A=B. Επειδή το A είναι two και το B one, η κλήση two=one γίνεται fail. Ετσι η Prolog κάνει backtracking στην πλησιέστερη μη ντετερμινιστική κλήση. Αυτή είναι η f(C,three), η οποία μπορεί να δώσει και δεύτερη απάντηση: Προηγουμένως είχε απαντήσει C=s(s(1)) από το fact. Ομως μπορεί να ικανοποιηθεί και με τον κανόνα, με N=three και C=s(s(s(X))). Μένει τώρα να βρεθεί η τιμή του X. Από το body του κανόνα, έχουμε κλήση της f(X,three), η οποία δίνει (από το fact) X=s(s(1)). Και έτσι το C είναι s(s(s(s(1))))).

Η write(C) γράφει την έκφραση και το A=B προκαλεί πάλι fail: Πάλι backtracking και η κλήση που είδαμε προηγουμένως, f(X,three), μπορεί επίσης να ικανοποιηθεί και από τον κανόνα.
Και η διαδικασία επαναλαμβάνεται.

Τελικά:

```
two
one
s(s(1))
s(s(s(s(1))))
s(s(s(s(s(s(s(1)))))))
      :
      :
      :
```

<u>ΑΣΚΗΣΗ -5-</u>
Είναι γνωστή στην Prolog η έννοια του αναδρομικού κανόνα (ένας κανόνας που καλεί τον εαυτό του, π.χ. "p :- p."). Η αναδρομικότητα, όμως, μπορεί να εκφράζεται και αλλιώς: δυο κανόνες που καλούν ο ένας τον άλλον, όπως στο:

```
p :- q.
q :- p.
```

Μια τέτοια περίπτωση φαίνεται στο παρακάτω πρόγραμμα:

```
a(32,20):-!.
a(X,Y):- f1(X,Z), b(Z,Y).
b(X,Y):- f2(X,Z), a(Z,Y).
f1(N,NewN):- NewN is N+4.
f2(N,NewN):- NewN is N*2.
```

Α) Εξηγήστε τι απαντάει η Prolog στα queries:

```
α)   ?- a(2,Reply).
β)   ?- a(12,Reply).
γ)   ?- a(22,Reply).
```

Β) Βρείτε μια αρνητική (ακέραια) τιμή για το χ, ώστε το query:

```
?- a(χ,20).
```

να είναι TRUE (το χ εδώ δεν είναι μεταβλητή ή atom της Prolog, αλλά παριστάνει τον ζητούμενο αρνητικό ακέραιο).

Α)

α) Οι διαδοχικές κλήσεις έχουν ως εξής:

a(2,Reply)
f1(2,6), b(6,Reply)
f2(6,12), a(12,Reply)
f1(12,16), b(16,Reply)
f2(16,32), a(32,Reply)
Reply = 20

β) Η a(12,Reply) είναι μια ενδιάμεση κλήση από αυτές που είδαμε προηγουμένως και οδηγεί στο Reply=20.

γ) a(22,Reply)
f1(22,26), b(26,Reply)
f2(26,52), a(52,Reply)
Παρατηρούμε ότι το πρώτο όρισμα της a/2 έχει ξεπεράσει την τιμή 32 που έχει η τερματική σχέση a(32,20). Επιπλέον, δεν υπάρχει περίπτωση να ξαναμειωθεί, μια που οι f1/2 και f2/2 είναι αύξουσες συναρτήσεις. Ετσι, η Prolog θα πέσει σε λογικό loop.

Β) Είδαμε ότι η σχέση γίνεται TRUE για τους θετικούς ακεραίους 2,12,32. Παρατηρούμε ότι η ακολουθία μπορεί να περιγραφθεί από τη σχέση: $x_{i+1} = 2x_i + 8$

Για να βρούμε τον ακέραιο που προηγείται του 2, έχουμε:

$$2 - 2x + 8 \Leftrightarrow x = \frac{2-8}{2} \Leftrightarrow x = -3$$

ΑΣΚΗΣΗ -6-

Φτιάξτε μια συνάρτηση Prolog που θα παίρνει μια λίστα και θα επιστρέφει μια δεύτερη λίστα με στοιχεία τα atoms που περιέχονται στην πρώτη λίστα, σε οποιοδήποτε βάθος.

Π.χ:

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

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

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

Ενας από τους πολλούς τρόπους υλοποίησης της ζητούμενης συνάρτησης είναι ο ακόλουθος:

```
function([],[]):- !.
function([H|T],[H|R]):- \+ lst(H), !, function(T,R).
function([H|T],L):- function(H,L1), function(T,L2),
                append(L1,L2,T).
```

Οπου η append/3 ορίζεται ως:

```
append([],L,L).
append([H|F],B,[H|L]):- append(F,B,L).
```

ΑΣΚΗΣΗ -7-

Εστω η συνάρτηση:

```
function([],[]):- !.
function([H|T],[H|R]):- \+ member(H,T), !, function(T,R).
function([_|T],R):- function(T,R).
```

(α) Γράψτε τι απάντηση θα δώσει η Prolog στην ερώτηση:

```
?- function([8,2,1,7,1,4,5,5,8,9],X).
```

(β) Ορίστε κατάλληλο τελεστή ώστε η ερώτηση:

```
?- [8,2,1,7,1,4,5,5,8,9] gives X.
```

να δίνει την ίδια απάντηση με το (α)

(α) `X = [2,7,4,5,8,9]`

Η 2η και η 3η σχέση δηλώνουν ότι το 2ο argument της function/2 θα κρατήσει το head του 1ου argument, μόνο αν αυτό δεν περιέχεται στο tail του 1ου argument. Επομένως η λειτουργία της function/2 είναι να αφαιρεί από μια λίστα τα στοιχεία που εμφανίζονται περισσότερες από μια φορές, αφήνοντας μόνο την τελευταία εμφάνισή τους στη λίστα.

Ετσι, το πρώτο 8 θα φύγει γιατί υπάρχει κι άλλο 8 μετά από αυτό. Το 2 δεν φεύγει γιατί είναι μοναδικό, το 1 φεύγει γιατί υπάρχει κι άλλο, το 7 μένει, το 1 που ακολουθεί μένει επίσης γιατί δεν υπάρχει άλλο 1 μετά από αυτό, το 4 μένει, το πρώτο 5 φεύγει και μένει το δεύτερο και τέλος τα 8 και 9 μένουν. Αποτέλεσμα: [2,7,4,5,8,9].

(β)
```
:- op(700,xfx,gives).          % εδώ ορίζεται ο τελεστής
A gives B :- function(A,B).    % και εδώ η λειτουργία του
```

ΑΣΚΗΣΗ -8-

Ορίζουμε μια συνάρτηση με όνομα *order*, σύμφωνα με τις παρακάτω σχέσεις:

```
order([],[]).
```

```
order([H|T],[H|R]):- order(T,L), reverse(L,R).
```

Η συνάρτηση *reverse* αντιστρέφει λίστες, π.χ:

```
?- reverse([1,2,3,4],X).
X = [4,3,2,1]
```

(α) Τι τιμή θα πάρει το X αν εκτελεστεί η κλήση:

```
?- order([1,2,3,4,5,6],X).
```

(β) Τι τιμή φαντάζεστε ότι θα έπαιρνε το X αν στην ίδια κλήση, αντί για τη λίστα [1,2,3,4,5,6] είχαμε μια λίστα με τους ακεραίους από το 1 ως το 100 διατεταγμένους κατά φθίνουσα σειρά;

(α)
Η αναδρομική σχέση καλεί τον εαυτό της πριν κληθεί η reverse/2. Αυτό σημαίνει ότι για να κληθεί η reverse/2 θα πρέπει πρώτα να τερματίσει επιτυχώς κάποια κλήση της order/2.

Με άλλα λόγια, ένα trace θα έδινε κάτι τέτοιο:

```
?- trace, order([1,2,3,4,5,6],X).
   Call: (8) order([1, 2, 3, 4, 5, 6], _G381)        Η order/2 με κάθε κλήση
   Call: (9) order([2, 3, 4, 5, 6], _L163)      διαχωρίζει το head από τη λίστα
   Call: (10) order([3, 4, 5, 6], _L184)
   Call: (11) order([4, 5, 6], _L205)
   Call: (12) order([5, 6], _L226)
   Call: (13) order([6], _L247)
   Call: (14) order([], _L268)                          ...ώσπου η λίστα να γίνει κενή.
   Exit: (14) order([], [])      Τότε η order/2 γίνεται true από την τερματική σχέση
   Call: (14) lists:reverse([], _G482)          και εκτελείται η πρώτη reverse/2.
   Exit: (14) lists:reverse([], [])     Στην αντεστραμμένη λίστα προστίθεται το
   Exit: (13) order([6], [6])              head και γίνεται true η "parent" order/2.
   Call: (13) lists:reverse([6], _G479)          Αλλά η επαλήθευση της order/2
   Exit: (13) lists:reverse([6], [6])          προκαλεί την κλήση της reverse/2
   Exit: (12) order([5, 6], [5, 6])                του προηγούμενου κύκλου,
   Call: (12) lists:reverse([5, 6], _G476)                          κ.ο.κ.
   Exit: (12) lists:reverse([5, 6], [6, 5])
   Exit: (11) order([4, 5, 6], [4, 6, 5])
   Call: (11) lists:reverse([4, 6, 5], _G473)
   Exit: (11) lists:reverse([4, 6, 5], [5, 6, 4])
   Exit: (10) order([3, 4, 5, 6], [3, 5, 6, 4])
   Call: (10) lists:reverse([3, 5, 6, 4], _G470)
   Exit: (10) lists:reverse([3, 5, 6, 4], [4, 6, 5, 3])
   Exit: (9) order([2, 3, 4, 5, 6], [2, 4, 6, 5, 3])          ...ώσπου να
   Call: (9) lists:reverse([2, 4, 6, 5, 3], _G467)       επαληθευτεί και η
   Exit: (9) lists:reverse([2, 4, 6, 5, 3], [3, 5, 6, 4, 2])  η αρχική
```

```
   Exit: (8) order([1, 2, 3, 4, 5, 6], [1, 3, 5, 6, 4, 2])      κλήση.
```

Το output είναι:

```
X = [1, 3, 5, 6, 4, 2]
```

Παρατηρούμε ότι η order/2 διαχώρισε τους άρτιους από τους περιττούς της αρχικής λίστας τοποθετώντας τους στο τέλος και με αντίθετη (φθίνουσα) διάταξη.

(β)
Χρησιμοποιώντας την προηγούμενη παρατήρηση, μπορούμε να πούμε με βεβαιότητα ότι η κλήση:

```
?- order([100,99,98,......,3,2,1],X).
```

θα δώσει:

```
X = [100,98,96,94,....,4,2,1,3,....,93,95,97,99]
```

ΑΣΚΗΣΗ -9-
ΓΕΝΝΗΤΡΙΑ ΣΕΙΡΑΣ ΑΡΙΘΜΩΝ
Να φτιαχτεί σε Prolog μια συνάρτηση, έστω count(FROM,TO,STEP), η οποία θα τυπώνει φυσικούς αριθμούς από τον FROM μέχρι τον TO με βήμα STEP. Πχ. η εντολή count(5,12,3) θα τυπώνει: 5,8,11. Αν ο αριθμός στη μεταβλητή TO είναι μικρότερος από τον FROM, η συνάρτηση θα μετράει αντίστροφα. Πχ. η count(17,11,2) θα δίνει 17,15,13,11.

```
count(FROM,TO,STEP):- FROM=<TO, write(FROM), nl,
              NEW_FROM is FROM+STEP,
              again_if_lower(NEW_FROM,TO,STEP).
count(FROM,TO,STEP):- FROM>=TO, write(FROM), nl,
              NEW_FROM is FROM-STEP,
              again_if_higher(NEW_FROM,TO,STEP).

again_if_lower(FROM,TO,STEP):- FROM=<TO, !, count(FROM,TO,STEP).
again_if_lower(_,_,_).

again_if_higher(FROM,TO,STEP):- FROM>=TO,!, count(FROM,TO,STEP).
again_if_higher(_,_,_).
```

Βιβλιογραφία

[1] **Amble T.** "*Logic Programming and Knowledge Engineering*", Addison-Wesley Publishing Company, Inc., 1987.

[2] **Bratko I.** "*Prolog Programming for Artificial Intelligence*" 3rd edition, Addison-Wesley Publishing Company, Inc., 2000.

[3] **Clocksin W.F. & Mellish C.S.** "*Programming in Prolog – Using the ISO Standard*" 5th edition, Springer-Verlag 2003.

[4] **Deransart P., Ed-Dbali A., Cervoni L.** "*Prolog: The Standard - Reference Manual*", Springer-Verlag, 1996.

[5] **Kowalski R.** "*Logic for Problem Solving*", Elsevier North-Holland, 1984.

[6] **Logic Programming Associates Ltd.** "*LPA Prolog Professional (Programming and Language Reference Manual)*", 1990.

[7] **Walker A., McCord M., Sowa J.F. & Wilson W.G.** "*Knowledge Systems and Prolog*", Addison-Wesley Publishing Company, Inc., 1987.

[8] **Wielemaker J.** "*SWI-Prolog Reference Manual*" (Updated for version 5.6.7), http://www.swi-prolog.org, University of Amsterdam, March 2006.