Λογικός και Συναρτησιακός Προγραμματισμός

Παναγιώτης Σταματόπουλος

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά Συγγράμματα και Βοηθήματα
www.kallipos.gr

HEALLINK
Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών

Ευρωπαϊκή Ένωση
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013

ΠΑΝΑΓΙΩΤΗΣ ΣΤΑΜΑΤΟΠΟΥΛΟΣ
ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΛΟΓΙΚΟΣ ΚΑΙ ΣΥΝΑΡΤΗΣΙΑΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Λογικός και Συναρτησιακός Προγραμματισμός

Συγγραφή

Παναγιώτης Σταματόπουλος

Κριτικός αναγνώστης

Παναγιώτης Ροντογιάννης

Συντελεστές έκδοσης

ΓΛΩΣΣΙΚΗ ΕΠΙΜΕΛΕΙΑ: Φωτεινή Ξιφαρά

ΓΡΑΦΙΣΤΙΚΗ ΕΠΙΜΕΛΕΙΑ: Σπυρίδων Παπαβασιλείου

ΤΕΧΝΙΚΗ ΕΠΕΞΕΡΓΑΣΙΑ: Σπυρίδων Παπαβασιλείου

ΣΥΝΔΕΣΜΟΣ ΕΛΛΗΝΙΚΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΒΙΒΛΙΟΘΗΚΩΝ

Εθνικό Μετσόβιο Πολυτεχνείο

Ηρώων Πολυτεχνείου 9, 15780 Ζωγράφου

www.kallipos.gr

ISBN: 978-960-603-335-3

Περιεχόμενα

1 Διαδικαστικός και δηλωτικός προγραμματισμός **9**
1.1 Αλγόριθμοι και αξιώματα 9

2 Λογικός προγραμματισμός – Υπολογισμός με λογική **19**
2.1 Ιστορική αναδρομή 19
2.2 Λογική και κανόνας modus ponens 21

3 Η γλώσσα λογικού προγραμματισμού Prolog – Τα βασικά **27**
3.1 Γεγονότα, κανόνες και ερωτήσεις 27
3.2 Αναδρομή 35
3.3 Διαδικασία απόδειξης – Οπισθοδρόμηση 36
3.4 Εφαρμογές της Prolog 41
3.4.1 Έμπειρα συστήματα 41
3.4.2 Κατανόηση φυσικής γλώσσας 42
3.4.3 Προβλήματα αναζήτησης 42
3.4.4 Απόδειξη θεωρημάτων 43
3.4.5 Συμβολική επεξεργασία 43
3.5 Υλοποίηση συστημάτων Prolog 44

4 Δομές στην Prolog **53**
4.1 Όροι 53
4.2 Λίστες 56

5 Ενσωματωμένες δυνατότητες και επεκτάσεις της Prolog **69**
5.1 Τελεστές 69
5.2 Αριθμητική 73
5.3 Έλεγχος οπισθοδρόμησης 77
5.4 Άρνηση 80
5.5 Ενσωματωμένα κατηγορήματα 84

6 Λογικός προγραμματισμός με περιορισμούς **109**
6.1 Προβλήματα ικανοποίησης περιορισμών 109
6.2 Η βιβλιοθήκη `ic` της ECL^{i}PS^e 115
6.3 Η βιβλιοθήκη `branch_and_bound` της ECL^{i}PS^e 119

7 Λογική πρώτης τάξης **131**
7.1 Προτάσεις Horn 131
7.2 Ενοποίηση 137
7.3 Αρχή της ανάλυσης 139

8 Σημασιολογία λογικών προγραμμάτων **145**
8.1 Μοντελοθεωρητική σημασιολογία 145
8.2 Σημασιολογία σταθερού σημείου 151
8.3 Λειτουργική σημασιολογία 153

9 Συναρτησιακός προγραμματισμός – Υπολογισμός με συναρτήσεις **165**
9.1 Ιστορική αναδρομή και γενικά 165

10 Η γλώσσα συναρτησιακού προγραμματισμού Haskell – Τα βασικά **171**
10.1 Ορισμός συναρτήσεων και υπολογισμός εκφράσεων 171
10.2 Πλειάδες και λίστες 175
10.3 Πολυμορφισμός 178
10.4 Το πρελούδιο 180
10.5 Εφαρμογές της Haskell 181

11 Εκφραστικές δυνατότητες της Haskell **191**
11.1 Περιφραστικές λίστες 191
11.2 Συναρτήσεις ανώτερης τάξης 194
11.3 Σύνθεση συναρτήσεων 196
11.4 Συναρτήσεις πολλών ορισμάτων – Currying 198
11.5 Αποδείξεις ιδιοτήτων 200
11.6 Άλλα χαρακτηριστικά της Haskell 204

12 Τα εσωτερικά του συναρτησιακού προγραμματισμού **213**
12.1 Λάμδα λογισμός 213
12.2 Συνδυαστές 219
12.3 Σειρές αναγωγής 222
12.4 Αναγωγή γράφων 226

Εισαγωγή

Αντικείμενο του παρόντος συγγράμματος είναι δύο προγραμματιστικές φιλοσοφίες αρκετά διαφορετικές από αυτήν του **διαδικαστικού προγραμματισμού**, ο **λογικός προγραμματισμός** και ο **συναρτησιακός προγραμματισμός**, οι οποίες, παρότι αρκετά διαφορετικές μεταξύ τους, υποστηρίζουν έναν κοινό τρόπο προγραμματισμού, τον **δηλωτικό προγραμματισμό**. Οι μεθοδολογίες αυτές αντιμετώπισης προβλημάτων εφαρμόζονται στην πράξη μέσω συγκεκριμένων γλωσσών προγραμματισμού, αλλά ταυτόχρονα έχουν και αυστηρή θεωρητική τεκμηρίωση, αποτελώντας σημαντικό τμήμα της Επιστήμης των Υπολογιστών.

Στο Κεφάλαιο 1 μελετάται η έννοια του δηλωτικού προγραμματισμού και αντιδιαστέλλεται με αυτήν του διαδικαστικού προγραμματισμού, κυρίως μέσω παραδειγμάτων επίλυσης συγκεκριμένων προβλημάτων. Στο Κεφάλαιο 2 γίνεται αναφορά στη φιλοσοφία του λογικού προγραμματισμού και στα Κεφάλαια 3, 4 και 5 παρουσιάζεται η γλώσσα λογικού προγραμματισμού **Prolog**, ως τυπικός εκπρόσωπος της φιλοσοφίας αυτής. Στο Κεφάλαιο 3 εξετάζονται με συνοπτικό τρόπο θέματα υλοποίησης συστημάτων Prolog. Στο Κεφάλαιο 6 εισάγεται η έννοια των περιορισμών στον λογικό προγραμματισμό και αναφέρεται ο τρόπος με τον οποίο η συγκεκριμένη επέκταση μπορεί να βοηθήσει στην αποτελεσματικότερη αντιμετώπιση προβλημάτων συνδυαστικής αναζήτησης, δηλαδή προβλημάτων ικανοποίησης περιορισμών. Στο Κεφάλαιο 7 δίνονται στοιχεία από τη λογική πρώτης τάξης, που είναι το μαθηματικό υπόβαθρο του λογικού προγραμματισμού, και στο Κεφάλαιο 8 παρουσιάζονται, σε σχετικά υψηλό επίπεδο, οι διάφορες προσεγγίσεις μελέτης της σημασίας των λογικών προγραμμάτων. Στο Κεφάλαιο 9 επιχειρείται μια εισαγωγή στη δεύτερη δηλωτική μεθοδολογία προγραμματισμού, αυτή του συναρτησιακού προγραμματισμού, και στα Κεφάλαια 10 και 11 παρουσιάζεται η **Haskell**, μια αντιπροσωπευτική γλώσσα συναρτησιακού προγραμματισμού. Τέλος, στο Κεφάλαιο 12 εξετάζονται θέματα σχετικά με το θεωρητικό υπόβαθρο του συναρτησιακού προγραμματισμού, όπως ο λάμδα λογισμός και οι συνδυαστές, καθώς και άλλα που αναφέρονται στις τεχνικές υλοποίησης των γλωσσών συναρτησιακού προγραμματισμού, όπως οι σειρές αναγωγής και η αναγωγή γράφων.

Κεφάλαιο 1
Διαδικαστικός και δηλωτικός προγραμματισμός

Σύνοψη

Στο κεφάλαιο αυτό γίνεται συγκριτική παρουσίαση, κυρίως μέσω απλών παραδειγμάτων, του διαδικαστικού και του δηλωτικού προγραμματισμού, δύο μεθοδολογιών για την αντιμετώπιση προβλημάτων στην Επιστήμη των Υπολογιστών.

Με τον διαδικαστικό προγραμματισμό μπορούμε να επιλύσουμε ένα πρόβλημα διατυπώνοντας τα βήματα ενός αλγορίθμου, δηλαδή ακολουθώντας μια διαδικασία που περιγράφει πώς πρέπει να λυθεί το πρόβλημα και να υλοποιηθεί ο αλγόριθμος, με τη βοήθεια μιας κατάλληλης γλώσσας διαδικαστικού προγραμματισμού.

Αντίθετα, στον δηλωτικό προγραμματισμό διατυπώνουμε αξιώματα που περιγράφουν τι ισχύει στον κόσμο του προβλήματος, τα οποία εκφράζουμε με τη βοήθεια μιας κατάλληλης γλώσσας δηλωτικού προγραμματισμού.

Πολλές φορές, η επίλυση ενός προβλήματος με αλγοριθμικό τρόπο οδηγεί σε προγράμματα πολύπλοκα, δυσανάγνωστα και, συνεπώς, δύσκολα συντηρήσιμα. Όταν έχουμε ένα πρόβλημα που ταιριάζει στη δηλωτική φιλοσοφία, δηλαδή διέπεται από απλά αξιώματα, είναι προτιμότερο να επιλέγουμε αυτόν τον τρόπο για την αντιμετώπισή του. Η μόνη επιφύλαξη που μπορούμε, ίσως, να διατυπώσουμε αφορά την απόδοσή του. Σε γενικές γραμμές, τα συστήματα δηλωτικού προγραμματισμού είναι λιγότερο αποδοτικά από αυτά του διαδικαστικού προγραμματισμού, χωρίς κάτι τέτοιο να σημαίνει ότι δεν υπάρχουν αρκετές δηλωτικές γλώσσες πολύ πιο «γρήγορες» από πολλές διαδικαστικές. Έτσι, αν μας ενδιαφέρει η απόδοση του προγράμματος που θα γράψουμε, πρέπει να ζυγίσουμε προσεκτικά όλες τις εμπλεκόμενες παραμέτρους, για να κάνουμε τη σωστή επιλογή.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει στοιχειώδεις γνώσεις απλών αλγορίθμων και διατύπωσής τους σε ψευδογλώσσα.

1.1 Αλγόριθμοι και αξιώματα

Στην Επιστήμη των Υπολογιστών, γνωρίζουμε την έννοια του αλγορίθμου [1]. Χρησιμοποιώντας μια σχετικά απλουστευμένη διατύπωση, μπορούμε να δώσουμε τον εξής ορισμό:

Ένας αλγόριθμος περιγράφει, μέσω μιας ακολουθίας στοιχειωδών εκτελέσιμων λειτουργιών, τη μέθοδο επίλυσης ενός συγκεκριμένου προβλήματος.

Με άλλα λόγια, ένας αλγόριθμος περιγράφει **πώς** λύνεται ένα πρόβλημα, βήμα προς βήμα, έτσι ώστε η επίλυσή του να είναι εύκολα υλοποιήσιμη, με τη βοήθεια μιας γλώσσας

προγραμματισμού. Υπάρχουν διάφορες γλώσσες προγραμματισμού που υποστηρίζουν τη διατύπωση αλγορίθμων για την επίλυση προβλημάτων, όπως η Basic, η Fortran, η Pascal, η C, η C++, η Java κ.ά. [2]. Στις γλώσσες αυτές, που ονομάζονται διαδικαστικές, βρίσκουμε εντολές που αντιστοιχούν άμεσα στις στοιχειώδεις λειτουργίες ενός αλγορίθμου, όπως οι εντολές ανάθεσης τιμών σε μεταβλητές μέσω μαθηματικών εκφράσεων, οι εντολές επανάληψης και οι εντολές ελέγχου της ροής μιας διαδικασίας. Με αυτόν τον τρόπο, εισάγουμε την έννοια του **διαδικαστικού προγραμματισμού** ως εξής:

Διαδικαστικός προγραμματισμός ονομάζεται η υλοποίηση ενός αλγορίθμου σε μια διαδικαστική γλώσσα προγραμματισμού.

Παράδειγμα 1.1

Έστω ότι ενδιαφερόμαστε αρχικά να διατυπώσουμε έναν αλγόριθμο και στη συνέχεια να τον υλοποιήσουμε σε κάποια γλώσσα διαδικαστικού προγραμματισμού, με τον οποίο να μπορούμε να υπολογίσουμε το πλήθος των στοιχείων μιας απλά συνδεδεμένης λίστας, δηλαδή το μήκος της [3]. Ας θυμηθούμε πρώτα ότι μια τέτοια λίστα δεν είναι τίποτε άλλο παρά μια ακολουθία από στοιχεία, με την ταυτότητά της να αντιστοιχεί στη θέση/διεύθυνση του πρώτου της στοιχείου, που λέγεται **κεφαλή** της λίστας. Η κεφαλή «δείχνει», με κάποιον τρόπο, στο επόμενο στοιχείο της λίστας ή, ισοδύναμα, στη λίστα που αρχίζει από το επόμενο της κεφαλής, η οποία ονομάζεται **ουρά** της αρχικής λίστας. Αυτό ισχύει για κάθε στοιχείο, εκτός φυσικά από το τελευταίο, το οποίο δεν «δείχνει» πουθενά ή, αλλιώς, «δείχνει» στην **κενή λίστα**. Ένας αλγόριθμος υπολογισμού του μήκους μιας λίστας, εκφρασμένος σε μια αυθαίρετη ψευδογλώσσα, θα μπορούσε να είναι ο εξής:

Υπολογισμός μήκους λίστας **list**
Βήμα 1: *Θέστε στο μετρητή* **count** *την τιμή 0*
Βήμα 2: *Ενόσω η λίστα* **list** *δεν είναι κενή*
Βήμα 3: *Αυξήστε τον* **count** *κατά 1*
Βήμα 4: *Κάντε τη* **list** *να δείχνει εκεί όπου δείχνει το πρώτο της στοιχείο, δηλαδή στην ουρά της*
Βήμα 5: *Επιστρέψτε την τιμή του* **count** *ως μήκος της λίστας*

Ο προηγούμενος αλγόριθμος μπορεί πολύ εύκολα να υλοποιηθεί σε μια γλώσσα διαδικαστικού προγραμματισμού. Στη γλώσσα C, για παράδειγμα, αν είχαμε δηλώσει τη δομή:

```
struct listnode {
   int value;
   struct listnode *next;
};
```

θα γράφαμε την εξής συνάρτηση, για τον υπολογισμό του μήκους μιας λίστας:

```
int length(struct listnode *list)
{  int count = 0;
   while (list != NULL) {
      count++;
      list = list->next;
   }
   return count;
}
```

Δεν δίνονται περισσότερες επεξηγήσεις για τη λειτουργία του προηγούμενου προγράμματος, δεδομένου ότι ο αναγνώστης έχει κάποια, στοιχειώδη έστω, εξοικείωση με τη γλώσσα C [4]. □

Αντί, όμως, να περιγράψουμε έναν αλγόριθμο, ο οποίος όταν ακολουθείται βήμα προς βήμα επιλύει ένα πρόβλημα, θα μπορούσαμε, εναλλακτικά, να περιγράψουμε τα αξιώματα που διέπουν το πρόβλημα. Ένας απλός ορισμός για το αξίωμα είναι ο εξής:

Αξίωμα είναι ένας ισχυρισμός του οποίου η αλήθεια ούτε αμφισβητείται, ούτε υπόκειται σε απόδειξη.

Με τη διατύπωση αξιωμάτων, αντί να εξετάσουμε πώς λύνεται ένα πρόβλημα, απλώς δηλώνουμε τι ισχύει στον κόσμο του προβλήματος. Όταν αυτό το κάνουμε με τη βοήθεια κάποιας γλώσσας προγραμματισμού, έχουμε την έννοια του **δηλωτικού προγραμματισμού**, ως εξής:

Δηλωτικός προγραμματισμός ονομάζεται η διατύπωση αξιωμάτων που ισχύουν στον κόσμο ενός προβλήματος σε μια κατάλληλη γλώσσα προγραμματισμού.

Μεταξύ άλλων, η γλώσσα λογικού προγραμματισμού **Prolog** [5] και η γλώσσα συναρτησιακού προγραμματισμού **Haskell** [6] είναι δύο τυπικοί εκπρόσωποι της φιλοσοφίας του δηλωτικού προγραμματισμού.

Στον δηλωτικό προγραμματισμό, όπως μαρτυρά και η ονομασία του, δηλώνουμε κάτι, αντί να περιγράφουμε μια διαδικασία, όπως συμβαίνει στον διαδικαστικό προγραμματισμό. Βέβαια, δεν είναι δυνατόν να λύσουμε ένα πρόβλημα διατυπώνοντας απλώς κάποια αξιώματα. Πρέπει να φροντίσουμε να επεξεργαστούμε τα δηλωθέντα αξιώματα του προβλήματος, για να δώσουμε την επιθυμητή λύση. Ωστόσο, αυτή η επεξεργασία δεν αφορά τον προγραμματιστή, γιατί γίνεται από το σύστημα που υποστηρίζει τον προγραμματισμό αυτού του είδους, δηλαδή τον δηλωτικό προγραμματισμό. Έτσι, ο προγραμματιστής είναι αφοσιωμένος στη διατύπωση γνώσης, παρά διαδικασιών.

Παράδειγμα 1.2
Αν επιστρέψουμε στο πρόβλημα του υπολογισμού του μήκους μιας απλά συνδεδεμένης λίστας και θυμηθούμε ότι μια μη κενή λίστα έχει κεφαλή (το πρώτο της στοιχείο) και ουρά (τη λίστα που ακολουθεί το πρώτο της στοιχείο), μπορούμε να διατυπώσουμε τα εξής αξιώματα:

Αξιώματα για το μήκος της λίστας
Αξίωμα 1: *Το μήκος της κενής λίστας είναι 0.*
Αξίωμα 2: *Το μήκος μιας μη κενής λίστας είναι ίσο με το μήκος της ουράς της αυξημένο κατά 1.*

Στη γλώσσα λογικού προγραμματισμού Prolog, τα προηγούμενα αξιώματα θα μπορούσαν να διατυπωθούν ως εξής:

```
length([], 0).
length([X|L], N) :- length(L, M), N is M+1.
```

και στη γλώσσα συναρτησιακού προγραμματισμού Haskell ως εξής:

```
length []    = 0
length (a:x) = 1 + length x
```

Φυσικά, θα μπορέσουμε να κατανοήσουμε καλύτερα τα προγράμματα Prolog και Haskell σε επόμενα κεφάλαια, στα οποία δίνονται λεπτομέρειες για τον τρόπο προγραμματισμού στις γλώσσες αυτές. □

Στο Παράδειγμα 1.2 είναι σαφής η φιλοσοφία του δηλωτικού προγραμματισμού [7]. Πουθενά δεν περιγράψαμε μια διαδικασία για να βρούμε το μήκος μιας λίστας. Αντίθετα, ορίσαμε τι είναι μήκος μιας λίστας, με στόχο, όταν προκύψει κάποια στιγμή ένα πρόβλημα εύρεσης του μήκους συγκεκριμένης λίστας, να μπορέσουμε να το διατυπώσουμε σε ένα σύστημα λογικού ή συναρτησιακού προγραμματισμού, μαζί με τον κατάλληλο ορισμό, και να ζητήσουμε τον υπολογισμό της απάντησης σε αυτό. Όσον αφορά τον τρόπο με τον οποίο διατυπώνουμε προβλήματα σε κάθε περίπτωση, θα πάρουμε μια ιδέα στη συνέχεια, αλλά θα τον μελετήσουμε με περισσότερες λεπτομέρειες σε επόμενα κεφάλαια.

Ας εξετάσουμε τώρα άλλα δύο παραδείγματα, στα οποία θα είναι εμφανέστερη η αντιδιαστολή ανάμεσα στον διαδικαστικό και τον δηλωτικό προγραμματισμό.

Παράδειγμα 1.3
Έστω ότι έχουμε πληροφορίες για διάφορα άτομα και, συγκεκριμένα, για τα ονόματα των γονέων τους. Θα θέλαμε να μάθουμε αν ο παππούς του *Γιώργου* είναι αδελφός του *Νίκου*. Μια πιθανή διαδικαστική προσέγγιση για να λύσουμε αυτό το πρόβλημα είναι ο εξής αλγόριθμος:

Έλεγχος συγκεκριμένης συγγένειας μεταξύ Γιώργου και Νίκου
Βήμα 1: *Θέστε στο* **flag** *την τιμή 0*
Βήμα 2: *Θέστε στο* **x** *τη μητέρα του Γιώργου*
Βήμα 3: *Θέστε στο* **y** *τον πατέρα του* **x**
Βήμα 4: *Θέστε στο* **z** *τον πατέρα του* **y**
Βήμα 5: *Αν ο* **z** *είναι πατέρας του Νίκου, απαντήστε καταφατικά και τερματίστε*
Βήμα 6: *Θέστε στο* **z** *τη μητέρα του* **y**
Βήμα 7: *Αν η* **z** *είναι μητέρα του Νίκου, απαντήστε καταφατικά και τερματίστε*
Βήμα 8: *Αν το* **flag** *είναι 1, απαντήστε αρνητικά και τερματίστε*
Βήμα 9: *Θέστε στο* **flag** *την τιμή 1*
Βήμα 10: *Θέστε στο* **x** *τον πατέρα του Γιώργου*
Βήμα 11: *Πηγαίνετε στο βήμα 3*

Φυσικά, ο αλγόριθμος αυτός θα μπορούσε να διατυπωθεί με πιο γενικό τρόπο, αν στη θέση του *Γιώργου* και του *Νίκου* είχαμε δύο μεταβλητές **A** και **B**, και σε κάθε εκτέλεση του αλγορίθμου τα **A** και **B** ήταν δεδομένα εισόδου. Επί της ουσίας, όμως, δεν θα υπήρχε διαφορά. □

Από την άλλη πλευρά, θα μπορούσαμε να είχαμε επιλέξει μια δηλωτική αντιμετώπιση του προβλήματος του Παραδείγματος 1.3, όπως φαίνεται στη συνέχεια.

Παράδειγμα 1.4
Ο δηλωτικός τρόπος θεώρησης του προβλήματος, αν ο παππούς του *Γιώργου* είναι αδελφός του *Νίκου*, συνίσταται απλώς στη διατύπωση των αξιωμάτων τα οποία ορίζουν τις συγγένειες που μας ενδιαφέρουν:

Αξιώματα για συγγένειες

Αξίωμα 1: *Παππούς κάποιου είναι ο πατέρας του πατέρα του ή της μητέρας του.*

Αξίωμα 2: *Αδελφός κάποιου είναι ένας άνδρας με τον οποίο έχουν τον ίδιο πατέρα ή την ίδια μητέρα.*

Για άλλη μία φορά, βλέπουμε στο παράδειγμα ότι δεν περιγράφουμε πώς θα λύσουμε το πρόβλημα, αλλά δηλώνουμε τι ισχύει στον φυσικό κόσμο σχετικά με το πρόβλημα.

Αν θέλαμε να διατυπώσουμε τα προηγούμενα αξιώματα σε μια γλώσσα δηλωτικού προγραμματισμού, θα μπορούσαμε, χρησιμοποιώντας, για παράδειγμα, τη γλώσσα Prolog, να γράψουμε:

```
grandfather(X, Z) :- father(X, Y), (father(Y, Z) ; mother(Y, Z)).
brother(X, Y) :- male(X), ((father(Z, X), father(Z, Y)) ;
                           (mother(Z, X), mother(Z, Y))).
```

Υποτίθεται ότι αυτά τα αξιώματα είναι δυνατόν να χρησιμοποιηθούν από ένα σύστημα λογικού προγραμματισμού, εφόσον αυτό είναι εφοδιασμένο και με τη γνώση του κόσμου μας σχετικά με το ποιος είναι πατέρας ποιων (`father`), ποια είναι μητέρα ποιων (`mother`) και ποιοι είναι οι άνδρες (`male`). Αν υπάρχει αυτή η γνώση, τότε η επίλυση του προβλήματος, δηλαδή αν ο παππούς του *Γιώργου* (`george`) είναι αδελφός του *Νίκου* (`nick`), ανάγεται στη διατύπωση της εξής ερώτησης στο σύστημα Prolog:

```
?- grandfather(X, george), brother(X, nick).
```

Με την επεξεργασία των αξιωμάτων που του δόθηκαν, το σύστημα θα απαντήσει καταφατικά, στην περίπτωση που υπάρχει μεταξύ *Γιώργου* και *Νίκου* η συγκεκριμένη συγγένεια, αλλιώς θα απαντήσει αρνητικά. □

Από τα παραδείγματα που προηγήθηκαν, πήραμε μια ιδέα για τις διαφορές ανάμεσα στη διαδικαστική και στη δηλωτική αντιμετώπιση προβλημάτων. Στον διαδικαστικό προγραμματισμό περιγράφουμε τη διαδικασία επίλυσης ενός προβλήματος. Σε ορισμένες περιπτώσεις, όπως στο Παράδειγμα 1.3, αυτό μπορεί να οδηγήσει στη διατύπωση πολύπλοκων και δυσνόητων αλγορίθμων, με σοβαρές επιπτώσεις στον τρόπο με τον οποίο μπορούμε εύκολα να ελέγξουμε την ορθότητα μεγάλων προγραμμάτων και, φυσικά, να τα συντηρήσουμε. Αντίθετα, με τον δηλωτικό προγραμματισμό, στον οποίο διατυπώνουμε τα αξιώματα του κόσμου μας που απαιτούνται για τη λύση του προβλήματός μας, η κατάσταση είναι απλούστερη. Θυμηθείτε το Παράδειγμα 1.4 και συγκρίνετε την προσπάθειά σας να το κατανοήσετε σε σχέση με αυτήν για το Παράδειγμα 1.3. Πιστεύουμε ότι η διαφορά είναι εμφανής. Έτσι, αν ακολουθήσουμε μια δηλωτική φιλοσοφία προγραμματισμού, η παραγωγικότητά μας θα αυξηθεί, όσον αφορά τόσο το χρόνο που απαιτείται για την αντιμετώπιση ενός προβλήματος, όσο και την ποιότητα της λύσης που επιτυγχάνεται.

Βέβαια, θα αναρωτιέστε εάν μια δηλωτική αντιμετώπιση κάποιου προβλήματος είναι πάντοτε προτιμότερη από μια διαδικαστική. Η απάντηση είναι αρνητική, γιατί, αφενός, δεν είναι βέβαιο ότι για κάθε πρόβλημα υπάρχει μια απλή και κατανοητή περιγραφή των αξιωμάτων που το διέπουν και, αφετέρου, τα συστήματα δηλωτικού προγραμματισμού οδηγούν, εν γένει, σε λιγότερο αποδοτικές υλοποιήσεις από αυτές που βασίζονται σε διαδικαστικές γλώσσες προγραμματισμού. Έτσι, αν είναι πολύ κρίσιμη η απόδοση ενός προγράμματος που θέλουμε να γράψουμε, μάλλον θα πρέπει να επιλέξουμε τη διατύπωση ενός αλγορίθμου σε κάποια «γρήγορη» γλώσσα διαδικαστικού προγραμματισμού αντί της δηλωτικής

διατύπωσης αξιωμάτων. Αυτό δεν θα πρέπει να το δεχτούμε ως απαράβατο κανόνα, γιατί σήμερα πολλά συστήματα δηλωτικού προγραμματισμού είναι σε απόδοση σχεδόν εφάμιλλα των διαφόρων γλωσσών διαδικαστικού προγραμματισμού. Όσο η έρευνα στον τομέα του δηλωτικού προγραμματισμού προοδεύει, τόσο περισσότερα πλεονεκτήματα θα παρουσιάζει η πρακτική εφαρμογή του.

Θα τελειώσουμε αυτό το κεφάλαιο, στο οποίο επικεντρωθήκαμε στις διαφορές ανάμεσα στη διαδικαστική και στη δηλωτική αντιμετώπιση προβλημάτων, με μερικές ασκήσεις, με τις οποίες θα μπορέσετε να εμπεδώσετε καλύτερα το περιεχόμενό του. Ακολουθούν οι απαντήσεις στις ασκήσεις και δίνονται κάποια προβλήματα προς λύση.

Άσκηση 1.1

Έστω ότι ενδιαφερόμαστε να έχουμε έναν τρόπο για να μπορούμε να υπολογίζουμε το παραγοντικό ενός μη αρνητικού ακέραιου αριθμού. Για το σκοπό αυτό, θα μπορούσαμε να χρησιμοποιήσουμε κάποια από τις παρακάτω μεθόδους:

1. Το 0! ισούται με 1 και το n! (για $n \neq 0$) ισούται με (n-1)! $\times$ n

2. Το 0! ισούται με 1 και, για να υπολογίσουμε το n! (για $n \neq 0$), πρέπει να πολλαπλασιάσουμε όλους τους ακέραιους αριθμούς από το 1 έως το n

Ποια από τις δύο μεθόδους θα χαρακτηρίζατε διαδικαστική και ποια δηλωτική;

Άσκηση 1.2

Έστω ότι θέλουμε να κατασκευάσουμε μια διαδικασία με την οποία να αντιστρέφεται μια λίστα **list**. Μια (ημιτελής) απάντηση σε αυτό το πρόβλημα θα μπορούσε να ήταν η εξής:

Αντιστροφή λίστας **list**
Βήμα 1: *Κάντε τη* **next** *να δείχνει εκεί όπου δείχνει η* **list**
Βήμα 2: *Θέστε στην* **prev** *την κενή λίστα*
Βήμα 3: *Ενόσω η λίστα* [**X**] *δεν είναι κενή*
Βήμα 4: *Κάντε τη* **list** *να δείχνει εκεί όπου δείχνει η* **next**
Βήμα 5: *Κάνε τη* **next** *να δείχνει εκεί όπου δείχνει το πρώτο της στοιχείο*
Βήμα 6: *Κάντε το πρώτο στοιχείο της* **list** *να δείχνει* [**Y**]
Βήμα 7: *Κάντε την* **prev** *να δείχνει εκεί όπου δείχνει η* **list**
Βήμα 8: *Επιστρέψτε τη* [**Z**] *ως τη ζητούμενη αντεστραμμένη λίστα*

Ποια επιλογή από τις παρακάτω θα πρέπει να συμπληρωθεί στις θέσεις **X**, **Y** και **Z**, για να είναι σωστός ο αλγόριθμος;

X: 1. **list**, 2. **next**, 3. **prev**
Y: 1. στην κενή λίστα, 2. στη **next**, 3. στην **prev**
Z: 1. **list**, 2. **next**, 3. **prev**

Άσκηση 1.3

Εξετάζοντας πάλι το πρόβλημα της αντιστροφής μιας λίστας, θα μπορούσαμε, αντί να περιγράψουμε έναν αλγόριθμο, όπως στην Άσκηση 1.2, να δώσουμε τα αξιώματα που ορίζουν πότε μια λίστα είναι η αντίστροφη μιας άλλης. Μια (ημιτελής) διατύπωση τέτοιων αξιωμάτων είναι η εξής:

Αξιώματα για αντίστροφη λίστας

Αξίωμα 1: *Η αντίστροφη της κενής λίστας είναι* [**X**] .

Αξίωμα 2: *Η αντίστροφη μιας μη κενής λίστας είναι η λίστα που προκύπτει από την τοποθέτηση* [**Y**] *στο τέλος* [**Z**] .

Τι θα πρέπει να συμπληρώσουμε στις θέσεις **X**, **Y** και **Z**, για να είναι τα προηγούμενα αξιώματα σωστά;

Άσκηση 1.4

Δείτε πάλι την Άσκηση 1.1 και διατυπώστε στη μορφή που μάθατε σε αυτό το κεφάλαιο τόσο τα αξιώματα για τον δηλωτικό ορισμό του παραγοντικού, όσο και τα βήματα του αλγορίθμου για τον υπολογισμό του παραγοντικού.

Άσκηση 1.5

Θεωρήστε ότι έχετε στη διάθεσή σας έναν κατευθυνόμενο γράφο χωρίς κύκλους. Θυμηθείτε ότι ένας κατευθυνόμενος γράφος αποτελείται από ένα σύνολο κόμβων και ένα σύνολο ακμών (με κατεύθυνση) μεταξύ κόμβων. Ο γράφος δεν έχει κύκλους, αν δεν υπάρχει περίπτωση να ξεκινήσουμε από κάποιον κόμβο και, ακολουθώντας κατευθυνόμενες ακμές, να επιστρέψουμε στον κόμβο αυτόν. Διατυπώστε τα βήματα ενός αλγορίθμου ο οποίος να ελέγχει αν για δύο κόμβους του γράφου υπάρχει μονοπάτι που να συνδέει τον πρώτο με τον δεύτερο.

Άσκηση 1.6

Δώστε μια δηλωτική προσέγγιση για το πρόβλημα της Άσκησης 1.5, δηλαδή τον έλεγχο της σύνδεσης μεταξύ δύο κόμβων σε έναν κατευθυνόμενο γράφο χωρίς κύκλους.

Απαντήσεις ασκήσεων

Απάντηση άσκησης 1.1

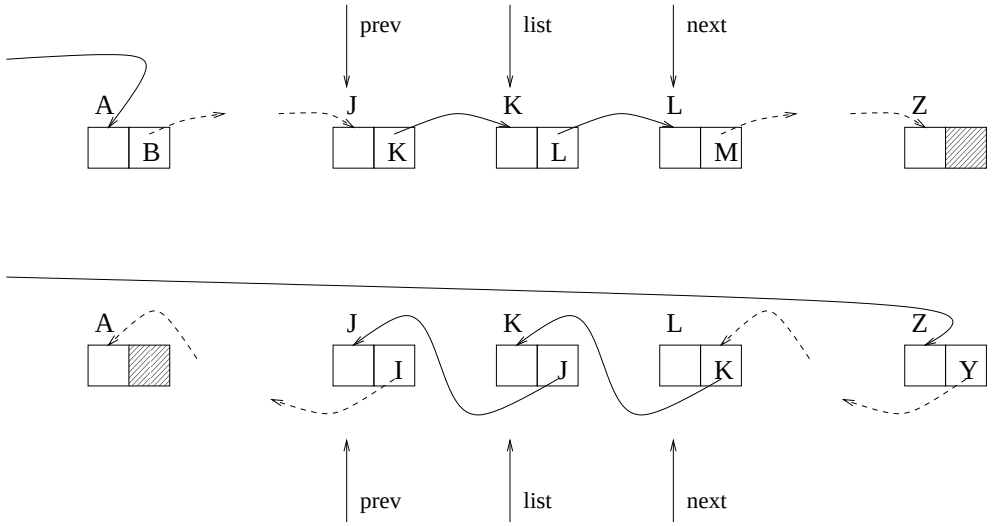
Η μέθοδος 1 είναι δηλωτική, επειδή δίνει τον ορισμό του παραγοντικού, δηλαδή δηλώνει κάτι που ισχύει στα μαθηματικά. Η μέθοδος 2 είναι διαδικαστική, επειδή περιγράφει μια διαδικασία (αλγόριθμο) με την οποία μπορεί κάποιος να υπολογίσει το παραγοντικό ενός αριθμού.

Απάντηση άσκησης 1.2

Οι σωστές απαντήσεις είναι:

X: 2. **next**
Y: 3. στην **prev**
Z: 1. **list**

Στον αλγόριθμο, η λίστα διασχίζεται στοιχείο προς στοιχείο. Σε κάθε επανάληψη κάνουμε το τρέχον στοιχείο να δείχνει στο προηγούμενό του, αντί για το επόμενό του. Βάζουμε το πρώτο στοιχείο της αρχικής λίστας να δείχνει στην κενή λίστα, ενώ το τελευταίο στοιχείο της αρχικής λίστας δεν είναι τίποτε άλλο παρά το πρώτο στοιχείο της νέας αντεστραμμένης λίστας. Σε κάθε επανάληψη του αλγορίθμου, **list** είναι η διεύθυνση του τρέχοντος στοιχείου, **prev** η διεύθυνση του προηγούμενού του και **next** η διεύθυνση του επόμενού του. Πιο παραστατικά, αυτά φαίνονται στο Σχήμα 1.1.



Σχήμα 1.1: Αντιστροφή λίστας.

Στη θέση **X** του αλγορίθμου πρέπει να βάλουμε το «**next**», επειδή, όταν το τρέχον στοιχείο δεν έχει επόμενο (δείχνει στην κενή λίστα), τότε έχουμε φτάσει στο τέλος της αρχικής λίστας. Στη θέση **Y** πρέπει να βάλουμε το «στην **prev**», επειδή, για να επιτευχθεί η αντιστροφή, πρέπει το τρέχον στοιχείο να δείξει στο προηγούμενό του. Τέλος, στη θέση **Z** του αλγορίθμου πρέπει να βάλουμε το «**list**», επειδή, όταν έχει τελειώσει η επαναληπτική διαδικασία του αλγορίθμου, η διεύθυνση του τρέχοντος στοιχείου είναι, ουσιαστικά, η διεύθυνση της αντεστραμμένης λίστας.

Απάντηση άσκησης 1.3
Οι σωστές απαντήσεις είναι:

X: η κενή λίστα
Y: της κεφαλής της
Z: της αντίστροφης της ουράς της

Όσον αφορά την απάντηση για το **X**, δεν θα μπορούσε η αντίστροφη μιας κενής λίστας να είναι τίποτε άλλο παρά η κενή λίστα (θυμηθείτε τις λεγόμενες οριακές περιπτώσεις στα μαθηματικά). Όσο για τα **Y** και **Z**, προκειμένου να διαπιστώσετε ότι, με τις παραπάνω αντικαταστάσεις, το δεύτερο αξίωμα ορίζει σωστά την αντίστροφη μιας μη κενής λίστας, δείτε το εξής παράδειγμα: «*Για να βρούμε την αντίστροφη της λίστας 1, 2, 3, 4, αρκεί να πάρουμε την ουρά της (2, 3, 4) και να τοποθετήσουμε στο τέλος της αντίστροφής της (4, 3, 2) την κεφαλή της αρχικής λίστας (1), παίρνοντας έτσι τη λίστα 4, 3, 2, 1.*».

Επειδή, βέβαια, ο δηλωτικός ορισμός για την αντίστροφη μιας λίστας, που δόθηκε στην άσκηση, περιλαμβάνει και την τοποθέτηση ενός στοιχείου στο τέλος μιας λίστας, για λόγους πληρότητας, δίνουμε τα κατάλληλα αξιώματα και γι' αυτό το πρόβλημα.

Αξιώματα για τοποθέτηση στοιχείου στο τέλος λίστας

Αξίωμα 1: *Η λίστα που προκύπτει από την τοποθέτηση ενός στοιχείου στο τέλος της κενής λίστας είναι μια λίστα που περιλαμβάνει μόνο το στοιχείο αυτό.*

Αξίωμα 2: *Η λίστα που προκύπτει από την τοποθέτηση ενός στοιχείου στο τέλος μιας μη κενής λίστας έχει κεφαλή την κεφαλή της αρχικής και*

ουρά το αποτέλεσμα της τοποθέτησης στο τέλος της ουράς της αρχικής του στοιχείου αυτού.

Απάντηση άσκησης 1.4
Για τη μέθοδο 1, που είναι δηλωτική, θα μπορούσαμε να γράψουμε τα εξής αξιώματα:

Αξιώματα για ορισμό παραγοντικού
Αξίωμα 1: *Το παραγοντικό του 0 ισούται με 1.*
Αξίωμα 2: *Το παραγοντικό ενός θετικού ακέραιου αριθμού ισούται με το γινόμενο του παραγοντικού του προηγούμενου αριθμού επί τον ίδιο τον αριθμό.*

Για τη διαδικαστική μέθοδο 2, ένας αλγόριθμος είναι ο εξής:

Υπολογισμός παραγοντικού του **n**
Βήμα 1: *Αν το* **n** *ισούται με 0, επιστρέψτε το 1 σαν τιμή του παραγοντικού*
Βήμα 2: *Θέστε στο* **fact** *την τιμή 1*
Βήμα 3: *Για κάθε* **i** *από το 1 έως το* **n** *με βήμα 1*
Βήμα 4: *Θέστε στο* **fact** *το* **fact * i**
Βήμα 5: *Επιστρέψτε το* **fact** *ως τιμή του παραγοντικού*

Απάντηση άσκησης 1.5
Ένας αλγόριθμος που ελέγχει αν υπάρχει μονοπάτι που να συνδέει δύο κόμβους σε έναν κατευθυνόμενο γράφο χωρίς κύκλους είναι ο εξής:

Έλεγχος σύνδεσης κόμβων **A** *και* **B** *σε έναν κατευθυνόμενο γράφο χωρίς κύκλους*
Βήμα 1: *Αρχικοποιήστε μια στοίβα* **stack**
Βήμα 2: *Βάλτε στη* **stack** *τον κόμβο* **A**
Βήμα 3: *Αν η* **stack** *είναι άδεια, απαντήστε αρνητικά και τερματίστε*
Βήμα 4: *Βγάλτε από τη* **stack** *το πρώτο της στοιχείο και θέστε το* **current** *ίσο με αυτό*
Βήμα 5: *Αν το* **current** *είναι το* **B***, απαντήστε καταφατικά και τερματίστε*
Βήμα 6: *Βάλτε στη* **stack** *τους κόμβους που είναι επόμενοι του* **current***, αν υπάρχουν τέτοιοι*
Βήμα 7: *Πηγαίνετε στο βήμα 3*

Στον αλγόριθμο αυτόν χρησιμοποιείται μια στοίβα. Θυμηθείτε ότι σε μια στοίβα εισάγουμε στοιχεία στο άκρο της και εξάγουμε στοιχεία από αυτό, με αποτέλεσμα κάθε φορά να εξάγουμε το πιο πρόσφατα εισαχθέν στοιχείο. Στη στοίβα αυτή, ο αλγόριθμος εισάγει κόμβους του γράφου (αρχίζοντας από τον κόμβο **A**), που πρέπει να εξεταστούν αν οδηγούν στον τελικό κόμβο **B**. Σε κάθε επανάληψη, εξάγεται ένας κόμβος από τη στοίβα και, αν δεν είναι ο **B**, εισάγονται στη στοίβα εκείνοι οι κόμβοι στους οποίους μπορούμε να μεταβούμε από αυτόν.

Απάντηση άσκησης 1.6
Για τη δηλωτική αντιμετώπιση του ελέγχου της σύνδεσης δύο κόμβων σε έναν κατευθυνόμενο γράφο χωρίς κύκλους, μπορούμε να διατυπώσουμε τα εξής αξιώματα:

Αξιώματα για σύνδεση δύο κόμβων σε έναν κατευθυνόμενο γράφο χωρίς κύκλους

Αξίωμα 1: *Ένας κόμβος συνδέεται με κάποιον άλλο, αν υπάρχει απευθείας σύνδεση από τον πρώτο στον δεύτερο.*

Αξίωμα 2: *Ένας κόμβος συνδέεται με κάποιον άλλο, αν υπάρχει ένας ενδιάμεσος κόμβος, τέτοιος ώστε να συνδέεται απευθείας ο πρώτος κόμβος με τον ενδιάμεσο και ο ενδιάμεσος να συνδέεται τελικά με τον δεύτερο.*

Προβλήματα

Πρόβλημα 1.1
Δώστε δηλωτικό ορισμό για το άθροισμα των ακέραιων αριθμών από το 1 μέχρι δεδομένο ακέραιο αριθμό n.

Πρόβλημα 1.2
Διατυπώστε τα βήματα αλγορίθμου για τον υπολογισμό του αθροίσματος των ακέραιων αριθμών από το 1 μέχρι δεδομένο ακέραιο αριθμό n.

Πρόβλημα 1.3
Διατυπώστε δηλωτικό ορισμό για τη συνένωση δύο λιστών, δηλαδή την προσθήκη στο τέλος της πρώτης λίστας των στοιχείων της δεύτερης κατά σειρά.

Πρόβλημα 1.4
Ζητείται αλγόριθμος για την εύρεση του n-οστού στοιχείου μιας λίστας, για δεδομένο θετικό ακέραιο n.

Βιβλιογραφικές αναφορές

[1] L. Goldschlager and A. Lister, *Εισαγωγή στη Σύγχρονη Επιστήμη των Υπολογιστών*, Δίαυλος, 1994.

[2] T. Pratt and M. Zelkowitz, *Programming Languages: Design and Implementation*, Prentice Hall, 2000.

[3] J. Martin, *Data Types and Data Structures*, Prentice Hall, 1986.

[4] B. Kernighan and D. Ritchie, *Η Γλώσσα Προγραμματισμού C*, Κλειδάριθμος, 1990.

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

[6] S. Thompson, *Haskell: The Craft of Functional Programming*, Addison Wesley (3rd Edition), 2011.

[7] A. Fischer and F. Grodzinsky, *The Anatomy of Programming Languages*, Prentice Hall, 1993.

Κεφάλαιο 2
Λογικός προγραμματισμός – Υπολογισμός με λογική

Σύνοψη

Το κεφάλαιο αυτό χωρίζεται σε δύο ενότητες. Στην πρώτη ενότητα επιχειρείται μια ιστορική αναδρομή στη λογική και τον λογικό προγραμματισμό, ενώ στη δεύτερη παρουσιάζεται ο κανόνας modus ponens και γίνεται εφαρμογή του σε απλά παραδείγματα.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει στοιχειώδεις γνώσεις μαθηματικής λογικής.

2.1 Ιστορική αναδρομή

Ο **λογικός προγραμματισμός** είναι η πρώτη από τις δύο δηλωτικές προγραμματιστικές φιλοσοφίες που θα μελετήσουμε [1, 2]. Εμφανίστηκε στις αρχές της δεκαετίας του 1970, ως αποτέλεσμα μακροχρόνιας έρευνας στην αυτόματη απόδειξη θεωρημάτων κυρίως, καθώς και προσπαθειών στην επεξεργασία φυσικής γλώσσας και στην τεχνητή νοημοσύνη, γενικότερα.

Βάση του λογικού προγραμματισμού είναι η **λογική**, της οποίας οι ρίζες εντοπίζονται στην εποχή του Αριστοτέλη (4ος αιώνας π.Χ.). Ο Αριστοτέλης πρώτος μελέτησε συστηματικά και θεμελίωσε τη Λογική με τη μορφή που τη μελετάμε ακόμα και σήμερα. Παρ' όλα αυτά, η αυστηρή μαθηματική θεμελίωσή της άρχισε να κτίζεται από το τελευταίο τέταρτο του 19ου αιώνα και συνεχίζεται μέχρι τις μέρες μας.

Σύμφωνα με τον J.A. Robinson [3], ένα πρώτο σημαντικό βήμα στη θεμελίωση της σύγχρονης μορφής της λογικής, αυτό που ονομάζουμε «συμβολική» ή «μαθηματική» λογική, ήταν η εμφάνιση του **κατηγορηματικού λογισμού**. Ο κατηγορηματικός λογισμός ξεκινά από ιδέες που εμφανίστηκαν στο δεύτερο μισό του 19ου αιώνα από τον Gottlob Frege. Βασικό χαρακτηριστικό του είναι ο σαφής ορισμός της έννοιας της απόδειξης, ως μαθηματικής οντότητας η οποία πρέπει να αναλυθεί. Το ίδιο γίνεται για τις προτάσεις και τις άλλες τυπικές εκφράσεις της λογικής, ώστε να μην είναι απλώς εκφραστικά εργαλεία. Πολλοί ερευνητές, ανάμεσά τους ο Kurt Gödel και ο Jacques Herbrand, επεξεργάστηκαν τις ιδέες που εισήχθησαν από τον Frege. Σε αυτούς τους δύο ερευνητές οφείλεται η ανακάλυψη της λεγόμενης **πληρότητας** του κατηγορηματικού λογισμού, που συνδέει την έννοια της αποδειξιμότητας μιας γενικευμένης πρότασης με την έννοια του αληθούς μιας γενικευμένης πρότασης. Η πρώτη έννοια αποτελεί μια συντακτική ιδιότητα, ενώ η δεύτερη μια σημασιολογική. Σύμφωνα με τον Robinson, στο χώρο της σημασιολογίας του κατηγορηματικού λογισμού, σημαντικότατη είναι η συνεισφορά του Alfred Tarski. Αυτός όρισε με σαφή τρόπο

τις έννοιες της ικανοποιησιμότητας, της αλήθειας δεδομένης μιας ερμηνείας, του λογικού επακόλουθου κτλ. Παρ' όλα τα θετικά χαρακτηριστικά του κατηγορηματικού λογισμού, το 1936 ανακαλύφθηκε από τον Alonzo Church, καθώς και από τον Alan Turing, ανεξάρτητα, και ένα αρνητικό. Μολονότι υπάρχει διαδικασία απόδειξης όλων των περιπτώσεων στις οποίες η γενικευμένη πρόταση αποτελεί λογικό επακόλουθο ενός συνόλου γενικευμένων προτάσεων, δεν υπάρχει αντίστοιχη διαδικασία για όλες τις περιπτώσεις στις οποίες αυτό δεν ισχύει.

Αρχικά, οι κανόνες απόδειξης που συνόδευαν τον κατηγορηματικό λογισμό ήταν προσανατολισμένοι σε απλά στοιχειώδη βήματα, εύκολα αντιληπτά από τον ανθρώπινο νου. Χαρακτηριστικό παράδειγμα τέτοιου κανόνα είναι ο **modus ponens**, σύμφωνα με τον οποίο, υποθέτοντας ότι ισχύει το «A» και το «αν A, τότε B», μπορούμε να συμπεράνουμε ότι ισχύει το «B». Η προσέγγιση αυτή ανετράπη όταν ο Robinson εισήγαγε τη λεγόμενη **αρχή της ανάλυσης**. Η αρχή της ανάλυσης, μαζί με μια διαδικασία **ενοποίησης** (ταιριάσματος), μπορούσε να λειτουργήσει ως ένας μηχανισμός απόδειξης θεωρημάτων από αξιώματα διατυπωμένα στον κατηγορηματικό λογισμό. Η προσέγγιση αυτή οδηγεί σε μεθοδολογίες προσανατολισμένες περισσότερο για επεξεργασία από μηχανές παρά για κατανόηση από τον ανθρώπινο νου. Προς τη μεθοδολογία αυτή κατευθύνθηκε ο Robert Kowalski, μελετώντας μια μορφή γραμμικής ανάλυσης, την **SL-ανάλυση** [4].

Την εποχή εκείνη, είχε ξεσπάσει επιστημονική διαμάχη μεταξύ των οπαδών της λογικής και των οπαδών των διαδικαστικών μεθοδολογιών, ως προς την καταλληλότητά τους για την επίλυση προβλημάτων στην περιοχή της τεχνητής νοημοσύνης. Οι δεύτεροι ανήκαν κυρίως στην ομάδα του Αμερικανικού Πανεπιστημίου MIT. Η διαμάχη αυτή απομάκρυνε πολλούς ερευνητές από τη μελέτη μεθοδολογιών βασισμένων στη λογική. Την ίδια εποχή, στη Μασσαλία [5], ο Alain Colmerauer, στο πλαίσιο της μελέτης του για την κατανόηση της φυσικής γλώσσας, χρησιμοποιούσε τη λογική, για να αναπαραστήσει το νόημα των προτάσεων, και την αρχή της ανάλυσης, για να υπολογίσει απαντήσεις σε ερωτήσεις. Ενδιαφέρθηκε, λοιπόν, για την SL-ανάλυση και κάλεσε τον Kowalski στη Μασσαλία. Στο πλαίσιο της συνεργασίας τους, γεννήθηκε αυτό που κατανοούμε σήμερα ως λογικό προγραμματισμό και η εφαρμογή του στην πράξη από τον βασικότερο εκπρόσωπό του, τη γλώσσα προγραμματισμού Prolog. Σε τυπικό επίπεδο, διαπιστώθηκε από τον Kowalski [6] ότι συγκεκριμένες προτάσεις, οι **προτάσεις Horn**, μπορούν να ερμηνευθούν και διαδικαστικά. Αυτό που σήμερα ονομάζεται λογικός προγραμματισμός βασίζεται στη συγκεκριμένη διαπίστωση. Συγκεκριμένα, κάθε πρόταση θεωρείται μια διαδικασία και οι αρνήσεις στην πρόταση αυτή θεωρούνται κλήσεις διαδικασιών. Μάλιστα, ο Kowalski και η ομάδα του Εδιμβούργου ανέπτυξαν και μια ειδική μορφή ανάλυσης, την **SLD-ανάλυση**, που αναφέρεται στις προτάσεις Horn. Θεωρώντας αυτήν τη διαδικασία απόδειξης, ο Kowalski ισχυρίζεται [6] ότι ένα σύνολο από προτάσεις Horn αποτελεί γνώση οργανωμένη και δηλωτικά και διαδικαστικά. Η ονομασία των προτάσεων Horn αποδίδεται στον Alfred Horn, μελετητή της λογικής, που πρώτος εξέτασε κάποιες από τις μαθηματικές τους ιδιότητες [7]. Πρέπει, όμως, εδώ να σημειώσουμε ότι, προκειμένου να μπορέσει να υλοποιηθεί η ιδέα αυτή, ώστε να αποτελέσει μια γλώσσα προγραμματισμού, έπρεπε να παρθούν κάποιες αποφάσεις. Συγκεκριμένα, για να υπάρξει γραμμική εκτέλεση, αποφασίστηκε μια αυστηρά «από επάνω προς τα κάτω» και «από αριστερά προς τα δεξιά» επιλογή. Αυτή η αντιμετώπιση οδήγησε την Prolog να μη διαθέτει κάποιες από τις εξαιρετικές ιδιότητες του λογικού προγραμματισμού, συγκεκριμένα αυτήν της ισοδυναμίας της μοντελοθεωρητικής και της λειτουργικής σημασιολογίας, που είχε παρουσιαστεί από τους M.H. van Emden και R.A. Kowalski [8].

Κλείνοντας, πρέπει να αναφέρουμε ότι ο λογικός προγραμματισμός αποτέλεσε το υπόβαθρο για διάφορες επεκτάσεις και παραλλαγές. Χαρακτηριστικά παραδείγματα είναι η

εισαγωγή άρνησης στον λογικό προγραμματισμό, ο λογικός προγραμματισμός με περιορισμούς, ο απαγωγικός λογικός προγραμματισμός, ο διαζευκτικός λογικός προγραμματισμός κτλ. Επίσης, πρέπει να αναφέρουμε την υλοποίηση του λογικού προγραμματισμού σε παράλληλους υπολογιστές, αλλά και τη χρήση της λογικής και του λογικού προγραμματισμού στις βάσεις δεδομένων.

2.2 Λογική και κανόνας modus ponens

Μην ανησυχείτε που δεν έχουμε δώσει ορισμούς για πολλές από τις έννοιες της προηγουμένης ενότητας, όπως της αρχής της ανάλυσης, της ενοποίησης κτλ. Θα το κάνουμε αρκετά αναλυτικά στο Κεφάλαιο 7. Στο παρόν κεφάλαιο, θα μελετήσουμε τις προτάσεις Horn, τον τρόπο με τον οποίο διατυπώνουμε αξιώματα μέσω αυτών, τον ορισμό του κανόνα modus ponens και την εξαγωγή συμπερασμάτων με τη βοήθειά του.

Όπως είδαμε στο Κεφάλαιο 1, εκτός από τη μεθοδολογία του διαδικαστικού προγραμματισμού, υπάρχει και η μεθοδολογία του δηλωτικού προγραμματισμού. Σε αυτήν, όπως θα θυμάστε, ορίζουμε ένα σύνολο από αξιώματα, με τα οποία περιγράφουμε τι ισχύει στον κόσμο του προβλήματος που μας ενδιαφέρει. Σε ένα περιβάλλον λογικού προγραμματισμού, αυτά τα αξιώματα αποτελούν ένα λογικό πρόγραμμα και, όπως θα δούμε στη συνέχεια, δεν είναι τίποτε άλλο παρά προτάσεις Horn της **λογικής πρώτης τάξης**. Δηλαδή:

Λογικό πρόγραμμα είναι ένα σύνολο από αξιώματα που ισχύουν σε κάποιον κόσμο, διατυπωμένα στη μορφή προτάσεων Horn της λογικής πρώτης τάξης.

Ειδική μορφή λογικής είναι η **προτασιακή λογική**, στην οποία μπορούμε να διατυπώνουμε προτάσεις χρησιμοποιώντας προτασιακά σύμβολα, που παριστάνουν ισχυρισμούς στον περιβάλλοντα κόσμο, αληθείς ή ψευδείς. Οι προτάσεις, εκτός από προτασιακά σύμβολα, είναι δυνατόν να περιλαμβάνουν και λογικούς συνδέσμους, όπως σύζευξη $\wedge$, διάζευξη $\vee$, συνεπαγωγή $\rightarrow$, ισοδυναμία $\leftrightarrow$ και άρνηση $\neg$, αλλά και σημεία στίξης, όπως τα «(», «)» και «,». Κάθε λογικός σύνδεσμος έχει την αναμενόμενη, με βάση την ονομασία του, σημασία. Για παράδειγμα, η συνεπαγωγή $p \rightarrow q$ σημαίνει «αν p, τότε q».

Ο modus ponens είναι ένας κανόνας εξαγωγής συμπερασμάτων στην προτασιακή λογική. Η απλούστερη διατύπωσή του είναι η εξής:

modus ponens: Αν γνωρίζουμε ότι η πρόταση p είναι αληθής και ότι η συνεπαγωγή $p \rightarrow q$ ισχύει (είναι αληθής), τότε μπορούμε να συμπεράνουμε ότι και η πρόταση q είναι αληθής.

Δηλαδή:

$$\left.\begin{array}{c} p \\ p \rightarrow q \end{array}\right\} \Rightarrow q$$

Παράδειγμα 2.1
Αν γνωρίζουμε ότι «*αν βρέχει, τότε έχει υγρασία*» και ότι τώρα «*βρέχει*», μπορούμε να συμπεράνουμε, εφαρμόζοντας τον modus ponens, ότι τώρα «*έχει υγρασία*». Ή, συμβολικά:

$$\left.\begin{array}{c} \textit{βρέχει} \\ \textit{βρέχει} \rightarrow \textit{υγρασία} \end{array}\right\} \Rightarrow \textit{υγρασία} \qquad \square$$

Μια γενικότερη διατύπωση του modus ponens είναι η εξής:

$$\left.\begin{array}{c} p_i \\ p_1 \wedge \ldots \wedge p_i \wedge \ldots \wedge p_n \rightarrow q \end{array}\right\} \Rightarrow p_1 \wedge \ldots \wedge p_{i-1} \wedge p_{i+1} \wedge \ldots \wedge p_n \rightarrow q$$

Ο κανόνας modus ponens αποτελεί ειδική περίπτωση της αρχής της ανάλυσης, όπως θα δούμε στην Ενότητα 7.3.

Ωστόσο, πολλά προβλήματα δεν είναι δυνατόν να επιλυθούν με τη διατύπωση κατάλληλης γνώσης στην απλούστερη εκδοχή της λογικής, που είναι η προτασιακή λογική. Χρειάζεται ένα ισχυρότερο μέσο αναπαράστασης, το οποίο είναι η λογική πρώτης τάξης. Ένα παράδειγμα τέτοιου προβλήματος είναι το εξής:

Παράδειγμα 2.2
Πώς μπορούμε να συμπεράνουμε ότι «*ο Σωκράτης είναι θνητός*», αν γνωρίζουμε ότι «*όλοι οι άνθρωποι είναι θνητοί*» και ότι «*ο Σωκράτης είναι άνθρωπος*»;

Η εφαρμογή του modus ponens στη λογική πρώτης τάξης είναι μια επέκταση για την προτασιακή λογική. Στη λογική πρώτης τάξης θα γράφαμε για το παράδειγμα αυτό:

$$\left.\begin{array}{c} man(Socrates) \\ (\forall x)(man(x) \rightarrow mortal(x)) \end{array}\right\} \Rightarrow mortal(Socrates) \qquad \square$$

Ενώ στην προτασιακή λογική έχουμε απλά **προτασιακά σύμβολα** (π.χ. p, q), στη λογική πρώτης τάξης έχουμε **άτομα**, όπως τα $man(Socrates)$ και $mortal(x)$, τα οποία δομούνται από **κατηγορήματα** (π.χ. man, $mortal$), **σταθερές** (π.χ. $Socrates$), **μεταβλητές** (π.χ. x), καθώς και πιο πολύπλοκες δομές, που ονομάζονται **σύνθετοι όροι** ή **δομές**. Στα αξιώματα που διατυπώσαμε για να περιγράψουμε τη γνώση του Παραδείγματος 2.2 είχαμε ένα απλό άτομο, το $man(Socrates)$, και μια συνεπαγωγή με συμπέρασμα απλό άτομο, το $mortal(x)$, και υπόθεση απλό άτομο, το $man(x)$. Σε κάθε περίπτωση, αν ένα αξίωμα εμπλέκει και μεταβλητές, τότε έχουμε και καθολική ποσοτικοποίηση των μεταβλητών αυτών με τον ποσοδείκτη $\forall$, όπως συμβαίνει στο παράδειγμά μας. Τα απλά άτομα, ως αξιώματα, είναι η μία από τις τρεις περιπτώσεις προτάσεων Horn στη λογική πρώτης τάξης, που ονομάζονται **γεγονότα**. Στις συνεπαγωγές επιτρέπουμε στην υπόθεση, εκτός από ένα απλό άτομο, να έχουμε και σύζευξη απλών ατόμων. Οπότε, έχουμε τη δεύτερη περίπτωση προτάσεων Horn, που είναι οι **κανόνες**. Το συμπέρασμα ενός κανόνα ονομάζεται **κεφαλή**, ενώ οι υποθέσεις του αποτελούν το **σώμα** του κανόνα. Ένα λογικό πρόγραμμα αποτελείται από γεγονότα και κανόνες. Η τρίτη περίπτωση προτάσεων Horn είναι η άρνηση μιας σύζευξης απλών ατόμων, η οποία αντιστοιχεί σε μια **ερώτηση** που υποβάλλεται σε ένα λογικό πρόγραμμα για απάντηση. Στη λογική, μια ερώτηση μπορεί να θεωρηθεί ένα προς απόδειξη θεώρημα, του οποίου η ισχύς μπορεί να προκύψει από τα τιθέντα αξιώματα, δηλαδή τα γεγονότα και τους κανόνες. Για παράδειγμα, η ερώτηση «*υπάρχει κάποιος που να είναι θνητός;*» αντιστοιχεί στην πρόταση Horn $(\forall x)(\neg mortal(x))$. Έτσι, έχουμε:

Μια πρόταση Horn μπορεί να είναι ένα γεγονός, ένας κανόνας ή μια ερώτηση.

Τα αξιώματα λογικής πρώτης τάξης του Παραδείγματος 2.2 συνιστούν, ως προτάσεις Horn, ένα λογικό πρόγραμμα. Αυτό το λογικό πρόγραμμα, στο οποίο προτάσεις με μεταβλητές θεωρούνται ότι ισχύουν καθολικά, είναι το εξής:

$$
\begin{aligned}
man(Socrates) &\leftarrow \\
mortal(x) &\leftarrow man(x)
\end{aligned}
$$

Αν ένα σύστημα λογικού προγραμματισμού εφοδιαστεί με τα προηγούμενα αξιώματα (προτάσεις Horn) και του υποβληθεί η ερώτηση (πρόταση Horn):

$$\leftarrow mortal(x)$$

οφείλει να απαντήσει ότι $x = Socrates$. Δηλαδή, αποδεικνύεται ότι ο Σωκράτης είναι θνητός. Το συγκεκριμένο συμπέρασμα «υπολογίστηκε» όχι με κάποιον αριθμητικό υπολογισμό, αλλά με μια συμβολική επεξεργασία βασισμένη στη λογική. Αυτός είναι ο τρόπος υπολογισμού που υιοθετείται στον **λογικό προγραμματισμό**.

Παρατηρήστε ότι στον κανόνα γράφουμε πρώτα την κεφαλή και μετά το σώμα, με αντεστραμμένο το σύμβολο της συνεπαγωγής. Επίσης, παρατηρήστε ότι ένα γεγονός είναι ουσιαστικά ένας κανόνας χωρίς σώμα, ενώ μια ερώτηση είναι ένας κανόνας χωρίς κεφαλή.

Άσκηση 2.1
Έστω ότι γνωρίζουμε τα εξής: «*Όταν βρέχει και κάνει ζέστη, έχει υγρασία. Το καλοκαίρι κάνει ζέστη. Είναι καλοκαίρι. Βρέχει.*». Ορίζοντας τα προτασιακά σύμβολα p, q, r και s, που σημαίνουν:

p: βρέχει
q: κάνει ζέστη
r: έχει υγρασία
s: είναι καλοκαίρι

επιλέξτε τις προτάσεις της προτασιακής λογικής που αναπαριστούν τη γνώση του κόσμου της άσκησης: p, q, r, s, $p \rightarrow r$, $q \rightarrow s$, $s \rightarrow q$, $r \rightarrow q$, $q \wedge r \rightarrow p$, $s \wedge q \rightarrow r$, $p \wedge q \rightarrow r$, $r \wedge q \wedge p \rightarrow s$.

Άσκηση 2.2
Με ποιες διαδοχικές εφαρμογές του κανόνα εξαγωγής συμπερασμάτων modus ponens στη γνώση που περιγράφει τον κόσμο της Άσκησης 2.1 αποδεικνύεται ότι «*έχει υγρασία*»;

Απαντήσεις ασκήσεων

Απάντηση άσκησης 2.1
Οι απαιτούμενες προτάσεις που κωδικοποιούν τη γνώση του κόσμου της άσκησης είναι:

$p \wedge q \rightarrow r$: όταν βρέχει και κάνει ζέστη, έχει υγρασία
$s \rightarrow q$: το καλοκαίρι κάνει ζέστη
s : είναι καλοκαίρι
p : βρέχει

Όταν διατυπώνουμε σε λογική μια γνώση που είναι εκφρασμένη σε φυσική γλώσσα, δεν θα πρέπει να μας προβληματίζουν κάποιες πιο ελεύθερες εκφράσεις της φυσικής γλώσσας, αλλά θα πρέπει να προσπαθούμε να βρούμε μια ισοδύναμη έκφραση διατυπωμένη περισσότερο μαθηματικά. Για παράδειγμα, η έκφραση «*όταν βρέχει και κάνει ζέστη, έχει υγρασία*»

θα μπορούσε να διατυπωθεί και ως «*αν βρέχει και κάνει ζέστη, τότε έχει υγρασία*», και γι' αυτό να παριστάνεται με τη συνεπαγωγή $p \wedge q \rightarrow r$. Επίσης, η έκφραση «*το καλοκαίρι κάνει ζέστη*» είναι ισοδύναμη με την έκφραση «*αν είναι καλοκαίρι, τότε κάνει ζέστη*» ($s \rightarrow q$). Ένα τελευταίο σχόλιο για την άσκηση είναι το εξής: Αν και οι εκφράσεις «*κάνει ζέστη*» και «*έχει υγρασία*» είναι δυνατόν να αποδειχθούν από τη γνώση μας, με εφαρμογή του modus ponens, όπως θα δούμε στην άσκηση που ακολουθεί, δεν θα πρέπει στη διατύπωση της δοθείσας γνώσης να συμπεριλάβουμε τα q και r, γιατί αυτά δεν είναι αξιώματα του κόσμου που περιγράφουμε, αλλά αποδείξιμα θεωρήματα.

Απάντηση άσκησης 2.2
Μία σωστή απάντηση είναι ο εξής τρόπος απόδειξης:

$$\left.\begin{array}{c} s \\ s \rightarrow q \end{array}\right\} \Rightarrow q$$

$$\left.\begin{array}{c} q \\ p \wedge q \rightarrow r \end{array}\right\} \Rightarrow p \rightarrow r$$

$$\left.\begin{array}{c} p \\ p \rightarrow r \end{array}\right\} \Rightarrow r$$

Άρα, ισχύει το r, δηλαδή «*έχει υγρασία*». Ο προηγούμενος τρόπος απόδειξης δεν είναι ο μοναδικός. Θα μπορούσαμε, για παράδειγμα, στο δεύτερο βήμα να χρησιμοποιήσουμε τα p και $p \wedge q \rightarrow r$, για να αποδείξουμε το $q \rightarrow r$, και μετά, στο τρίτο βήμα, να συνδυάσουμε το $q \rightarrow r$ με το q, για να αποδείξουμε το r.

Προβλήματα

Πρόβλημα 2.1
Αποδείξτε, μέσω διαδοχικών εφαρμογών του κανόνα modus ponens, ότι η πρόταση:

$$u \rightarrow w$$

αποδεικνύεται από τις προτάσεις:

$$\begin{array}{rcl} & & p \\ & & q \\ & & r \\ p \wedge q & \rightarrow & s \\ q \wedge r \wedge s & \rightarrow & t \\ p \wedge s \wedge t \wedge u & \rightarrow & w \end{array}$$

Πρόβλημα 2.2
Διερευνήστε αν οι προτάσεις $p \rightarrow \neg q \vee r$ και $p \wedge q \rightarrow r$ είναι ουσιαστικά ισοδύναμες. Αιτιολογήστε την απάντησή σας, αποδίδοντας σε καθένα από τα προτασιακά σύμβολα p, q και r κάποια φυσική σημασία από τον πραγματικό κόσμο, έτσι ώστε, μέσω αυτών, η ισοδυναμία των παραπάνω προτάσεων να είναι ζήτημα απλής κοινής λογικής.

Πρόβλημα 2.3
Διατυπώστε σε λογική πρώτης τάξης τη γνώση «*οι εφοπλιστές που δεν έχουν χρεοκοπήσει είναι πλούσιοι*».

Βιβλιογραφικές αναφορές

[1] Γ. Μητακίδης, *Απο τη Λογική στο Λογικό Προγραμματισμό και την Prolog*, Καρδαμίτσας, 1992.

[2] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1993.

[3] J. A. Robinson, Logic and Logic Programming, *CACM*, 35(3), 40-65, 1992.

[4] R. Kowalski and D. Kuehnm, Linear Resolution with Selection Function, *Artificial Intelligence*, 2, 227-260, 1971.

[5] R. Kowalski, A Short Story of My Life and Work, `http://www.doc.ic.ac.uk/~rak/history.pdf`, 2015.

[6] R. Kowalski, Predicate Logic as Programming Language, *Proceedings of IFIP '74*, 569-574, 1974.

[7] R. Kowalski, Logic Programming, in *Computational Logic*, Elsevier, 2014

[8] M. H. van Emden and R. A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *JACM*, 23(4), 73-742, 1976.

Κεφάλαιο 3
Η γλώσσα λογικού προγραμματισμού Prolog – Τα βασικά

Σύνοψη

Στο κεφάλαιο αυτό εξετάζεται αρχικά ο τρόπος σύνταξης προγραμμάτων Prolog, στα οποία εκφράζεται απλή γνώση, υπό τη μορφή γεγονότων και κανόνων, και υποβάλλονται ερωτήσεις για την επίλυση προβλημάτων. Στη συνέχεια, γίνεται αναφορά στη δυνατότητα διατύπωσης αναδρομικών ορισμών, μέσω της Prolog, για την κωδικοποίηση της γνώσης του κόσμου με απλό και κομψό τρόπο. Επίσης, παρουσιάζονται η διαδικασία που εφαρμόζεται στην Prolog για τον υπολογισμό των απαντήσεων στις υποβαλλόμενες ερωτήσεις και ο μηχανισμός της οπισθοδρόμησης, που ενεργοποιείται όταν η διαδικασία αυτή οδηγείται σε αδιέξοδο, με τον οποίο της δίνεται η ευκαιρία να εξετάσει εναλλακτικές δυνατότητες απόδειξης του ζητουμένου. Τέλος, αναφέρονται συνοπτικά οι κυριότερες περιοχές εφαρμογών της Prolog και οι δύο προσεγγίσεις για την υλοποίηση συστημάτων Prolog, μέσω διερμηνέων και μεταγλωττιστών.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης δεν απαιτείται να έχει συγκεκριμένες γνώσεις.

3.1 Γεγονότα, κανόνες και ερωτήσεις

Στο Κεφάλαιο 2 αναφερθήκαμε στα λογικά προγράμματα με τα οποία μπορούμε να διατυπώνουμε αξιώματα υπό τη μορφή προτάσεων Horn και να εξάγουμε συμπεράσματα μέσω του κανόνα modus ponens. Η γλώσσα λογικού προγραμματισμού Prolog είναι ένα συγκεκριμένο περιβάλλον, το οποίο μας επιτρέπει να το εφοδιάζουμε με γνώση σαν ένα σύνολο προτάσεων που ισχύουν και να του υποβάλλουμε ερωτήματα, για να μας απαντά με τη βοήθεια της γνώσης με την οποία το έχουμε εφοδιάσει. Η ονομασία της προέρχεται από τα αρχικά γράμματα των λέξεων της έκφρασης «**Pro**γραμματίζοντας στη **log**ική» [1, 2, 3].

Παίρνοντας το παράδειγμα του λογικού προγράμματος που είδαμε στο Κεφάλαιο 2:

$$\begin{aligned} man(Socrates) &\leftarrow \\ mortal(x) &\leftarrow man(x) \end{aligned}$$

θα μπορούσαμε να γράψουμε το εξής αντίστοιχο πρόγραμμα Prolog:

```
man(socrates).
mortal(X) :- man(X).
```

Σε αυτό το πρόγραμμα Prolog, θα μπορούσαμε να υποβάλουμε την ερώτηση:

```
?- mortal(X).
```

που σημαίνει: «*γνωρίζεις κάποιον που είναι θνητός;*», και να πάρουμε την απάντηση:

```
X = socrates
```

Παρατηρήστε ότι οι συντακτικές διαφορές ανάμεσα στο λογικό πρόγραμμα και το πρόγραμμα Prolog είναι επουσιώδεις. Το $\leftarrow$ αντικαταστάθηκε από το `:-`, η σταθερά $Socrates$ έγινε `socrates`, με μικρό «s», η μεταβλητή x έγινε `X`, με κεφαλαίο χαρακτήρα, και οι προτάσεις τελειώνουν με «`.`». Η σύνταξη αυτή είναι η συνηθισμένη στα συστήματα Prolog.[1]

Πολλές είναι οι υλοποιήσεις της γλώσσας Prolog στα σχεδόν 40 χρόνια της ύπαρξής της. Από τις υλοποιήσεις αυτές, άλλες είναι εμπορικά συστήματα, ενώ άλλες είναι κατασκευασμένες από την ακαδημαϊκή και ερευνητική κοινότητα, και διατίθενται ελεύθερα, για ανάπτυξη πειραματικών εφαρμογών και εκπαίδευση. Μεταξύ των εμπορικών συστημάτων, μπορούμε να αναφέρουμε την **Prolog IV**, την **LPA Prolog** και την **SICStus Prolog**. Μερικά αρκετά διαδεδομένα ελεύθερα διαθέσιμα συστήματα είναι η **BProlog**, η **ECL^{i}PS^e**, η **GNU Prolog**, η **SWI Prolog** και η **XSB Prolog**.

Για να αποκτήσετε εμπειρία υψηλού επιπέδου στον προγραμματισμό με την Prolog, απαραίτητη προϋπόθεση είναι να χρησιμοποιείτε, παράλληλα με τη μελέτη σας, και ένα σύστημα Prolog στον υπολογιστή, ώστε να μπορείτε να βλέπετε στην πράξη πώς συμπεριφέρονται τα προγράμματά της που θα δίνονται στα παραδείγματα, αλλά να δοκιμάζετε και τις ασκήσεις που θα σας ζητούνται για απάντηση. Δεν έχει σημασία ποιο σύστημα Prolog θα χρησιμοποιήσετε. Όλα τα συστήματα έχουν τον ίδιο πυρήνα, που είναι ο βασικός κορμός της Prolog. Συνεπώς, οποιοδήποτε από τα προαναφερθέντα συστήματα, αλλά ίσως και κάποιο άλλο, θα μπορεί να σας βοηθήσει στην πρακτική εξάσκηση με το αντικείμενο τόσο αυτού του κεφαλαίου, όσο και των επομένων, που αφορούν τον προγραμματισμό στην Prolog.

Θα αρχίσουμε την παρουσίαση των δυνατοτήτων της γλώσσας με ένα απλό, αλλά ταυτόχρονα πολύ ισχυρό παράδειγμα, το οποίο θα μπορούσε να χαρακτηριστεί το κλασικό παράδειγμα του οποίου παραλλαγές χρησιμοποιούν σχεδόν όλα τα βιβλία που πραγματεύονται τον προγραμματισμό σε Prolog, για μια ουσιώδη και κατανοητή εισαγωγή στη γλώσσα. Τηρουμένων των αναλογιών, το πρόγραμμα το οποίο θα παρουσιαστεί αντιστοιχεί στο πρώτο πρόγραμμα που γράφουμε σε μια γλώσσα διαδικαστικού προγραμματισμού, αυτό το οποίο εκτυπώνει το μήνυμα «`Hello, world!`».

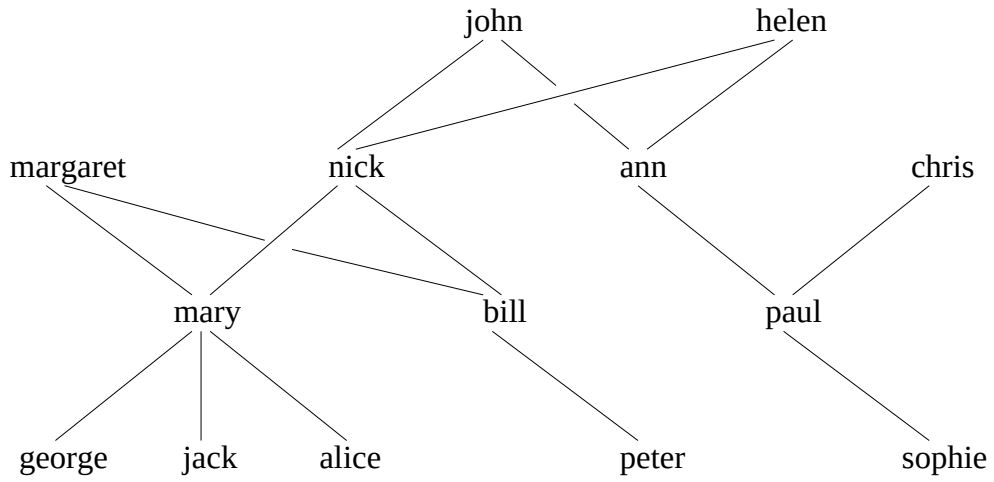
Παράδειγμα 3.1
Θεωρήστε τον εξής κόσμο: «*Ο Γιάννης και η Ελένη είναι γονείς του Νίκου και της Άννας. Ο Νίκος και η Μαργαρίτα έχουν παιδιά τη Μαίρη και τον Βασίλη. Ο Γιώργος, ο Ιάκωβος και η Αλίκη είναι παιδιά της Μαίρης, και ο Πέτρος είναι παιδί του Βασίλη. Ο Χρήστος και η Άννα είναι γονείς του Παύλου, ο οποίος έχει παιδί τη Σοφία*».

[1] Θα μπορούσαμε να χρησιμοποιήσουμε λέξεις της ελληνικής γλώσσας για να παραστήσουμε τα αντικείμενα και τις έννοιες σε ένα πρόγραμμα Prolog. Δηλαδή, αντί να είχαμε `man`, `mortal` και `socrates` στο πρόγραμμα, θα ήταν αρκετό να γράφαμε `άνθρωπος`, `θνητός` και `σωκράτης`, αντίστοιχα. Επί της ουσίας, δεν θα υπήρχε κανένα πρόβλημα, γιατί τα σύμβολα αυτά δεν είναι κάποιες λέξεις-κλειδιά της γλώσσας, ώστε να πρέπει να γραφούν με συγκεκριμένο τρόπο. Είναι σύμβολα της επιλογής του προγραμματιστή. Άρα, θα μπορούσε εκείνος, αν τον εξυπηρετούσε, να χρησιμοποιήσει τις ελληνικές λέξεις. Θα ανέκυπτε όμως πρακτικό πρόβλημα, γιατί δεν υπάρχει κάποιο διαδεδομένο σύστημα Prolog το οποίο να επιτρέπει στα προγράμματα που του δίνονται να περιλαμβάνουν σύμβολα με ελληνικούς χαρακτήρες. Γι' αυτόν το λόγο, είμαστε πρακτικά δεσμευμένοι να χρησιμοποιούμε σύμβολα με λατινικούς χαρακτήρες. Μεταξύ των δύο επιλογών, δηλαδή να χρησιμοποιούμε ελληνικές λέξεις γραμμένες με λατινικούς χαρακτήρες ή αγγλικές λέξεις, επιλέξαμε τη δεύτερη, ως πιο κομψή.

Αυτό το γενεαλογικό δέντρο φαίνεται στο Σχήμα 3.1. Ένα πρόγραμμα Prolog που κωδικοποιεί τις σχέσεις γονέων-παιδιών του παραδείγματος είναι το εξής:

```
parent(john, nick).     parent(john, ann).      parent(helen, nick).
parent(helen, ann).     parent(nick, mary).     parent(nick, bill).
parent(margaret, mary). parent(margaret, bill). parent(mary, george).
parent(mary, jack).     parent(mary, alice).    parent(bill, peter).
parent(chris, paul).    parent(ann, paul).      parent(paul, sophie).
```

□



Σχήμα 3.1: Ένα γενεαλογικό δέντρο.

Στο πρόγραμμα που είδαμε στο Παράδειγμα 3.1, οποιαδήποτε έκφραση της μορφής «`parent(john, nick).`» αποτελεί ένα γεγονός που εκφράζει ότι ένας ισχυρισμός είναι αληθινός. Για παράδειγμα, «*ο Γιάννης είναι γονιός του Νίκου*». Το σύμβολο `parent` είναι ένα **κατηγόρημα**, που χρησιμοποιείται για να εκφράσει κάποια σχέση, αυτήν του γονιού-παιδιού, μεταξύ δύο οντοτήτων, γι' αυτό και λέμε ότι έχει **βαθμό** 2 και μπορούμε να το γράφουμε και ως `parent/2`. Τα `john`, `nick` κτλ. είναι σύμβολα που αποφασίσαμε να χρησιμοποιήσουμε για να παραστήσουμε συγκεκριμένες οντότητες του κόσμου μας, όπως τον *Γιάννη*, τον *Νίκο* κτλ., και γι' αυτόν τον λόγο ονομάζονται **σταθερές**.

Έχοντας εφοδιάσει ένα σύστημα Prolog με το πρόγραμμα του Παραδείγματος 3.1, μπορούμε να του υποβάλουμε ερωτήσεις, όπως «*είναι ο Νίκος γονιός του Βασίλη;*» ή «*είναι η Ελένη γονιός της Μαίρης;*» ή «*είναι η Άννα γονιός του Μιχάλη;*», όπως φαίνεται στη συνέχεια, και να πάρουμε τις κατάλληλες απαντήσεις:

```
?- parent(nick, bill).
yes
?- parent(helen, mary).
no
?- parent(ann, michael).
no
?-
```

Προσέξτε ότι στην τελευταία ερώτηση, παρότι στο πρόγραμμα δεν υπήρχε κάποια πληροφορία για τον *Μιχάλη* (`michael`), η απάντηση είναι αρνητική. Η απάντηση αυτή δεν πρέπει να ερμηνευθεί ως «*δεν ισχύει ότι η Άννα είναι γονιός του Μιχάλη*», αλλά ως «*δεν γνωρίζω*

ότι η Άννα είναι γονιός του Μιχάλη». Αυτές είναι δύο εντελώς διαφορετικές απαντήσεις και θα πρέπει να γίνει από πολύ νωρίς κτήμα μας ότι τα αρνητικά συμπεράσματα στην Prolog σημαίνουν «δεν γνωρίζω» και όχι «δεν ισχύει».

Εκτός, όμως, από ερωτήσεις που επιδέχονται απλώς μια καταφατική ή μια αρνητική απάντηση, θα μπορούσαμε να υποβάλουμε και ερωτήσεις που περιμένουν και κάποια επιπλέον πληροφορία ως απάντηση. Αυτό μπορεί να γίνει με τη χρήση **μεταβλητών** που παριστάνουν τυχαίες οντότητες του κόσμου μας. Συντακτικά, οι μεταβλητές διακρίνονται από τις σταθερές και τα κατηγορήματα, καθώς είναι σύμβολα που αρχίζουν με κεφαλαίο χαρακτήρα, ενώ οι σταθερές και τα κατηγορήματα αρχίζουν με μικρό. Δείτε κάποιες τέτοιες ερωτήσεις και τις αντίστοιχες απαντήσεις ενός συστήματος Prolog:

```
?- parent(chris, X).
X = paul
?- parent(X, alice).
X = mary
?- parent(X, john).
no
?- parent(michael, X).
no
?-
```

Η πρώτη ερώτηση σημαίνει «*γνωρίζεις κάποιον με γονιό τον Χρήστο;*» και δέχεται την κατάλληλη απάντηση, όπως συμβαίνει και με την τρίτη ερώτηση «*γνωρίζεις κάποιο γονιό του Γιάννη;*», στην οποία η απάντηση είναι, φυσιολογικά, αρνητική.

Επίσης, είναι δυνατόν σε κάποια ερώτηση με μεταβλητές που υποβάλλουμε να υπάρχουν περισσότερες της μιας απαντήσεις. Τότε, τα συστήματα Prolog δίνουν την πρώτη απάντηση και περιμένουν κάποια είσοδο από το χρήστη. Αν η είσοδος αυτή είναι «`;`», τότε δίνουν και επόμενη απάντηση. Αν η είσοδος είναι οτιδήποτε άλλο, τότε σταματούν την περαιτέρω παρουσίαση απαντήσεων. Δείτε μια τέτοια ερώτηση:

```
?- parent(mary, X).
X = george     ;
X = jack       ;
X = alice
?-
```

Μπορούμε, ακόμα, να υποβάλουμε ερωτήσεις με περισσότερες της μιας μεταβλητές, όπως «*γνωρίζεις κάποιο ζευγάρι γονιού-παιδιού;*», ως εξής:

```
?- parent(X, Y).
X = john
Y = nick       ;
X = john
Y = ann        ;
X = helen
Y = nick       ;
.................
?-
```

Με αυτήν την ερώτηση, ουσιαστικά, μας επιστρέφονται όλα τα ζευγάρια που έχουν οριστεί στο πρόγραμμα με το κατηγόρημα `parent/2`.

Εκτός από τις απλές ερωτήσεις (ή **στόχους**), που είδαμε μέχρι τώρα, μπορούμε να υποβάλουμε και σύνθετες, οι οποίες αποτελούνται από δύο ή περισσότερες απλές ερωτήσεις,

μεταξύ των οποίων υπονοείται λογική σύζευξη. Συντακτικά, η σύζευξη των στόχων παριστάνεται με «,». Οι σύνθετες ερωτήσεις έχουν ενδιαφέρον, όταν υπάρχουν κοινές μεταβλητές μεταξύ των στόχων που τις αποτελούν, γιατί αυτό σημαίνει ότι οι στόχοι θα πρέπει να αληθεύουν ταυτόχρονα για τις ίδιες τιμές των κοινών μεταβλητών τους. Παραδείγματα τέτοιων ερωτήσεων είναι «*γνωρίζεις κάποιο γονιό του Παύλου και κάποιο γονιό του γονιού του;*» και «*γνωρίζεις κάποιο γονιό του Βασίλη και της Μαίρης;*», με τις εξής αντίστοιχες ερωτήσεις και απαντήσεις σε Prolog:

```
?- parent(X, paul), parent(Y, X).
X = ann
Y = john       ;
X = ann
Y = helen      ;
no
?- parent(X, bill), parent(X, mary).
X = nick       ;
X = margaret   ;
no
?-
```

Πριν προχωρήσουμε σε περισσότερο ισχυρούς τρόπους διατύπωσης γνώσης σε προγράμματα Prolog, είναι χρήσιμο να δοκιμάσετε να λύσετε την άσκηση που ακολουθεί, για να μπορέσετε να διαπιστώσετε αν έχετε κατανοήσει πλήρως ό,τι παρουσιάστηκε μέχρι στιγμής σε αυτήν την ενότητα.

Άσκηση 3.1
Δεδομένου του προγράμματος με τα γενεαλογικά δεδομένα αυτής της ενότητας, αντιστοιχίστε τις ερωτήσεις Prolog του Πίνακα 3.1 στις κατάλληλες απαντήσεις. Μπορεί να υπάρχουν παραπάνω της μιας ερωτήσεις με την ίδια απάντηση, όπως και απαντήσεις που δεν αντιστοιχούν σε καμία ερώτηση. □

Εκτός από την εξαντλητική καταγραφή γεγονότων για τον ορισμό μιας σχέσης, όπως κάναμε με το κατηγόρημα `parent`, μπορούμε στην Prolog να ορίσουμε έμμεσα μια σχέση μέσω μιας άλλης ή άλλων σχέσεων. Αυτό μπορεί να γίνει με τους **κανόνες**, που αποτελούν, μαζί με τα γεγονότα, τους δύο τρόπους αναπαράστασης γνώσης σε προγράμματα Prolog. Τα γεγονότα και οι κανόνες σε ένα πρόγραμμα Prolog είναι οι **προτάσεις** του προγράμματος. Δείτε έναν κανόνα που θα μπορούσαμε να γράψουμε:

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

Αυτός ο κανόνας ορίζει το κατηγόρημα `grandparent/2` μέσω του `parent/2` ως εξής: «*Αν ο X είναι γονιός του Y και ο Y είναι γονιός του Z, τότε ο X είναι παππούς ή γιαγιά του Z.*». Ένας κανόνας εκφράζει ότι ισχύει κάτι αν ισχύει κάτι άλλο, δηλαδή είναι μια συνεπαγωγή. Η ιδιορρυθμία στη σύνταξη των κανόνων στην Prolog αφορά το γεγονός ότι το συμπέρασμα της συνεπαγωγής, που ονομάζεται και **κεφαλή** του κανόνα, είναι αριστερά και οι υποθέσεις, που αποτελούν το **σώμα** του κανόνα, είναι δεξιά. Το σύμβολο της συνεπαγωγής είναι το «`:-`», που ουσιαστικά σημαίνει το ←, και στο σώμα του κανόνα οι υποθέσεις είναι λογικά συζευγμένες, στοιχείο που εκφράζεται με το συνδετικό «,». Μπορούμε να υποβάλουμε ερωτήσεις για κατηγορήματα που ορίζονται με κανόνες, όπως κάνουμε και γι' αυτά που ορίζονται με γεγονότα. Για παράδειγμα:

Ερώτηση		Απάντηση	
1.	`?- parent(mary, nick).`	1.	`X = mary ;` `X = bill`
2.	`?- parent(jack, X).`	2.	`no`
3.	`?- parent(bill, peter).`	3.	`Error!`

4.	`?- parent(X, X).`	4.	`X = nick`
			`Y = bill ;`
			`no`

5.	`?- parent(mark, thomas).`	5.	`yes`
6.	`?- parent(X, mary).`	6.	`X = margaret`

7.	`?- parent(margaret, X),`
	`parent(nick, X).`
		7.	`X = chris`
			`Y = peter ;`
			`X = ann`
			`Y = peter ;`
			`no`

8.	`?- parent(X, Y),` `parent(Y, X).`	8.	`X = mary ;` `no`
9.	`?- parent(X, paul),` `parent(bill, Y).`	9.	`X = bill` `Y = nick`

10.	`?- parent(helen, X),`
	`parent(X, Y),`
	`parent(Y, peter).`
		10.	`X = nick ;`
			`X = margaret ;`
			`no`

Πίνακας 3.1: Μερικές ερωτήσεις Prolog και πιθανές απαντήσεις τους.

```
?- grandparent(X, mary).
X = john       ;
X = helen      ;
no
?- grandparent(john, X), grandparent(helen, X).
X = mary       ;
X = bill       ;
X = paul       ;
no
?-
```

Μπορούμε να διατυπώσουμε ακόμα πιο ενδιαφέροντες κανόνες, αν επαυξήσουμε τη γνώση μας με το ποια άτομα είναι άνδρες και ποια γυναίκες. Γι' αυτόν το σκοπό, μπορούμε να ορίσουμε τα κατηγορήματα `male/1` και `female/1` μέσω γεγονότων, έτσι ώστε το πρώτο να αληθεύει για κάθε άνδρα και το δεύτερο για κάθε γυναίκα:

```
male(john).              male(nick).
male(bill).              male(george).
male(jack).              male(peter).
male(chris).             male(paul).

female(helen).           female(margaret).
female(mary).            female(alice).
female(ann).             female(sophie).
```

Έτσι, μπορούμε, για παράδειγμα, να ορίσουμε ένα κατηγόρημα `father/2`, το οποίο να εκφράζει ότι «*πατέρας κάποιου είναι ένας άνδρας που είναι γονιός του*», ή ένα κατηγόρημα

`sister/2`, για να ορίσουμε ότι «*αδελφή κάποιου είναι μια γυναίκα που έχει κοινό γονιό με αυτόν*», ως εξής:

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

Κάποιες ερωτήσεις που θα μπορούσαμε να υποβάλουμε σε αυτά τα κατηγορήματα, με τις αντίστοιχες απαντήσεις τους, είναι:

```
?- father(X, paul).
X = chris       ;
no
?- father(nick, X).
X = mary        ;
X = bill
?- father(ann, X).
no
?- sister(X, jack).
X = alice       ;
no
?- sister(alice, X).
X = george      ;
X = jack        ;
X = alice       ;
no
?-
```

Θα σας προβλημάτισε, ίσως, η απάντηση στην τελευταία ερώτηση. Είναι δυνατόν η *Αλίκη* να είναι αδελφή του εαυτού της (`X = alice`); Φυσικά, δεν είναι, αλλά ένα σύστημα Prolog που δίνει μια τέτοια απάντηση δεν μπορεί να γνωρίζει ότι αυτό είναι παράλογο, εκτός αν το έχουμε πληροφορήσει εμείς. Ουσιαστικά, το πρόβλημα βρίσκεται στον κανόνα με τον οποίο ορίσαμε το κατηγόρημα `sister/2`. Αυτός ο κανόνας δεν αποκλείει κάποια γυναίκα να είναι αδελφή του εαυτού της, αφού έχει κοινό γονιό με τον εαυτό της! Για να διορθώσουμε τον κανόνα, μπορούμε να προσθέσουμε στους στόχους του σώματός του ότι τα `X` και `Y` πρέπει να είναι διαφορετικά. Δηλαδή:

```
sister(X, Y) :- female(X), parent(Z, X), parent(Z, Y), diff(X, Y).
```

Φυσικά, τώρα θα πρέπει να ορίσουμε και το κατηγόρημα `diff/2`, που ορίζει πότε δύο σταθερές είναι διαφορετικές. Αυτό δεν είναι εύκολο με τη γνώση που έχουμε μέχρι αυτήν τη στιγμή. Θα μπορούσαμε απλώς, για κάθε ζευγάρι διαφορετικών σταθερών που γνωρίζουμε, να δηλώσουμε ένα κατάλληλο γεγονός. Για παράδειγμα:

```
diff(john, helen).
diff(john, nick).
diff(ann, john).
.......................
```

Έτσι, αφού έχουμε 14 σταθερές στο πρόγραμμά μας, θα πρέπει να δηλώσουμε $14 \times 13 = 182$ γεγονότα `diff/2`, για να έχουμε τον σωστό ορισμό του `sister/2`. Καταλαβαίνουμε, βέβαια, ότι αυτό δεν είναι ιδιαίτερα έξυπνο και αποδοτικό, ιδίως αν έχουμε πολύ περισσότερες σταθερές. Ωστόσο, αυτό είναι το καλύτερο που μπορούμε να κάνουμε τη δεδομένη στιγμή. Σε μεταγενέστερο κεφάλαιο, θα δούμε μια κομψή λύση του προβλήματος.

Θα κλείσουμε αυτήν την ενότητα με κάποιες ασκήσεις, για να μπορέσετε να εμπεδώσετε περισσότερο τη διαδικασία συγγραφής κανόνων στην Prolog.

Άσκηση 3.2
Έστω ότι στο γενεαλογικό μας πρόγραμμα (κατηγορήματα `parent/2`, `male/1`, `female/1`, `grandparent/2`, `father/2` και `sister/2` - το τελευταίο χωρίς το στόχο `diff/2` στο σώμα του κανόνα), υποβάλουμε την εξής ερώτηση:

```
?- sister(X, nick).
```

Ποια από τις παρακάτω απαντήσεις πιστεύετε ότι θα πάρουμε;

1.
```
X = ann ;
no
```
2.
```
X = ann ;
X = ann ;
no
```
3.
```
no
```

Άσκηση 3.3
Συμπληρώστε κατάλληλα τους ορισμούς που ακολουθούν:

1.
```
% mother/2: μητέρα κάποιου
mother(X, Y) :- 
```
2.
```
% brother/2: αδελφός κάποιου
brother(X, Y) :- 
```
3.
```
% uncle/2: θείος κάποιου
uncle(X, Y) :- 
```
4.
```
% aunt/2: θεία κάποιου
aunt(X, Y) :- 
```
5.
```
% grandchild/2: εγγόνι κάποιου
grandchild(X, Y) :- 
```
6.
```
% grandfather/2: παππούς κάποιου
grandfather(X, Y) :- 
```
7.
```
% grandmother/2: γιαγιά κάποιου
grandmother(X, Y) :- 
```
8.
```
% son/2: γιος κάποιου
son(X, Y) :- 
```
9.
```
% daughter/2: κόρη κάποιου
daughter(X, Y) :- 
```
10.
```
% hbsad/2: κάποιος έχει και γιο και κόρη
hbsad(X) :- 
```

Νομίζετε ότι υπάρχει κάποιο πρόβλημα με τους ορισμούς των κατηγορημάτων `uncle/2` και `aunt/2`; Τι απάντηση θα πάρουμε αν στο γενεαλογικό μας πρόγραμμα υποβάλουμε την ερώτηση `uncle(nick, bill)`;

Στην άσκηση αυτή βλέπετε και έναν τρόπο να συμπεριλαμβάνετε σχόλια στα προγράμματά σας. Οτιδήποτε βρίσκεται έπειτα από ένα `%` και μέχρι το τέλος της γραμμής είναι σχόλιο.

3.2 Αναδρομή

Στην Ενότητα 3.1 είδαμε πώς μπορούμε να περιγράψουμε απλούς κόσμους στην Prolog διατυπώνοντας προγράμματα που αποτελούνται από γεγονότα και κανόνες, και πώς, στη συνέχεια, μπορούμε να υποβάλουμε ερωτήσεις σε αυτά τα προγράμματα, για την επίλυση προβλημάτων που αντιμετωπίζουμε. Αν θεωρήσουμε πάλι τον κόσμο του Παραδείγματος 3.1, ένα τέτοιο πρόβλημα θα μπορούσε να ήταν η επιβεβαίωση αν κάποιος είναι πρόγονος (ή απόγονος) κάποιου άλλου ή, ακόμα, και η εύρεση των προγόνων (ή απογόνων) κάποιου. Για την αντιμετώπιση προβλημάτων σαν κι αυτά, θα αρκούσε ίσως να σκεφτούμε ότι «*πρόγονος κάποιου είναι ένας γονιός του ή ένας γονιός ενός γονιού του ή ένας γονιός ενός γονιού ενός γονιού του κ.ο.κ.*». Θα μπορούσαμε, δηλαδή, να ορίσουμε ένα σύνολο από εναλλακτικούς ορισμούς, όπως φαίνεται στη συνέχεια, μέσω του κατηγορήματος `ancestor/2`, που να καλύπτει τα διάφορα πιθανά ενδεχόμενα:

```
ancestor(X1, X2) :- parent(X1, X2).
ancestor(X1, X3) :- parent(X1, X2), parent(X2, X3).
ancestor(X1, X4) :- parent(X1, X2), parent(X2, X3), parent(X3, X4).
.................................
```

Ωστόσο, δεν είναι ιδιαίτερα πρακτικό να διατυπώσουμε έναν τόσο μεγάλο αριθμό κανόνων, γιατί, αφού αυτός θα είναι, εκ των πραγμάτων, πεπερασμένος, ουσιαστικά ο ορισμός μας θα είναι ελλιπής. Αυτό συμβαίνει επειδή υπάρχει ένα ανώτατο όριο στο βάθος του γενεαλογικού δέντρου μας μέχρι το οποίο θα μπορούμε να αποδείξουμε ότι κάποιος είναι πρόγονος κάποιου άλλου. Συνεπώς, το ερώτημα που τίθεται αφορά τη δυνατότητα να ορίσουμε το κατηγόρημα `ancestor/2` με πιο κομψό τρόπο, έτσι ώστε να μπορεί να περιγραφεί η σχέση «πρόγονος» σε αυθαίρετο βάθος. Η απάντηση σε αυτό το ερώτημα είναι, ευτυχώς, καταφατική. Αρκεί να ορίσουμε το κατηγόρημά μας με τη βοήθεια **αναδρομής**, δηλαδή να πούμε ότι «*πρόγονος κάποιου είναι ένας γονιός του ή ένας γονιός ενός προγόνου του*». Η αναδρομή έγκειται στο γεγονός ότι, για τον ορισμό της σχέσης «πρόγονος», χρησιμοποιείται η ίδια σχέση. Οι κανόνες Prolog που υλοποιούν τον ορισμό αυτόν είναι οι εξής:

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

Έτσι, επαυξάνοντας το πρόγραμμά μας με τους προηγούμενους κανόνες, ο δεύτερος από τους οποίους είναι αναδρομικός, μπορούμε να υποβάλουμε και ερωτήσεις για προγόνους (και απογόνους). Για παράδειγμα:

```
?- ancestor(helen, alice).
yes.
?- ancestor(nick, sophie).
no.
?- ancestor(ann, X).
X = paul       ;
X = sophie     ;
no
?- ancestor(X, bill).
X = nick       ;
X = margaret   ;
X = john       ;
X = helen      ;
no
?-
```

Οι δύο τελευταίες ερωτήσεις σημαίνουν «*ποιοι είναι οι απόγονοι της Άννας;*» και «*ποιοι είναι οι πρόγονοι του Βασίλη;*», αντίστοιχα.

Με τη δυνατότητα διατύπωσης αναδρομικών κανόνων στην Prolog, έχουμε στα χέρια μας ένα πολύ δυνατό προγραμματιστικό εργαλείο, το οποίο μας επιτρέπει σε αρκετές περιπτώσεις να καταγράφουμε γνώση με πολύ απλό και κομψό τρόπο. Υπάρχουν διάφορα προβλήματα, όπως αυτά των ασκήσεων που ακολουθούν, στα οποία η αναδρομή είναι πολύ εξυπηρετική, μέχρι άκρως απαραίτητη, θα τολμούσαμε να πούμε, για την έκφραση των αξιωμάτων που τα διέπουν.

Άσκηση 3.4

Έστω ότι μέσω ενός κατηγορήματος `successor/2` έχουμε εκφράσει στην Prolog τη σχέση «επόμενος», για τους μονοψήφιους μη αρνητικούς ακέραιους αριθμούς. Δηλαδή:

```
successor(1, 0).        successor(2, 1).        successor(3, 2).
successor(4, 3).        successor(5, 4).        successor(6, 5).
successor(7, 6).        successor(8, 7).        successor(9, 8).
```

Συμπληρώστε τον παρακάτω ορισμό του κατηγορήματος `greater/2`, στον οποίο ορίζεται, με τη βοήθεια του `successor/2`, πότε ένας μονοψήφιος μη αρνητικός ακέραιος αριθμός είναι μεγαλύτερος από κάποιον άλλο.

```
greater(N1, N2) :- successor(  [        ]  ).
greater(N1, N3) :- successor(N1, N2),  [                ].
```

Άσκηση 3.5

Υλοποιήστε σε Prolog τα αξιώματα της Άσκησης 1.6, για τη σύνδεση δύο κόμβων σε έναν κατευθυνόμενο γράφο χωρίς κύκλους.

3.3 Διαδικασία απόδειξης – Οπισθοδρόμηση

Μέχρι στιγμής έχουμε δει πώς υποβάλλουμε ερωτήσεις σε ένα σύστημα Prolog, για να πάρουμε τις κατάλληλες απαντήσεις, αφού βέβαια έχουμε πρώτα εφοδιάσει το σύστημα με το απαραίτητο πρόγραμμα Prolog, δηλαδή τις προτάσεις που περιγράφουν τη γνώση του κόσμου μας. Δεν έχουμε δει όμως ποιος είναι ο μηχανισμός που χρησιμοποιεί η Prolog για να δώσει απαντήσεις στις ερωτήσεις που υποβάλλουμε. Ωστόσο, ίσως να υποψιάζεστε πώς λειτουργεί αυτή η διαδικασία απόδειξης, με βάση τον τρόπο που εσείς οι ίδιοι θα χρησιμοποιούσατε για να απαντήσετε τις συγκεκριμένες ερωτήσεις. Αντικείμενο αυτής της ενότητας είναι η αναλυτική περιγραφή, με αλγοριθμικό τρόπο, αυτής της διαδικασίας.

Ο μηχανισμός απόδειξης τον οποίο χρησιμοποιεί η Prolog για να απαντήσει σε μια ερώτηση που της υποβάλλεται είναι μια επαναληπτική διαδικασία, η οποία σε κάθε επανάληψη έχει ένα σύνολο από στόχους να ικανοποιήσει. Το αρχικό σύνολο στόχων αποτελείται από αυτούς που περιλαμβάνονται στην ερώτηση η οποία έχει υποβληθεί. Σε κάθε επαναληπτικό βήμα, η Prolog επιλέγει τον πρώτο από τους στόχους που πρέπει να ικανοποιήσει και προσπαθεί να βρει την πρώτη πρόταση στο πρόγραμμα, αυτήν η οποία θα μπορεί να οδηγήσει στην ικανοποίηση του στόχου. Μια πρόταση μπορεί να χρησιμοποιηθεί για την ικανοποίηση ενός στόχου όταν η κεφαλή της «ταιριάζει» με το στόχο. Θυμηθείτε ότι η κεφαλή ενός κανόνα είναι το συμπέρασμα του κανόνα, που γράφουμε αριστερά από το `:-`, ενώ ένα γεγονός είναι ένας κανόνας χωρίς σώμα, δηλαδή χωρίς υποθέσεις, άρα η κεφαλή του είναι

το ίδιο το γεγονός. Η έννοια του «ταιριάσματος» σημαίνει, ουσιαστικά, τη δυνατότητα του στόχου να ταυτιστεί με την κεφαλή, πιθανώς έπειτα και από κατάλληλες αντικαταστάσεις τιμών σε εμπλεκόμενες μεταβλητές. Αυτή η διαδικασία «ταιριάσματος» λέγεται **ενοποίηση** και, για τη μορφή των προγραμμάτων που έχουμε συζητήσει μέχρι τώρα, υπακούει στους ακόλουθους κανόνες:

- Μια σταθερά ενοποιείται με κάποια σταθερά όταν οι δύο σταθερές είναι ίδιες.
- Μια μεταβλητή χωρίς τιμή ενοποιείται με μία σταθερά και παίρνει ως τιμή τη σταθερά αυτή.
- Δύο μεταβλητές ενοποιούνται και αλληλοδεσμεύονται, έτσι ώστε, όταν κάποια από τις δύο πάρει μια τιμή, τότε και η άλλη να πάρει την ίδια τιμή.

Όταν ενοποιηθεί ένας στόχος με την κεφαλή της πρότασης που έχει επιλεγεί από το πρόγραμμα, τότε οι στόχοι του σώματος της πρότασης αυτής, αφού εφαρμοστούν επάνω τους οι αντικαταστάσεις τιμών σε μεταβλητές, λόγω της προηγηθείσας ενοποίησης, γίνονται και αυτοί στόχοι προς ικανοποίηση, και μάλιστα με προτεραιότητα έναντι αυτών που υπήρχαν ήδη εκείνη τη στιγμή. Ουσιαστικά, υπάρχει μια στοίβα από στόχους που πρέπει να ικανοποιηθούν, το άδειασμα της οποίας σημαίνει την ικανοποίηση της αρχικής ερώτησης. Οι στόχοι που προκύπτουν από το σώμα της πρότασης η οποία χρησιμοποιήθηκε για την ικανοποίηση του πρώτου στόχου της στοίβας εισάγονται στην κορυφή της στοίβας.

Ωστόσο, ενδέχεται η Prolog, αφού επιλέξει ένα στόχο για να ικανοποιήσει, να μην μπορέσει να βρει στο πρόγραμμα κάποια πρόταση της οποίας η κεφαλή να ενοποιείται με αυτόν το στόχο. Τότε, ενεργοποιείται ο μηχανισμός της **οπισθοδρόμησης**. Δηλαδή, επιστρέφει το σύστημα στον προηγούμενο χρονολογικά στόχο που είχε ικανοποιήσει, αναιρεί την ενοποίηση μεταξύ του στόχου αυτού και της κεφαλής της πρότασης του προγράμματος που είχε επιλεγεί, και δοκιμάζει την επόμενη πρόταση, με κεφαλή η οποία να ενοποιείται με το στόχο. Έτσι, επιλέγεται ένας εναλλακτικός τρόπος απόδειξης για το στόχο, ενώ η διαδικασία συνεχίζει από εκείνο το σημείο κατά τα γνωστά. Ας σημειωθεί εδώ ότι ο μηχανισμός της οπισθοδρόμησης ενεργοποιείται και στην περίπτωση που ζητούνται περισσότερες της μιας λύσεις. Φυσικά, αν δεν μπορεί να βρεθεί άλλη πρόταση της οποίας η κεφαλή να ενοποιείται με το στόχο στον οποίο έχει οπισθοδρομήσει η Prolog, γίνεται οπισθοδρόμηση ακόμα πιο πριν, στον προηγούμενο χρονολογικά στόχο που είχε ικανοποιηθεί. Αυτή η διαδικασία απόδειξης βασίζεται στην **αρχή της ανάλυσης**, που θα παρουσιάσουμε αναλυτικά στην Ενότητα 7.3. Όπως θα δούμε και στα παραδείγματα που θα ακολουθήσουν, η διαδικασία αυτή δημιουργεί, ουσιαστικά, ένα δέντρο, το **δέντρο ανάλυσης**, το οποίο εξερευνά με έναν εξαντλητικό *πρώτα κατά βάθος* και *από αριστερά προς τα δεξιά* τρόπο.

Ένα τελευταίο σημείο που πρέπει να προσέξουμε αφορά τα ονόματα των μεταβλητών που χρησιμοποιούνται στη διαδικασία της απόδειξης. Κάθε φορά που επιλέγεται μια πρόταση για την ικανοποίηση ενός στόχου, οι μεταβλητές της μετονομάζονται, έτσι ώστε να μη συγχέονται με μεταβλητές που έχουν χρησιμοποιηθεί μέχρι εκείνη τη στιγμή. Αυτό συμβαίνει γιατί οι μεταβλητές της πρότασης είναι εντελώς άσχετες μεταξύ τους κατά τις διαφορετικές φορές στις οποίες θα χρησιμοποιηθεί η πρόταση.

Δείτε όμως ένα παράδειγμα, το οποίο θα σας βοηθήσει να πάρετε μια πρώτη ιδέα για τη διαδικασία απόδειξης στην Prolog:

Παράδειγμα 3.2
Αν στο πρόγραμμα που έχουμε γράψει μέχρι στιγμής για τις οικογενειακές σχέσεις στο κεφάλαιο αυτό υποβάλουμε την ερώτηση:

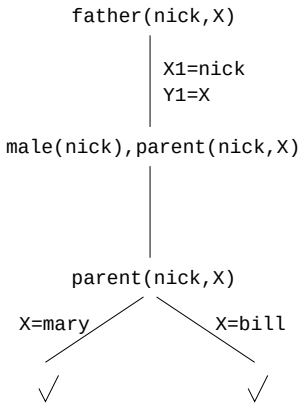
```
?- father(nick, X).
```

τότε η διαδικασία που θα ακολουθήσει η Prolog για να δώσει την κατάλληλη απάντηση θα είναι η εξής:

Δεδομένης της ερώτησης, βρίσκεται η πρώτη (και μόνη) πρόταση του προγράμματος που ορίζει το κατηγόρημα `father/2` και μετασχηματίζεται έτσι ώστε να έχει «φρέσκες» μεταβλητές, έστω στην:

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

Ο μοναδικός στόχος της ερώτησης ενοποιείται με την κεφαλή της πρότασης, κάνοντας και τις αναθέσεις `X1 = nick` και `Y1 = X`. Οπότε, έχουμε δύο νέους στόχους προς ικανοποίηση, τους `male(X1)` και `parent(X1, Y1)`, οι οποίοι, λόγω των αναθέσεων, γίνονται `male(nick)` και `parent(nick, X)`. Για τον πρώτο από τους στόχους αυτούς βρίσκεται πρόταση στο πρόγραμμα, το γεγονός «`male(nick).`», που τον ικανοποιεί. Για τον δεύτερο στόχο, η πρώτη πρόταση της οποίας η κεφαλή ενοποιείται με αυτόν είναι το γεγονός «`parent(nick, mary).`», δίνοντας και την ανάθεση `X = mary`, που αποτελεί την πρώτη απάντηση στην αρχική ερώτηση. Αλλά, ο στόχος `parent(nick, X)` μπορεί να ικανοποιηθεί, έπειτα από οπισθοδρόμηση, και από το γεγονός «`parent(nick, bill).`». Οπότε, παίρνουμε και τη δεύτερη απάντηση στην αρχική ερώτηση, την `X = bill`. Όλη αυτή η διαδικασία απόδειξης φαίνεται συνοπτικά στο δέντρο ανάλυσης του Σχήματος 3.2. □



Σχήμα 3.2: Δέντρο ανάλυσης για την ερώτηση «`?- father(nick, X).`».

Στη συνέχεια, παρατίθεται αναλυτική περιγραφή του αλγορίθμου που ακολουθεί η Prolog για να δώσει τις απαντήσεις σε μια ερώτηση η οποία της υποβάλλεται. Στον αλγόριθμο αυτό, κάθε στιγμή στο **G** βρίσκεται ο τρέχων στόχος προς ικανοποίηση, ενώ στη στοίβα **GS** βρίσκονται οι υπόλοιποι στόχοι. Στο **VL** περιλαμβάνονται οι μέχρι στιγμής αναθέσεις τιμών που έχουν γίνει σε μεταβλητές. Το **SS** είναι μια στοίβα η οποία χρειάζεται για να υπάρχει η δυνατότητα οπισθοδρόμησης σε προηγούμενες καταστάσεις, έτσι ώστε να μπορεί να ικανοποιηθεί ένας παρελθών στόχος και με εναλλακτικούς τρόπους. Στη στοίβα αυτή εισάγονται (και εξάγονται) τετράδες (**G**, **C**, **GS**, **VL**), καθεμία από τις οποίες δηλώνει ότι υπάρχει εναλλακτική δυνατότητα ικανοποίησης του στόχου **G** με την πρόταση **C** και ότι εκκρεμούν για ικανοποίηση οι στόχοι στη στοίβα **GS**, ενώ οι ισχύουσες αναθέσεις τιμών σε μεταβλητές είναι αυτές που περιέχονται στη λίστα **VL**.

Υπολογισμός απαντήσεων στην ερώτηση **Q**

Βήμα 1: *Αρχικοποιήστε τη στοίβα από στόχους προς ικανοποίηση* **GS***, θέτοντας*

σε αυτήν τους στόχους της αρχικής ερώτησης **Q**.

Βήμα 2: *Αρχικοποιήστε τη λίστα αναθέσεων μεταβλητών* **VL** *σαν μια κενή λίστα.*

Βήμα 3: *Αρχικοποιήστε τη στοίβα καταστάσεων* **SS** *σαν μια κενή στοίβα.*

Βήμα 4: *Αν η* **GS** *είναι κενή, τότε βρέθηκε λύση. Εκτυπώστε τις τιμές των μεταβλητών της αρχικής ερώτησης* **Q** *και, αν ζητήθηκαν και άλλες λύσεις, πηγαίνετε στο βήμα 7, αλλιώς τερματίστε.*

Βήμα 5: *Βγάλτε από την* **GS** *τον πρώτο στόχο και κάντε τον τρέχοντα στόχο προς ικανοποίηση, θέτοντάς τον στο* **G**.

Βήμα 6: *Θέστε στο* **C** *την πρώτη πρόταση* **H** :- **B1**, **B2**, . . ., **Bn** *του προγράμματος με την κεφαλή της οποίας* **H** *ενοποιείται ο στόχος* **G** *και πηγαίνετε στο βήμα 8, αλλιώς, αν δεν υπάρχει τέτοια πρόταση, συνεχίστε στο βήμα 7.*

Βήμα 7: *Αν η στοίβα* **SS** *είναι κενή, τερματίστε εκτυπώνοντας το μήνυμα ότι δεν υπάρχουν (άλλες) απαντήσεις, αλλιώς βγάλτε από τη στοίβα* **SS** *την τετράδα* (**G**, **C**, **GS**, **VL**) *από την κορυφή της και θέστε τα στοιχεία της τετράδας στα* **G**, **C**, **GS** *και* **VL**.

Βήμα 8: *Αν υπάρχει και επόμενη πρόταση* **CN**, *τέτοια ώστε η κεφαλή της να ταιριάζει με το στόχο* **G**, *τότε βάλτε στην κορυφή της στοίβας* **SS** *την τετράδα* (**G**, **CN**, **GS**, **VL**).

Βήμα 9: *Μετονομάστε τις μεταβλητές της πρότασης* **C**, *έτσι ώστε να έχετε μια νέα πρόταση* $\mathbf{C}'$ ($\equiv$ $\mathbf{H}'$:- $\mathbf{B1}'$, $\mathbf{B2}'$, . . ., $\mathbf{Bn}'$) *με νέες μεταβλητές.*

Βήμα 10: *Ενοποιήστε το* **G** *με το* $\mathbf{H}'$ *και τις προκύπτουσες αναθέσεις τιμών σε μεταβλητές αφενός προσθέστε τες στο* **VL** *και αφετέρου εφαρμόστε τες στο σώμα της πρότασης* $\mathbf{C}'$ *και βάλτε τους νέους στόχους* $\mathbf{B1}''$, $\mathbf{B2}''$, . . ., $\mathbf{Bn}''$ *στην κορυφή της* **GS**.

Βήμα 11: *Πηγαίνετε στο βήμα 4.*

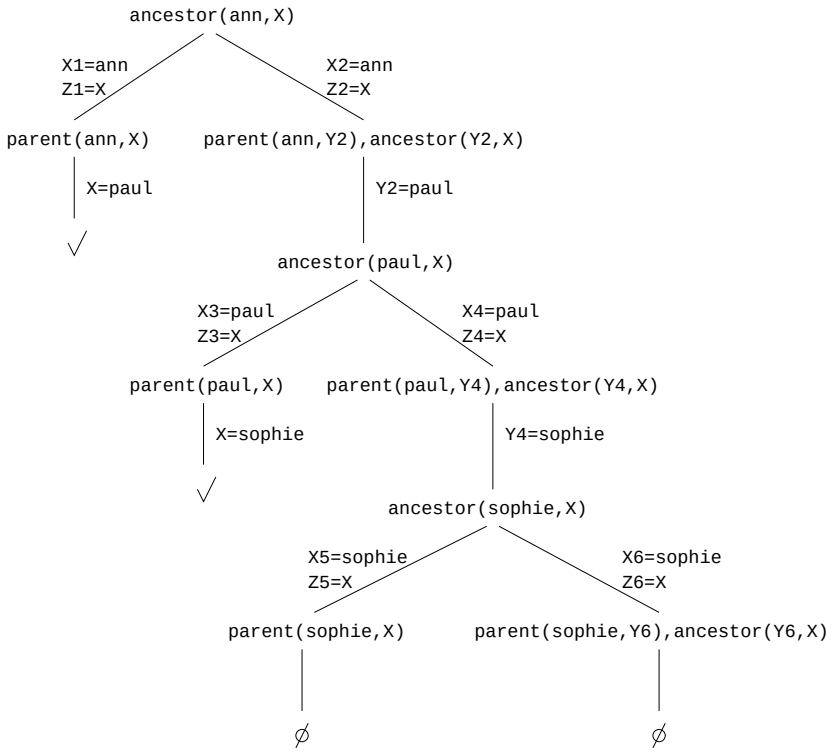
Ας δούμε όμως και άλλο ένα παράδειγμα, λίγο πιο πολύπλοκο αυτήν τη φορά, για να κάνουμε περισσότερο φανερή τη διαδικασία απόδειξης της Prolog:

Παράδειγμα 3.3
Αν στο γνωστό μας γενεαλογικό πρόγραμμα υποβάλουμε την ερώτηση:

```
?- ancestor(ann, X).
```

τότε η Prolog, για να υπολογίσει τις ζητούμενες απαντήσεις, θα εξερευνήσει το δέντρο ανάλυσης του Σχήματος 3.3. Όπως βλέπουμε στο σχήμα αυτό, κάθε κόμβος του δέντρου στον οποίο ο πρώτος στόχος αναφέρεται στο κατηγόρημα `ancestor/2` έχει δύο απογόνους, οι οποίοι αντιστοιχούν στους δύο εναλλακτικούς τρόπους ικανοποίησης του στόχου, λόγω των δύο προτάσεων που υπάρχουν για τον ορισμό του `ancestor/2`. Επίσης, στο σχήμα φαίνονται και οι κλάδοι του δέντρου που οδηγούν σε ικανοποίηση της αρχικής ερώτησης, για τους οποίους οι αντίστοιχες απαντήσεις είναι `X = paul` και `X = sophie`.

Σ' αυτό το σημείο πρέπει να τονίσουμε ότι, ακριβώς λόγω του αλγορίθμου τον οποίο χρησιμοποιεί η Prolog, για να υπολογίζει τις απαντήσεις σε ερωτήσεις που υποβάλλονται, δεδομένου κάποιου προγράμματος, έχουν πολύ μεγάλη σημασία τόσο η σειρά με την οποία βρίσκονται μέσα στο πρόγραμμα οι προτάσεις που ορίζουν ένα κατηγόρημα, όσο και η σειρά με την οποία οι στόχοι βρίσκονται στα σώματα των κανόνων. Αυτό πιθανώς να σας ξενίζει λίγο, αφού κάθε πρόταση είναι ουσιαστικά μια συνεπαγωγή. Με άλλα λόγια, δεν θα έπρεπε να παίζει ρόλο η σειρά με την οποία διατυπώνουμε τις συνεπαγωγές που καταλήγουν



Σχήμα 3.3: Δέντρο ανάλυσης για την ερώτηση «`?- ancestor(ann, X).`».

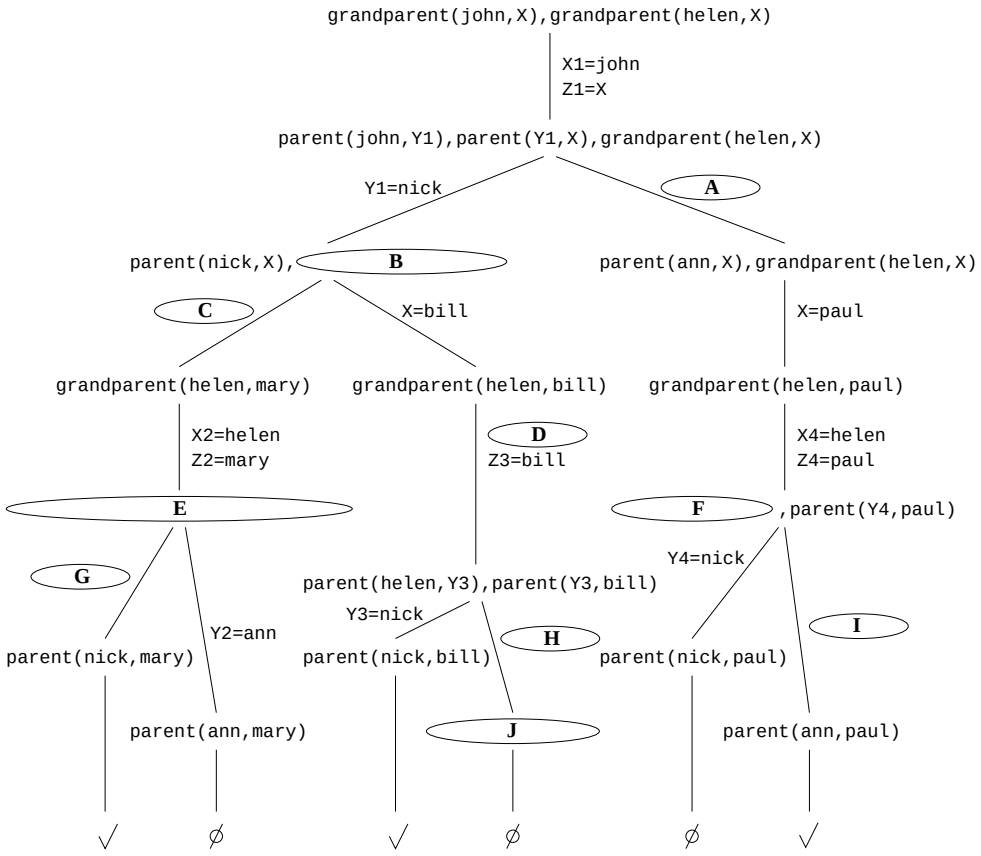
σε συμπεράσματα για το ίδιο κατηγόρημα. Επίσης, επειδή οι στόχοι στα σώματα των κανόνων αποτελούν τα συστατικά των συζεύξεων, δηλαδή τις υποθέσεις των κανόνων, είναι βέβαιο ότι δύσκολα μπορούμε να δεχτούμε πως η σειρά με την οποία γράφονται οι συζευκτέοι είναι σημαντική. Παρ' όλα αυτά όμως, όντως έτσι συμβαίνει και πρέπει να το έχουμε υπόψη μας, έτσι ώστε να γράφουμε σωστά προγράμματα Prolog. Όταν βλέπουμε ένα πρόγραμμα Prolog ως ένα σύνολο από τύπους της λογικής, τότε αναφερόμαστε στη δηλωτική σημασία του. Ωστόσο, πραγματικά πρέπει να μας ενδιαφέρει η διαδικαστική σημασία του, όπως προσδιορίζεται από τον τρόπο υπολογισμού των απαντήσεων σε υποβαλλόμενες ερωτήσεις, σύμφωνα με τον αλγόριθμο που παρουσιάστηκε προηγουμένως. Γι' αυτόν το λόγο, θα κλείσουμε την ενότητα με κάποιες ασκήσεις, που πιστεύουμε ότι θα σας βοηθήσουν να κατανοήσετε πλήρως τον συγκεκριμένο αλγόριθμο.

Άσκηση 3.6
Στο Σχήμα 3.4 είναι σχεδιασμένο το δέντρο ανάλυσης για τη διαδικασία υπολογισμού των απαντήσεων στην ερώτηση:

```
?- grandparent(john, X), grandparent(helen, X).
```

με δεδομένο το πρόγραμμα που έχουμε αναπτύξει σταδιακά στο Κεφάλαιο 3. Το δέντρο αυτό όμως είναι ελλιπές, όσον αφορά κάποιους στόχους στους κόμβους του, αλλά και τις αναθέσεις τιμών που καταγράφουμε στις ακμές του. Τι πρέπει να συμπληρώσουμε στις θέσεις **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**, **I** και **J**, για να είναι πλήρες το δέντρο; Ποιες είναι οι απαντήσεις τελικά στην ερώτηση;

Άσκηση 3.7
Σχεδιάστε το δέντρο ανάλυσης για την ερώτηση:



Σχήμα 3.4: Ελλιπές δέντρο ανάλυσης.

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

θεωρώντας ότι ο ορισμός του `sister/2` είναι αυτός που δεν έχει το στόχο `diff/2` στο σώμα του κανόνα, δηλαδή ο:

```
sister(X, Y) :- female(X), parent(Z, X), parent(Z, Y).
```

και τα γεγονότα για τα `female/1` και `parent/2` είναι αυτά που έχουν ήδη αναφερθεί.

3.4 Εφαρμογές της Prolog

Από τις προηγούμενες ενότητες αυτού του κεφαλαίου θα διαπιστώσατε ότι η γλώσσα προγραμματισμού Prolog παρουσιάζει κάποιες ιδιαιτερότητες, σε σχέση με τις παραδοσιακές διαδικαστικές γλώσσες προγραμματισμού. Ίσως αναρωτηθήκατε αν είναι δυνατόν αυτή η ιδιόμορφη φιλοσοφία της να είναι χρήσιμη στην υλοποίηση πρακτικών εφαρμογών από τον πραγματικό κόσμο και, μάλιστα, αν υπάρχουν περιοχές εφαρμογών στις οποίες η Prolog να είναι η ενδεδειγμένη λύση. Πράγματι, υπάρχει πληθώρα τέτοιων περιοχών, στις οποίες θα αναφερθούμε περιληπτικά στην ενότητα αυτή.

3.4.1 Έμπειρα συστήματα

Πρόκειται για εφαρμογές που προσομοιώνουν τη συμπεριφορά ενός έμπειρου/ειδικού σε κάποιον τομέα. Για παράδειγμα, θα μπορούσαμε να υλοποιήσουμε ένα έμπειρο σύστημα για τη διάγνωση καρδιακών παθήσεων και την πρόταση θεραπευτικής αγωγής, ένα άλλο

σύστημα για τον εντοπισμό βλαβών σε αυτοκίνητα και τη σύσταση κατάλληλης διαδικασίας για την αποκατάστασή τους, ένα έμπειρο σύστημα παροχής επενδυτικών συμβουλών για το χρηματιστήριο κ.ο.κ. Τα έμπειρα συστήματα, για να εκτελέσουν με επιτυχία την εργασία στην οποία ειδικεύονται, είναι εφοδιασμένα με την απαραίτητη γνώση, την οποία επεξεργάζεται η «μηχανή συμπερασμού» τους, προκειμένου να μπορέσουν να δώσουν τις κατάλληλες συμβουλές στους χρήστες τους. Η γνώση των έμπειρων συστημάτων είναι, κατά κανόνα, διατυπωμένη στη μορφή **if-then** κανόνων. Για το λόγο αυτό, η Prolog, επειδή είναι μια γλώσσα που υποστηρίζει τη διατύπωση και διαχείριση γνώσης αυτής της μορφής, ενδείκνυται για την υλοποίηση έμπειρων συστημάτων.

3.4.2 Κατανόηση φυσικής γλώσσας

Πολλές εφαρμογές έχουν βασικό ή δευτερεύοντα στόχο τους να φέρουν σε πέρας την αυτόματη κατανόηση είτε κάποιου κειμένου, περιορισμένου ή περισσότερου εκτεταμένου, είτε έστω κάποιας πρότασης, από κάποια συγκεκριμένη φυσική γλώσσα, όπως τα ελληνικά, τα αγγλικά ή οποιαδήποτε άλλη. Τέτοιες είναι οι εφαρμογές που δέχονται για τη διεπαφή με τους χρήστες τους κάποιο υποσύνολο φυσικής γλώσσας ή ακόμα και οι εφαρμογές των οποίων βασική εργασία είναι η κατανόηση φυσικής γλώσσας, για να επιτύχουν τον επιδιωκόμενο σκοπό τους, όπως τα συστήματα αυτόματης μετάφρασης από μια φυσική γλώσσα σε μια άλλη, τα συστήματα αυτόματης εξαγωγής περιλήψεων από κείμενα κ.ά. Για την επιτυχή κατανόηση μιας φυσικής γλώσσας, είναι απαραίτητο να έχουν κωδικοποιηθεί τόσο η γλωσσολογική γνώση όσο και η γνώση του κόσμου στον οποίο αναφέρεται το κείμενο ή οι προτάσεις για κατανόηση. Όσον αφορά και τα δύο είδη γνώσης, η Prolog προσφέρεται για τις ανάγκες τόσο της αναπαράστασης όσο και της διαχείρισης. Για το λόγο αυτό, έχει χρησιμοποιηθεί στο παρελθόν στην υλοποίηση μεγάλου αριθμού εφαρμογών στην περιοχή της κατανόησης φυσικής γλώσσας.

3.4.3 Προβλήματα αναζήτησης

Τα προβλήματα αναζήτησης συνιστούν μια μεγάλη κατηγορία προβλημάτων από την τεχνητή νοημοσύνη, στα οποία, με δεδομένα την αρχική κατάσταση του προβλήματος και το σύνολο των νόμιμων μεταβάσεων από μια κατάσταση σε μια άλλη, το ζητούμενο είναι να βρεθεί με ποια αλληλουχία από μεταβάσεις μεταξύ καταστάσεων μπορούμε να καταλήξουμε σε μια τελική κατάσταση, είτε ρητά διατυπωμένη είτε, απλώς, με γνωστές ιδιότητες, ξεκινώντας από την αρχική κατάσταση του προβλήματος. Τέτοια είναι τα προβλήματα της κατάστρωσης σχεδίου, στα οποία πρέπει να βρεθούν οι ενέργειες που μπορούν να επιτύχουν ένα στόχο, όταν εκτελεστούν (π.χ. πώς θα πρέπει να τακτοποιηθεί ένα ακατάστατο δωμάτιο), τα προβλήματα της χρονοδρομολόγησης, στα οποία ένα πλήθος από αλληλοεξαρτώμενες δραστηριότητες πρέπει να τοποθετηθεί στον άξονα του χρόνου (π.χ. κατασκευή του ωρολόγιου προγράμματος μαθημάτων ενός σχολείου), καθώς και τα προβλήματα ανάθεσης πόρων, στα οποία πρέπει, χωρίς παραβίαση εμπλεκόμενων περιορισμών, να ανατεθεί ένα σύνολο από πόρους σε ένα σύνολο από δραστηριότητες (π.χ. ανάθεση του ιπτάμενου προσωπικού στις πτήσεις μιας αεροπορικής εταιρείας).

Στα προβλήματα αναζήτησης, πολύ συχνά οι πιθανές καταστάσεις που μπορούν να ακολουθήσουν μια κατάσταση είναι περισσότερες της μιας, χωρίς να γνωρίζουμε σε αυτήν τη φάση ποια έχει προοπτικές να μας οδηγήσει σε λύση του προβλήματος. Το γεγονός αυτό, σε συνάρτηση και με το δεδομένο ότι το μοντέλο υπολογισμού στην Prolog βασίζεται στην εξερεύνηση δέντρων ανάλυσης, σε κάθε κόμβο των οποίων ακολουθείται ένα μονοπάτι, αλλά επιλέγεται κάποιο άλλο σε περίπτωση αδιεξόδου, μέσω οπισθοδρόμησης, θα πρέπει

να μας κάνει να συμπεράνουμε ότι η Prolog είναι η ιδανική γλώσσα για την αντιμετώπιση προβλημάτων αναζήτησης. Όντως, αυτό είναι αλήθεια. Το μόνο πρόβλημα εντοπίζεται στην αποδοτικότητα των εφαρμογών που υλοποιούμε σε Prolog για την επίλυση προβλημάτων αναζήτησης. Αν ο χώρος αναζήτησης ενός προβλήματος είναι σχετικά μεγάλος (πολλά βήματα από την αρχική στην τελική κατάσταση και πολλές εναλλακτικές διάδοχες καταστάσεις από τη μια στην άλλη), η εξαντλητική αναζήτηση λύσης στο χώρο αυτό ίσως να μην είναι επιτυχής, μέσα σε αποδεκτά χρονικά πλαίσια. Οπότε, αυτό που πρέπει να κάνουμε είναι να εμπεδώσουμε στην εφαρμογή ευρετική γνώση για το συγκεκριμένο πρόβλημα, με στόχο να πάρουμε με έξυπνο τρόπο τις σωστές αποφάσεις όταν διασχίζουμε το χώρο αναζήτησης. Επίσης, μπορούμε να καταφύγουμε σε μια επέκταση της Prolog, τον λογικό προγραμματισμό με περιορισμούς, που προσφέρει πολλές δυνατότητες αποδοτικής εξερεύνησης ενός χώρου αναζήτησης, και στον οποίο θα αναφερθούμε με περισσότερες λεπτομέρειες στο Κεφάλαιο 6. Η επέκταση αυτή χρησιμοποιείται πάρα πολύ συχνά, για την αντιμετώπιση προβλημάτων από τον πραγματικό κόσμο.

3.4.4 Απόδειξη θεωρημάτων

Πολλές φορές μας ενδιαφέρει να υλοποιήσουμε ένα σύστημα, έτσι ώστε, αν το εφοδιάσουμε με ένα σύνολο αξιωμάτων, να μπορεί να αποδεικνύει αυτόματα ό,τι είναι δυνατόν να προκύψει λογικά από τα αξιώματα, δηλαδή τα πιθανά θεωρήματα που στηρίζονται στα αξιώματα αυτά. Η ορολογία που χρησιμοποιείται πιθανώς να σας παραπέμπει στη διαδικασία απόδειξης θεωρημάτων στα μαθηματικά (π.χ. άλγεβρα, γεωμετρία κτλ.). Πράγματι, είναι πολύ καλή ιδέα, σε συγκεκριμένους κλάδους των μαθηματικών, να διατυπώνουμε τα αξιώματα που ισχύουν και να έχουμε ένα σύστημα με το οποίο να μπορούμε να αποδεικνύουμε θεωρήματα στον κλάδο αυτό, είτε ήδη γνωστά είτε –και αυτό έχει πολύ μεγάλο ενδιαφέρον– θεωρήματα που πιθανώς δεν έχουμε σκεφτεί μέχρι τώρα να αποδείξουμε. Όντως, έχουν υλοποιηθεί στο παρελθόν τέτοιου είδους εφαρμογές, τόσο στα μαθηματικά, όσο και σε άλλες περιοχές, που έχουν δεδομένη αξιωματική θεμελίωση. Η Prolog, ως γλώσσα που κατεξοχήν υποστηρίζει αυτό το μοντέλο υπολογισμού, δηλαδή τη διατύπωση αξιωμάτων και την αυτόματη εξαγωγή συμπερασμάτων, είναι ιδανική για την υλοποίηση τέτοιων εφαρμογών.

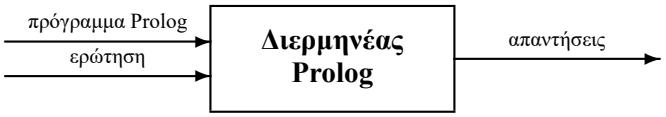
3.4.5 Συμβολική επεξεργασία

Η τελευταία περιοχή εφαρμογών της Prolog στην οποία θα αναφερθούμε είναι η συμβολική επεξεργασία. Ο συγκεκριμένος όρος αφορά όλες εκείνες τις εφαρμογές που επιτυγχάνουν το στόχο τους μέσω της διαχείρισης και της επεξεργασίας συμβόλων. Αυτές τις εφαρμογές πρέπει σαφώς να τις αντιδιαστείλουμε με εκείνες τις εφαρμογές που κάνουν, κατά βάση, αριθμητική επεξεργασία. Παρότι, όπως θα δούμε σε επόμενο κεφάλαιο, έχει και η Prolog κάποιες στοιχειώδεις αριθμητικές δυνατότητες, δεν είναι η καταλληλότερη γλώσσα για να υλοποιήσουμε εφαρμογές των οποίων οι ανάγκες σε αριθμητική επεξεργασία είναι μεγάλες. Αντίθετα, για εφαρμογές επεξεργασίας συμβόλων, παρέχει μια σειρά από δυνατότητες που δεν βρίσκουμε εύκολα σε άλλες γλώσσες προγραμματισμού. Ενδεικτικά, ως εφαρμογές που χρειάζονται συμβολική επεξεργασία, μπορούμε να αναφέρουμε τα συστήματα συμβολικής παραγώγισης και συμβολικής ολοκλήρωσης, τα συστήματα συμβολικής (όχι αριθμητικής) επίλυσης εξισώσεων και συστημάτων από εξισώσεις, τα συστήματα απλοποίησης πολυωνύμων και, γενικότερα, μαθηματικών τύπων κτλ.

3.5 Υλοποίηση συστημάτων Prolog

Όπως ισχύει για τη συντριπτική πλειονότητα των γλωσσών προγραμματισμού, έτσι και για την Prolog, υπάρχουν δύο προσεγγίσεις υλοποίησής της, είτε μέσω ενός διερμηνέα είτε μέσω ενός μεταγλωττιστή [4].

Ένας διερμηνέας Prolog είναι ένα πρόγραμμα που δέχεται στην είσοδό του ένα πρόγραμμα Prolog και μια ερώτηση που απευθύνεται σε αυτό και υπολογίζει, εφαρμόζοντας την αρχή της ανάλυσης, τις ζητούμενες απαντήσεις, όπως φαίνεται στο Σχήμα 3.5.



Σχήμα 3.5: Δομή συστήματος Prolog βασισμένου σε διερμηνέα.

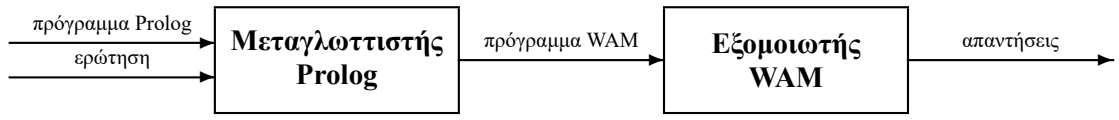
Ο διερμηνέας κωδικοποιεί το πρόγραμμα Prolog σαν μια δομή δεδομένων στην περιοχή προγράμματος και χρησιμοποιεί μια στοίβα, για να διαχειριστεί το δέντρο ανάλυσης που κατασκευάζεται κατά την προσπάθεια απάντησης της ερώτησης. Στη στοίβα φυλάσσονται οι πληροφορίες που απαιτούνται για την υποστήριξη, εκτός των άλλων, και της δυνατότητας οπισθοδρόμησης.

Σε κάθε κόμβο του δέντρου ανάλυσης, που αντιστοιχεί σε μια εγγραφή της στοίβας, κωδικοποιείται η ενοποίηση ενός στόχου με την κεφαλή μιας πρότασης και δημιουργείται ένα περιβάλλον, στο οποίο, για κάθε μεταβλητή της επιλεγμένης πρότασης, υπάρχει διαθέσιμος χώρος να καταχωρηθεί η διεύθυνση της τιμής της. Επειδή μια μεταβλητή κάποιου περιβάλλοντος μπορεί να πάρει τιμή σε μεταγενέστερο κόμβο από αυτόν της δημιουργίας της, πρέπει να υπάρχει η δυνατότητα αναίρεσης της απόδοσης της τιμής, αν γίνει οπισθοδρόμηση πριν από το σημείο της απόδοσης. Αυτό επιτυγχάνεται μέσω των λεγόμενων ιχνών, που είτε καταχωρούνται μέσα σε περιβάλλοντα είτε συνιστούν χωριστή στοίβα.

Τέλος, η τιμή μιας μεταβλητής μπορεί να παρασταθεί είτε από κάποιον δείκτη σε μια δομή στην περιοχή προγράμματος και σε ένα περιβάλλον στο οποίο θα βρεθούν οι τιμές των μεταβλητών της δομής είτε από ένα δείκτη σε μια περιοχή, που ονομάζεται σωρός, στην οποία έχει κτιστεί η τιμή της μεταβλητής. Δηλαδή, η πρώτη προσέγγιση βασίζεται στην κοινή χρήση δομών, ενώ η δεύτερη στην αντιγραφή δομών.

Μια δεύτερη τεχνική υλοποίησης συστημάτων Prolog βασίζεται σε ένα μεταγλωττιστή. Ο μεταγλωττιστής είναι ένα πρόγραμμα που δέχεται στην είσοδό του ένα πρόγραμμα Prolog, το οποίο μεταφράζει σε μια ειδική γλώσσα χαμηλού επιπέδου (τύπου «assembly»), τη WAM, της λεγόμενης **αφηρημένης μηχανής του Warren** (Warren Abstract Machine) [5, 6]. Όταν απευθύνεται μια ερώτηση στο πρόγραμμα Prolog, αυτή μεταγλωττίζεται σε WAM. Το συνολικό πρόγραμμα που προκύπτει από τη μεταγλώττιση του προγράμματος Prolog και της υποβληθείσας ερώτησης δίνεται στην είσοδο ενός άλλου προγράμματος, που λέγεται **εξομοιωτής WAM**, το οποίο, μετά την απαραίτητη εκτέλεση, υπολογίζει τις ζητούμενες απαντήσεις. Η όλη διαδικασία φαίνεται στο Σχήμα 3.6.

Η μηχανή WAM μπορεί να περιγραφεί από την αρχιτεκτονική της και το σύνολο εντολών της. Οι εντολές διαχειρίζονται τη μνήμη της μηχανής και τους καταχωρητές της. Η μνήμη είναι διαμερισμένη σε χώρους αντίστοιχους μ' αυτούς που συναντάμε σε διερμηνείς (περιοχή προγράμματος, στοίβα, σωρός, ίχνη κτλ.), αν και με σημαντικά διαφορετικό τρόπο καταχώρησης των σχετικών πληροφοριών.



Σχήμα 3.6: Δομή συστήματος Prolog βασισμένου σε μεταγλωττιστή.

Τα συστήματα Prolog που είχαν υλοποιηθεί κατά τα αρχικά στάδια εμφάνισης της γλώσσας ήταν διερμηνείς. Αυτός ήταν ο προφανής τρόπος για την κατασκευή ενός πρακτικού συστήματος που να δουλεύει και να επιδεικνύει τις δυνατότητες της γλώσσας. Σχετικά σύντομα όμως, ο Warren συνέλαβε και πρότεινε την εξαιρετικά πρωτοποριακή ιδέα της αφηρημένης μηχανής του, και έκτοτε όλα τα «σοβαρά» συστήματα Prolog βασίζονται σε μεταγλωττιστές και στη WAM. Μάλιστα, πολλοί ερευνητές έχουν επεκτείνει τη WAM προς διάφορες κατευθύνσεις, με σκοπό την υλοποίηση χρήσιμων επεκτάσεων της Prolog.

Απαντήσεις ασκήσεων

Απάντηση άσκησης 3.1
Η σωστή αντιστοιχία ερωτήσεων και απαντήσεων είναι: 1-2, 2-2, 3-5, 4-2, 5-2, 6-10, 7-1, 8-2, 9-7 και 10-4. Ας σχολιάσουμε λίγο τις απαντήσεις:

1. Η απάντηση είναι `no`, γιατί το `parent(mary, nick)` δεν γνωρίζουμε αν ισχύει, παρότι το `parent(nick, mary)` είναι αληθές. Η σειρά έχει σημασία.
2. Δεν γνωρίζουμε αν ο *Ιάκωβος* έχει παιδιά.
3. Είναι ρητά δηλωμένο ότι ισχύει.
4. Δεν γνωρίζουμε αν υπάρχει κάποιος που είναι γονιός του εαυτού του.
5. Δεν γνωρίζουμε αν ο *Μάρκος* είναι γονιός του *Θωμά*. Για την ακρίβεια, δεν γνωρίζουμε τίποτα γι' αυτούς τους δύο. Αυτό δεν σημαίνει ότι δεν θα πρέπει η απάντηση στην ερώτησή μας να μην είναι `no`.
6. Οι γονείς της *Μαίρης* είναι ο *Νίκος* και η *Μαργαρίτα*.
7. Τα παιδιά της *Μαργαρίτας* και του *Νίκου* είναι η *Μαίρη* και ο *Βασίλης*.
8. Δεν γνωρίζουμε αν υπάρχει κάποιος που είναι γονιός κάποιου γονιού του.
9. Για την ερώτηση `parent(X, paul)`, υπάρχουν δύο απαντήσεις, οι `X = chris` και `X = ann`, ενώ για την `parent(bill, Y)`, υπάρχει η απάντηση `Y = peter`. Δεν έχει σημασία το γεγονός ότι οι μεταβλητές των δύο στόχων δεν σχετίζονται. Η τελική απάντηση προκύπτει από το καρτεσιανό γινόμενο των δύο συνόλων λύσεων.
10. Στη γενεαλογική αλυσίδα από την *Ελένη* προς τον *Πέτρο* βρίσκονται ο *Νίκος* και ο *Βασίλης*.

Ένα τελευταίο σχόλιο θα πρέπει να κάνουμε σχετικά με την τρίτη υποψήφια απάντηση. Συχνά, πιστεύουμε λανθασμένα ότι σε ιδιόρρυθμες ερωτήσεις, όπως είναι η τέταρτη, η πέμπτη ή η όγδοη, θα δοθεί απάντηση κάποιο μήνυμα λάθους, αντί για μια καθαρή αρνητική απάντηση. Αυτό δεν είναι σωστό. Οποιαδήποτε ερώτηση και αν υποβάλουμε στο πρόγραμμά μας με το κατηγόρημα `parent/2`, δεν υπάρχει περίπτωση να πάρουμε τέτοια απάντηση.

Απάντηση άσκησης 3.2
Η σωστή απάντηση είναι η 2. Η άσκηση αυτή ήταν λίγο «πονηρή». Η απάντηση 3 δεν είναι σωστή, γιατί ο *Νίκος* έχει αδελφή, την *Άννα*. Η απάντηση 1 δεν είναι μεγάλο λάθος, γιατί, πράγματι, η λύση `X = ann` είναι σωστή. Η απάντηση 2, όμως, είναι πιο σωστή, γιατί η λύση `X = ann` θα δοθεί δύο φορές. Αυτό οφείλεται στον ορισμό που έχουμε δώσει για το κατηγόρημα `sister/2`. Θυμηθείτε ότι είχαμε δηλώσει: «*αδελφή κάποιου είναι μια γυναίκα που έχει κοινό γονιό με αυτόν*». Βρίσκουμε πως η *Άννα* είναι δύο φορές αδελφή του *Νίκου*, επειδή έχει δύο κοινούς γονιούς με αυτόν, τον *Γιάννη* και την *Ελένη*. Πράγματι, αυτό δεν θα θέλαμε να συμβαίνει, γιατί δεν έχει λογική βάση, αλλά προς το παρόν δεν μπορούμε να κάνουμε τίποτα για να το διορθώσουμε.

Απάντηση άσκησης 3.3
Οι ζητούμενοι ορισμοί θα μπορούσαν να συμπληρωθούν ως εξής:

1. «*Μητέρα κάποιου είναι μια γυναίκα που είναι γονιός του.*»
   `mother(X, Y) :- female(X), parent(X, Y).`
2. «*Αδελφός κάποιου είναι ένας άνδρας που έχει κοινό γονιό με αυτόν.*»
   `brother(X, Y) :- male(X), parent(Z, X), parent(Z, Y).`
3. «*Θείος κάποιου είναι ο αδελφός ενός γονιού του.*»
   `uncle(X, Y) :- parent(Z, Y), brother(X, Z).`
4. «*Θεία κάποιου είναι η αδελφή ενός γονιού του.*»
   `aunt(X, Y) :- parent(Z, Y), sister(X, Z).`
5. «*Εγγόνι κάποιου είναι αυτός/αυτή που τον έχει παππού/γιαγιά.*»
   `grandchild(X, Y) :- grandparent(Y, X).`
6. «*Παππούς κάποιου είναι ο άνδρας γονιός των γονιών του.*»
   `grandfather(X, Y) :- male(X), grandparent(X, Y).`
7. «*Γιαγιά κάποιου είναι η γυναίκα γονιός των γονιών του.*»
   `grandmother(X, Y) :- female(X), grandparent(X, Y).`
8. «*Γιος κάποιου είναι ένας άνδρας που τον έχει γονιό.*»
   `son(X, Y) :- male(X), parent(Y, X).`
9. «*Κόρη κάποιου είναι μια γυναίκα που τον έχει γονιό.*»
   `daughter(X, Y) :- female(X), parent(Y, X).`
10. «*Κάποιος έχει και γιο και κόρη όταν υπάρχουν άτομα που έχουν αυτήν την ιδιότητα.*»
    `hbsad(X) :- son(Y, X), daughter(Z, X).`

Οι ορισμοί για τα κατηγορήματα `uncle/2` και `aunt/2` δεν είναι απόλυτα σωστοί. Αυτό συμβαίνει επειδή τα κατηγορήματα `brother/2` και `sister/2` δεν είναι απόλυτα σωστά. Είδαμε ότι, αν στο σώμα του κανόνα για το `sister/2`, αλλά και του κανόνα για το `brother/2`, δεν προσθέσουμε και το στόχο `diff(X, Y)`, τότε αποδεικνύεται ότι κάποια είναι αδελφή του εαυτού της, αλλά και ότι κάποιος είναι αδελφός του εαυτού του. Αυτό το λάθος επηρεάζει την ορθότητα των ορισμών για τα `uncle/2` και `aunt/2`, γιατί επιτρέπει την απόδειξη ότι κάποιος έχει θείο τον πατέρα του και θεία τη μητέρα του, αφού ο πατέρας του μπορεί να είναι αδελφός του εαυτού του και η μητέρα του αδελφή του εαυτού της. Έτσι, έχουμε το παρακάτω παράδοξο:

```
?- uncle(nick, bill).
yes
?-
```

Δηλαδή, ο *Νίκος* είναι πατέρας του *Βασίλη*, αλλά είναι και θείος του!

Δείτε και κάποια άλλα παραδείγματα ερωτήσεων, μαζί με τις απαντήσεις τους, στους κανόνες που γράψαμε σε αυτήν την άσκηση:

```
?- mother(X, paul).
X = ann        ;
no
?- brother(X, alice).
X = george     ;
X = jack       ;
no
?- uncle(mary, X).
no
?- aunt(mary, peter).
yes
?- grandchild(sophie, X).
X = chris      ;
X = ann        ;
no
?- grandfather(X, mary).
X = john       ;
no
?- grandmother(mary, X).
no
?- son(X, helen).
X = nick       ;
no
?- daughter(alice, X).
X = mary       ;
no
?- male(X), hbsad(X).
X = john       ;
X = nick       ;
no
?-
```

Η τελευταία ερώτηση σημαίνει: «*Ποιος άνδρας έχει και γιο και κόρη;*».

Απάντηση άσκησης 3.4

Η σωστή λύση είναι να συμπληρώσετε τα κενά ως εξής:

```
greater(N1, N2) :- successor(N1, N2).
greater(N1, N3) :- successor(N1, N2), greater(N2, N3).
```

Ο προηγούμενος ορισμός σημαίνει ότι «*ένας μονοψήφιος μη αρνητικός ακέραιος αριθμός είναι μεγαλύτερος από κάποιον άλλο αν είναι επόμενός του ή αν είναι επόμενος ενός αριθμού μεγαλύτερου από αυτόν τον άλλο*».

Απάντηση άσκησης 3.5

Για να περιγράψουμε αν υπάρχει μονοπάτι που να συνδέει δύο κόμβους σε έναν κατευθυνόμενο γράφο χωρίς κύκλους, χρειαζόμαστε, καταρχάς, ένα σύνολο από γεγονότα που

να αναπαριστούν τις ακμές του γράφου. Για παράδειγμα, θα μπορούσαμε να ορίσουμε ένα κατηγόρημα `edge/2` ως εξής:

```
edge(a, b).
edge(a, d).
edge(b, c).
edge(b, d).
...........
```

που σημαίνει ότι «*στο γράφο μας υπάρχουν ακμές από τον κόμβο* `a` *στον κόμβο* `b`*, από τον* `a` *στον* `d`*, από τον* `b` *στον* `c`*, από τον* `b` *στον* `d` *κτλ.*». Έτσι, για τον ορισμό της ύπαρξης μονοπατιού που να συνδέει δύο κόμβους στο γράφο, θα μπορούσαμε να ορίσουμε το κατηγόρημα `connected/2` ως εξής:

```
connected(X, Y) :- edge(X, Y).
connected(X, Z) :- edge(X, Y), connected(Y, Z).
```

Οι κανόνες αυτοί υλοποιούν σε Prolog τα αξιώματα που δόθηκαν στην απάντηση της Άσκησης 1.6.

Απάντηση άσκησης 3.6
Το δέντρο ανάλυσης της άσκησης πρέπει να συμπληρωθεί ως εξής:

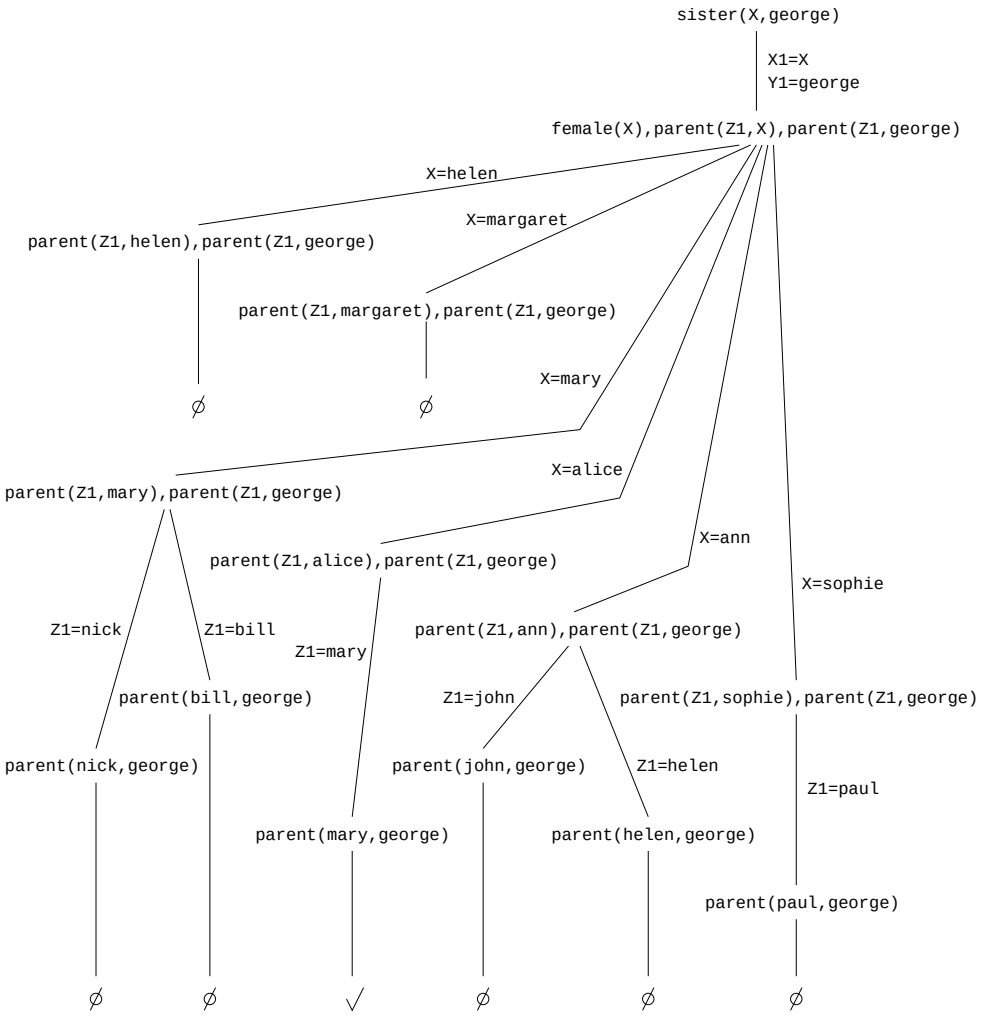
A: `Y1 = ann`
B: `grandparent(helen, X)`
C: `X = mary`
D: `X3 = helen`
E: `parent(helen, Y2), parent(Y2, mary)`
F: `parent(helen, Y4)`
G: `Y2 = nick`
H: `Y3 = ann`
I: `Y4 = ann`
J: `parent(ann, bill)`

Υπάρχουν τρεις απαντήσεις στην ερώτηση, οι `X = mary`, `X = bill` και `X = paul`.

Απάντηση άσκησης 3.7
Στο Σχήμα 3.7 φαίνεται το δέντρο ανάλυσης. Η ερώτηση έχει μία μόνο απάντηση, την `X = alice`. Στο δέντρο ανάλυσης της συγκεκριμένης ερώτησης θα μπορούσαμε να παρατηρήσουμε ένα ενδιαφέρον σημείο: ο αμέσως επόμενος κόμβος από τον αρχικό, δηλαδή ο «`female(X), parent(Z1,X), parent(Z1,george)`», έχει αρκετούς απογόνους, έξι συνολικά. Αυτό συμβαίνει γιατί ο πρώτος στόχος του, ο «`female(X)`», είναι αυτός που μέσω οπισθοδρόμησης επιλέγει μία-μία τις υπάρχουσες γυναίκες, για να ελεγχθεί στη συνέχεια, αν κάποιες από αυτές είναι αδελφές του *Γιώργου*.

Προβλήματα

Πρόβλημα 3.1
Όταν σε μια ερώτηση Prolog η απάντηση είναι `yes`, ποια από τις παρακάτω είναι η σωστή ερμηνεία της απάντησης;



Σχήμα 3.7: Δέντρο ανάλυσης για την ερώτηση «`?- sister(X, george).`».

1. *Με βάση τη γνώση που έχουμε, αυτό που ρωτάται είναι ίσως αληθές.*
2. *Με βάση τη γνώση που έχουμε, αυτό που ρωτάται είναι σίγουρα αληθές.*
3. *Ανεξάρτητα από τη γνώση που έχουμε, αυτό που ρωτάται είναι σίγουρα αληθές.*

Σε περίπτωση αρνητικής απάντησης `no`, ποια είναι η σωστή ερμηνεία της;

1. *Με βάση τη γνώση που έχουμε, αυτό που ρωτάται είναι ίσως ψευδές.*
2. *Με βάση τη γνώση που έχουμε, αυτό που ρωτάται είναι σίγουρα ψευδές.*
3. *Με βάση τη γνώση που έχουμε, δεν μπορούμε να γνωρίζουμε αν αυτό που ρωτάται είναι σίγουρα αληθές.*

Δικαιολογήστε τις απαντήσεις σας.

Πρόβλημα 3.2
Θεωρώντας δεδομένα τα γεγονότα `parent/2` του Παραδείγματος 3.1, ορίστε σε Prolog το κατηγόρημα `same_generation/2`, με τέτοιον τρόπο ώστε το `same_generation(X, Y)` να επιτυγχάνει όταν τα άτομα `X` και `Y` είναι της ίδιας γενιάς, δηλαδή βρίσκονται στο ίδιο επίπεδο

του γενεαλογικού δέντρου του Σχήματος 3.1. Για παράδειγμα, η `mary` και ο `paul` είναι ίδιας γενιάς.

Πρόβλημα 3.3
Με βάση τα δεδομένα του Παραδείγματος 3.1, ορίστε σε Prolog το κατηγόρημα `cousins/2`, με τέτοιον τρόπο ώστε το `cousins(X, Y)` να επιτυγχάνει όταν τα άτομα `X` και `Y` είναι πρώτα εξαδέλφια, όπως ο `jack` και ο `peter`.

Πρόβλημα 3.4
Με βάση τα δεδομένα του Παραδείγματος 3.1, ορίστε το κατηγόρημα `common_ancestor/3`, με τέτοιον τρόπο ώστε το `common_ancestor(X, Y, Z)` να επιτυγχάνει όταν τα άτομα `X` και `Y` έχουν κοινό πρόγονο το άτομο `Z`. Για παράδειγμα, ο `george` και η `ann` έχουν κοινό πρόγονο τον `john` (και την `helen`).

Πρόβλημα 3.5
Γνωρίζετε ότι η γνώση «*αν ισχύουν το `a` και το `b`, τότε θα ισχύει το `c`*» διατυπώνεται σε Prolog ως εξής:

```
c :- a, b.
```

Διατυπώστε τις παρακάτω δηλώσεις σε Prolog ή σχολιάστε τη δυνατότητα διατύπωσής τους σε Prolog:

1. «*αν ισχύει το `c`, τότε θα ισχύουν το `a` και το `b`*»
2. «*αν ισχύει το `a` ή το `b`, τότε θα ισχύει το `c`*»
3. «*αν ισχύει το `c`, τότε θα ισχύει το `a` ή το `b`*»

Πρόβλημα 3.6
Στην Ενότητα 3.2 είδαμε τον ορισμό του κατηγορήματος `ancestor/2`:

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

Θα μπορούσαμε να είχαμε δώσει εναλλακτικά τον ορισμό της σχέσης «πρόγονος» ως εξής;

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

Αν ναι, σε ποιες περιπτώσεις είναι περισσότερο συμφέρων ο ορισμός `ancestor/2` και πότε ο `alt_ancestor/2`; Αιτιολογήστε την απάντησή σας, κάνοντας, όπου χρειάζεται, τις απαιτούμενες διερευνήσεις.

Πρόβλημα 3.7
Ο ήρωας της μικρής μας ιστορίας διηγείται: «*Παντρεύτηκα μια χήρα, η οποία είχε μια κόρη. Ο πατέρας μου, ο οποίος μας επισκεπτόταν συχνά, γνωρίστηκε με την κόρη της γυναίκας μου και παντρεύτηκαν. Μήπως είμαι παππούς του εαυτού μου;*». Γράψτε πρόγραμμα Prolog και διατυπώστε, επίσης σε Prolog, την κατάλληλη ερώτηση, που θα απαντά στο θεμελιώδες

υπαρξιακό ερώτημα του φίλου μας. Στο πρόγραμμά σας να διατυπώσετε τη βασική γνώση του κόσμου που δόθηκε προηγουμένως, καθώς και την κοινή γενική γνώση που ισχύει στην καθημερινή ζωή. Σε αυτήν την κοινή γνώση, να συμπεριλάβετε ότι το παιδί της/του συζύγου κάποιου/ας είναι και παιδί του/της ίδιου/ας (παρότι μπορεί και να μην είναι βιολογικό του/της παιδί). Επίσης, σχεδιάστε και το δέντρο ανάλυσης που προκύπτει κατά τον υπολογισμό της απάντησης στην ερώτηση που πρέπει να υποβληθεί, με βάση το πρόγραμμα το οποίο γράψατε, ώστε να αποδειχθεί το ζητούμενο. Για να γίνει σαφής η σύνδεση του δέντρου ανάλυσης που θα σχεδιάσετε με τις προτάσεις του προγράμματός σας, αριθμήστε τις τελευταίες, μέσω σχολίων, στο πρόγραμμά σας.

Βιβλιογραφικές αναφορές

[1] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison Wesley (4th Edition), 2011.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer (5th Edition), 2003.

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

[4] J. D. Newmarch, *Logic Programming: Prolog and Stream Parallel Languages*, Prentice Hall, 1990.

[5] H. Aït-Kaci, *WAM: A Tutorial Reconstruction*, The MIT Press, 1991.

[6] D. H. D. Warren, *An Abstract Prolog Instruction Set*, (309) SRI International, 1983.

Κεφάλαιο 4
Δομές στην Prolog

Σύνοψη

Στο κεφάλαιο αυτό περιγράφονται αρχικά οι όροι, οι οποίοι είναι τα μέσα που υπάρχουν στην Prolog για την αναπαράσταση οντοτήτων, απλών ή σύνθετων. Επίσης, εξετάζεται μια ειδική μορφή όρων, οι λίστες, οι οποίες χρησιμοποιούνται για την αναπαράσταση συλλογών από ομοειδείς οντότητες.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει μελετήσει και κατανοήσει το Κεφάλαιο 3.

4.1 Όροι

Στο Κεφάλαιο 3 είδαμε πώς μπορούμε να διατυπώνουμε απλά προγράμματα Prolog, για να κωδικοποιούμε στοιχειώδεις κόσμους [1, 2, 3]. Ειδικότερα, στο Παράδειγμα 3.1 δώσαμε την αναπαράσταση σε Prolog, ως ένα σύνολο από γεγονότα, ενός γενεαλογικού δέντρου. Τα πρόσωπα που συμμετείχαν στο δέντρο αναπαραστάθηκαν μέσω σταθερών (π.χ. `john`, `nick` κτλ.), ενώ στις ερωτήσεις που υποβάλαμε χρησιμοποιήσαμε μεταβλητές, για να αναφερθούμε σε τυχαία πρόσωπα. Οι σταθερές και οι μεταβλητές είναι δύο περιπτώσεις όρων στην Prolog. Σχετικά με τη σημασία ενός όρου, ισχύει το εξής:

Ένας όρος στην Prolog αναπαριστά μια οντότητα του κωδικοποιούμενου κόσμου.

Πολλές φορές όμως, δεν μας αρκεί να αναπαραστήσουμε μια οντότητα του κόσμου μας με ένα απλό σύμβολο. Για παράδειγμα, στον κόσμο του γενεαλογικού μας δέντρου, ίσως να είναι απαραίτητο να αναπαραστήσουμε τα πρόσωπα όχι μόνο με το μικρό τους όνομα, αλλά και με το επώνυμό τους. Τη λύση σε αυτό το πρόβλημα μας τη δίνουν οι **δομές** ή **σύνθετοι όροι** στην Prolog. Για παράδειγμα, τον *Γιάννη* του κόσμου μας, του οποίου το πλήρες όνομα είναι *Γιάννης Ιωάννου*, θα μπορούσαμε να τον αναπαραστήσουμε με το «`full_name(john, jones)`». Αυτός είναι ένας σύνθετος όρος, που πάλι αναπαριστά μια οντότητα του κόσμου, λίγο πιο «πολύπλοκη» από αυτές που αναπαριστούμε με σταθερές. Το `full_name` είναι ένα **συναρτησιακό σύμβολο**, που χρησιμοποιείται για τη δόμηση του σύνθετου όρου. Κάθε συναρτησιακό σύμβολο έχει και αυτό, όπως τα κατηγορήματα, ένα βαθμό. Ο βαθμός του `full_name` είναι 2, επειδή ο όρος που δομείται από αυτό έχει δύο στοιχειώδη συστατικά, που είναι τα **ορίσματά** του, δηλαδή το `john` και το `jones`. Η γενική μορφή ενός σύνθετου όρου είναι η

$$functor(term_1, term_2, \ldots, term_n)$$

στην οποία το $functor$ είναι το συναρτησιακό σύμβολο δόμησης του όρου, με βαθμό n, και τα $term_1$, $term_2$, ..., $term_n$ είναι τα ορίσματά του, που είναι όροι. Οι όροι αυτοί μπορεί να είναι σύνθετοι, δηλαδή ο ορισμός του σύνθετου όρου κρύβει αναδρομή. Συνοψίζοντας:

Ένας όρος στην Prolog μπορεί να είναι σταθερά, μεταβλητή ή δομή (σύνθετος όρος).

Παράδειγμα 4.1
Αν αναδιατυπώσουμε το πρόγραμμα Prolog του Παραδείγματος 3.1, έτσι ώστε τα πρόσωπα να αναπαριστάνονται και με το επώνυμό τους, τότε θα έχουμε (χωρίς να λάβουμε υπόψη μας όσα ισχύουν στο οικογενειακό δίκαιο στην Ελλάδα):

```
parent(full_name(john, jones), full_name(nick, jones)).
parent(full_name(john, jones), full_name(ann, smith)).
parent(full_name(helen, jones), full_name(nick, jones)).
parent(full_name(helen, jones), full_name(ann, smith)).
parent(full_name(nick, jones), full_name(mary, clark)).
parent(full_name(nick, jones), full_name(bill, jones)).
parent(full_name(margaret, jones), full_name(mary, clark)).
parent(full_name(margaret, jones), full_name(bill, jones)).
parent(full_name(mary, clark), full_name(george, clark)).
parent(full_name(mary, clark), full_name(jack, clark)).
parent(full_name(mary, clark), full_name(alice, clark)).
parent(full_name(bill, jones), full_name(peter, jones)).
parent(full_name(chris, smith), full_name(paul, smith)).
parent(full_name(ann, smith), full_name(paul, smith)).
parent(full_name(paul, smith), full_name(sophie, smith)).
```

Φυσικά, για λόγους συμβατότητας, θα πρέπει να αναδιατυπώσουμε και τους ορισμούς των κατηγορημάτων `male/1` και `female/1`, έτσι ώστε να αναφέρονται και αυτά στα συγκεκριμένα πρόσωπα, αλλά με τη νέα τους αναπαράσταση. Δηλαδή:

```
male(full_name(john, jones)).
male(full_name(nick, jones)).
..................
female(full_name(mary, clark)).
..................
```
□

Παρά την τροποποίηση που κάναμε στα γεγονότα `parent/2`, `male/1` και `female/1`, για όλα τα υπόλοιπα κατηγορήματα που ορίσαμε στο Κεφάλαιο 3, όπως `grandparent/2`, `father/2`, `sister/2`, `ancestor/2` κτλ., δεν υπάρχει λόγος να κάνουμε οποιαδήποτε αλλαγή. Τις ερωτήσεις που μπορούσαμε να απαντήσουμε με την προηγούμενη αναπαράσταση των προσώπων μπορούμε να τις απαντήσουμε και με τη νέα αναπαράσταση. Για παράδειγμα:

```
?- father(full_name(nick, jones), X).
X = full_name(mary, clark)      ;
X = full_name(bill, jones)
?- ancestor(X, full_name(paul, smith)).
X = full_name(chris, smith)     ;
X = full_name(ann, smith)       ;
X = full_name(john, jones)      ;
X = full_name(helen, jones)
?-
```

Μπορούμε όμως να υποβάλουμε και ακόμα πιο ενδιαφέρουσες ερωτήσεις, όπως «*υπάρχει κάποιος* `smith` *που ο παππούς του να είναι* `jones` *και ποιο είναι το μικρό του όνομα;*» ή «*υπάρχει κάποια που να έχει το ίδιο μικρό όνομα με τη γιαγιά της;*»:

```
?- grandfather(full_name(_, jones), full_name(X, smith)).
X = paul
?- female(full_name(X, Y)),
   grandmother(full_name(X, _), full_name(X, Y)).
no
?-
```

Το σύμβολο _ στις προηγούμενες ερωτήσεις είναι η **ανώνυμη μεταβλητή**, την οποία χρησιμοποιούμε για να δείξουμε ότι το συγκεκριμένο όρισμα είναι άγνωστο, δηλαδή μεταβλητή, αλλά δεν μας ενδιαφέρει ούτε η συσχέτιση με κάποιο άλλο όρισμα της πρότασης, ούτε, αν πρόκειται για ερώτηση, όπως εδώ, η τιμή της μεταβλητής αυτής για την οποία η ερώτηση ικανοποιείται.

Για να εμπεδώσετε ό,τι παρουσιάστηκε στην ενότητα αυτή, δοκιμάστε να αντιμετωπίσετε τις ασκήσεις που ακολουθούν.

Άσκηση 4.1

Συμπληρώστε τις επόμενες ερωτήσεις, όπως είναι διατυπωμένες σε Prolog:

1. «*Υπάρχει κάποιος πρόγονος ενός* `clark`*;*»
   `?- ancestor(X,` ☐ `).`
2. «*Υπάρχει κάποιος του οποίου και οι δύο παππούδες να έχουν το ίδιο μικρό όνομα;*»
   `?- grandfather(full_name(X, Y),` ☐ `),`
   ☐ `(` ☐ `, W), diff(Y, Z).`
3. «*Ποιο είναι το μικρό όνομα μιας γυναίκας που έχει κόρη με διαφορετικό επώνυμο από το δικό της;*»
   `?-` ☐ `(full_name(X, Y)),`
   `daughter(` ☐ `,` ☐ `),` ☐ `(` ☐ `, Z).`

Υπενθυμίζεται ότι το κατηγόρημα `diff/2`, στο οποίο αναφερθήκαμε στην Ενότητα 3.1, ικανοποιείται όταν τα δύο ορίσματά του είναι διαφορετικά.

Άσκηση 4.2

Διατυπώστε, με την εισαγωγή ενός κατηγορήματος και κατάλληλων συναρτησιακών συμβόλων και σταθερών, την εξής γνώση: «*Ο* `john jones` *είναι άνδρας, είναι παντρεμένος με την* `helen jones`*, γεννήθηκε στις 12 Απριλίου 1931 και είναι συνταξιούχος, με μηνιαία σύνταξη 1.200 ευρώ. Η* `alice clark` *είναι ανύπαντρη γυναίκα, γεννήθηκε στις 25 Αυγούστου 1990 και δεν εργάζεται. Η* `ann smith` *είναι γυναίκα, είναι παντρεμένη με τον* `chris smith`*, γεννήθηκε στις 17 Φεβρουαρίου 1982 και εργάζεται στη* `lufthansa`*, με μηνιαίο μισθό 2.500 ευρώ.*».

Άσκηση 4.3

Έστω ότι οι αριθμοί 0, 1, 2, 3, ... παριστάνονται από τους όρους Prolog `0`, `s(0)`, `s(s(0))`, `s(s(s(0)))`, ... αντίστοιχα. Ορίστε σε Prolog το κατηγόρημα `mynumber/1`, έτσι ώστε το `mynumber(X)` να είναι αληθές όταν το `X` είναι μη αρνητικός ακέραιος αριθμός, σύμφωνα

με την αναπαράσταση `s(s(...))`. Υιοθετώντας αυτήν την αναπαράσταση, ορίστε επίσης σε Prolog τα κατηγορήματα `myplus/3`, `myminus/3`, `mytimes/3`, `mydiv/3`, `myexp/3` και `myfact/2`, με τις προδιαγραφές:

- Το `myplus(X,Y,Z)` αληθεύει όταν $\texttt{X} + \texttt{Y} = \texttt{Z}$.
- Το `myminus(X,Y,Z)` αληθεύει όταν $\texttt{X} - \texttt{Y} = \texttt{Z}$ και $\texttt{X} \geq \texttt{Y}$.
- Το `mytimes(X,Y,Z)` αληθεύει όταν $\texttt{X} * \texttt{Y} = \texttt{Z}$.
- Το `mydiv(X,Y,Z)` αληθεύει όταν $\texttt{X} \,/\, \texttt{Y} = \texttt{Z}$ και $\texttt{X}\ mod\ \texttt{Y} = 0$.
- Το `myexp(X,N,Y)` αληθεύει όταν $\texttt{X}^{\texttt{N}} = \texttt{Y}$.
- Το `myfact(N,X)` αληθεύει όταν $\texttt{N}! = \texttt{X}$.

4.2 Λίστες

Επανερχόμενοι στο Παράδειγμα 3.1, θα μπορούσαμε να σκεφτούμε ότι ένας εναλλακτικός τρόπος διατύπωσης του κόσμου που περιγράφεται μπορεί να προσανατολίζεται στην οικογένεια, αντί στις σχέσεις γονέων-παιδιών. Δηλαδή, θα μπορούσαμε να είχαμε ένα γεγονός για κάθε οικογένεια, με ορίσματα τον πατέρα, τη μητέρα και τα παιδιά. Αν το κατηγόρημα αυτό ήταν το `family`, θα μπορούσαμε εύλογα να γράψουμε για την οικογένεια του `john` και της `helen`:[1]

```
family(john, helen, nick, ann).
```

Ο προηγούμενος ορισμός όμως έχει ένα πολύ σοβαρό μειονέκτημα. Βασίζεται σε ένα κατηγόρημα `family`, το οποίο έχει βαθμό 4, επειδή η συγκεκριμένη οικογένεια έχει 2 παιδιά. Γενικά, για κάθε οικογένεια με N παιδιά, το `family` θα είχε βαθμό $N + 2$. Αυτό όμως είναι ανεπιθύμητο, γιατί, για να μπορούμε να αναφερόμαστε σε μια οικογένεια, θα πρέπει να γνωρίζουμε εκ των προτέρων το πλήθος των παιδιών της, έτσι ώστε να χρησιμοποιούμε το κατηγόρημα με τον σωστό βαθμό.

Ίσως θα μπορούσαμε να ισχυριστούμε ότι το προηγούμενο πρόβλημα θα αντιμετωπιστεί αν το κατηγόρημα `family` έχει πάντα βαθμό 3, με τα δύο πρώτα ορίσματα να παριστάνουν τον πατέρα και τη μητέρα της οικογένειας, και το τρίτο όρισμα να είναι ένας σύνθετος όρος, έστω με συναρτησιακό σύμβολο `children`, που να έχει ορίσματα τα παιδιά της οικογένειας. Δηλαδή:

```
family(john, helen, children(nick, ann)).
```

Πάλι όμως αυτή η προσέγγιση δεν μας λύνει το πρόβλημα, γιατί μεταθέτει την «αστάθεια» του βαθμού από το κατηγόρημα στο συναρτησιακό σύμβολο. Το `children` θα έχει βαθμό ίσο με το πλήθος των παιδιών της οικογένειας, δηλαδή διαφορετικό από οικογένεια σε οικογένεια. Και αυτή η ανομοιομορφία θα μας δημιουργούσε προβλήματα στο μέλλον. Για παράδειγμα, δεν θα μπορούσαμε να βρούμε έναν εύκολο τρόπο να μάθουμε τους γονείς του `nick`.

Από τα προηγούμενα συμπεραίνουμε ότι τα κατηγορήματα και τα συναρτησιακά σύμβολα είναι άμεσα συνυφασμένα με το βαθμό τους. Ουσιαστικά, δεν απαγορεύεται να έχουμε

[1] Στην ενότητα αυτή, δεν ακολουθούμε την αναπαράσταση των προσώπων με το πλήρες τους όνομα, όχι γιατί δεν πρέπει να γίνει έτσι, αλλά γιατί το σημείο στο οποίο θέλουμε να εστιαστούμε είναι άλλο. Αν εμπλέκαμε και τους σύνθετους όρους με τα πλήρη ονόματα, δεν θα είχαμε ιδιαίτερο πρόσθετο όφελος από εκπαιδευτική άποψη. Μάλλον, θα περιπλέκαμε άσκοπα τα πράγματα.

δύο κατηγορήματα (ή δύο συναρτησιακά σύμβολα) με το ίδιο όνομα, αλλά διαφορετικό βαθμό, μόνο που δεν θα μπορούν να συσχετιστούν και, συνεπώς, να χρησιμοποιηθούν για την αναπαράσταση ομοειδών σχέσεων (ή σύνθετων αντικειμένων). Ουσιαστικά, θα πρόκειται για διαφορετικά κατηγορήματα (ή συναρτησιακά σύμβολα), παρότι θα έχουν το ίδιο όνομα.

Τελικά, ακολουθώντας πιστά όσα εξετάσαμε μέχρι τώρα, θα μπορούσαμε, για να αντεπεξέλθουμε στο πρόβλημα της αναπαράστασης μιας οικογένειας, να υιοθετήσουμε μια «φαεινή ιδέα», η οποία δεν είναι τίποτε άλλο από την κωδικοποίηση των παιδιών της οικογένειας, με τη βοήθεια ενός συναρτησιακού συμβόλου `children/2`, που έχει πρώτο όρισμα ένα παιδί της οικογένειας και δεύτερο όρισμα τα υπόλοιπα παιδιά της. Αυτά τα υπόλοιπα παιδιά μπορούν να αναπαρασταθούν πάλι με μια δομή `children/2` κ.ο.κ., μέχρι να μην υπάρχουν υπόλοιπα παιδιά. Οπότε, αυτό μπορούμε να το περιγράψουμε με μια σταθερά, όπως τη `no_more_children`. Δηλαδή:

```
family(john, helen, children(nick, children(ann, no_more_children))).
```

Έτσι, το συναρτησιακό σύμβολο `children` έχει πάντα βαθμό 2, ανεξάρτητα από το πλήθος των παιδιών μιας οικογένειας. Φυσικά, αν μια οικογένεια δεν έχει παιδιά, το τρίτο όρισμα του γεγονότος `family/3` θα είναι το `no_more_children`.

Η δομή `children`, που προτείναμε για την κωδικοποίηση των παιδιών μιας οικογένειας, άνοιξε ένα πολύ μεγάλο θέμα σχετικά με την αναπαράσταση γνώσης στην Prolog, αλλά και γενικότερα στην Επιστήμη των Υπολογιστών. Η δομή αυτή παριστάνει μια συλλογή από ομοειδείς οντότητες, χωρίς να υπάρχει κάποια ουσιαστική δέσμευση σχετικά με το πλήθος αυτών των οντοτήτων. Η δομή αυτή είναι μια **λίστα**. Επειδή η χρησιμότητά της στην Prolog είναι τεράστια, προτείνονται, για λόγους ομοιόμορφης αντιμετώπισης των λιστών, συγκεκριμένο συναρτησιακό σύμβολο δόμησης λιστών (αντίστοιχο του `children/2`), το `./2`, και συγκεκριμένη σταθερά, η οποία δείχνει το τέλος μιας λίστας (αντίστοιχη του `no_more_children`), η `[]`, που ονομάζεται και **κενή λίστα**. Με αυτόν τον προτεινόμενο από την Prolog συμβολισμό, το γεγονός που προαναφέρθηκε θα μπορούσε να διατυπωθεί ως εξής:

```
family(john, helen, .(nick, .(ann, []))).
```

Για λίστες που είναι κωδικοποιημένες με τα σύμβολα `./2` και `[]`, η Prolog παρέχει και έναν ισοδύναμο, εναλλακτικό, αλλά περισσότερο φιλικό τρόπο διατύπωσης, τον εξής:

```
family(john, helen, [nick, ann]).
```

Δηλαδή, μια ακολουθία από όρους, κλεισμένη μέσα σε «`[`» και «`]`», οι οποίοι χωρίζονται με «`,`», είναι μία λίστα στην Prolog, σαν να την είχαμε δομήσει με τη βοήθεια του συναρτησιακού συμβόλου `./2` και της σταθεράς `[]`. Ουσιαστικά, το `./2` χρησιμοποιείται για να αναπαραστήσει μια λίστα μέσω του πρώτου της στοιχείου, που ονομάζεται κεφαλή της, και της υπόλοιπης λίστας, που ονομάζεται ουρά της. Δηλαδή, η λίστα με κεφαλή το `X` και ουρά το `L` είναι η `.(X,L)`. Η Prolog μάς προσφέρει τη δυνατότητα διατύπωσης αυτής της λίστας και με έναν φιλικότερο τρόπο, που είναι ο `[X|L]`.

Παράδειγμα 4.2

Λαμβάνοντας υπόψη τα προηγούμενα, μπορούμε να αναδιατυπώσουμε τη γνώση του Παραδείγματος 3.1, προσθέτοντας ότι η `mary` είναι παντρεμένη με τον `ted`, ο `bill` με τη `susan`, ο `paul` με τη `ruth` και η `sophie` με τον `eric`, ως εξής:

```
family(john, helen, [nick, ann]).
family(nick, margaret, [mary, bill]).
family(ted, mary, [george, jack, alice]).
family(bill, susan, [peter]).
family(chris, ann, [paul]).
family(paul, ruth, [sophie]).
family(eric, sophie, []).
```

Έχοντας τώρα εισαγάγει τη δομή της λίστας, η οποία μας επιτρέπει να αναπαραστήσουμε συλλογές από ομοειδείς οντότητες, είναι απολύτως βέβαιο ότι θα χρειαστεί να ορίσουμε και ένα πλήθος από κατηγορήματα, για τη διαχείριση των λιστών. Ένα τέτοιο κλασικό κατηγόρημα είναι το `member/2`, με το οποίο ορίζουμε πότε ένα στοιχείο είναι μέλος μιας λίστας. Ένα άλλο πολύ χρήσιμο κατηγόρημα είναι το `append/3`, με το οποίο ορίζουμε πότε μια λίστα είναι η συνένωση δύο άλλων λιστών. Οι ορισμοί των κατηγορημάτων αυτών δίνονται στα Παραδείγματα 4.3 και 4.4.

Παράδειγμα 4.3
Για να δώσουμε τον ορισμό σε Prolog του κατηγορήματος `member/2`, αρκεί να εκφράσουμε τα αξιώματα με τα οποία ορίζεται η σχέση που μας ενδιαφέρει, όπως μάθαμε στο Κεφάλαιο 1, και στη συνέχεια να τα διατυπώσουμε σε Prolog.

Αξιώματα για μέλος λίστας
Αξίωμα 1: *Ένα στοιχείο είναι μέλος μιας λίστας αν είναι η κεφαλή της.*
Αξίωμα 2: *Ένα στοιχείο είναι μέλος μιας λίστας αν είναι μέλος της ουράς της.*

Τα προηγούμενα αξιώματα υλοποιούνται σε Prolog ως εξής:

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

Στο πρόγραμμα αυτό μπορούμε να υποβάλουμε σχετικές ερωτήσεις και να πάρουμε τις κατάλληλες απαντήσεις:

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

Όμως, περισσότερο ενδιαφέρουσα ερώτηση είναι η εξής:

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

Δηλαδή, βλέπουμε ότι, παρότι το `member/2` ορίστηκε έτσι ώστε να ελέγχει αν ένα δεδομένο στοιχείο είναι μέλος μιας λίστας, μπορεί επίσης να «γεννά» τα στοιχεία της μέσω οπισθοδρόμησης.

Παράδειγμα 4.4
Για να διατυπώσουμε σε Prolog το κατηγόρημα `append/3`, που ορίζει τη συνένωση δύο λιστών, ας γράψουμε πρώτα τα αξιώματα τα οποία διέπουν τη λειτουργία που μας ενδιαφέρει:

Αξιώματα για τη συνένωση λιστών

Αξίωμα 1: *Η συνένωση της κενής λίστας με οποιαδήποτε λίστα είναι η λίστα αυτή.*

Αξίωμα 2: *Η συνένωση μιας λίστας που έχει κεφαλή και ουρά με μια δεύτερη λίστα είναι μια λίστα που έχει κεφαλή την κεφαλή της πρώτης λίστας και ουρά τη συνένωση της ουράς της πρώτης λίστας με τη δεύτερη λίστα.*

Τα αξιώματα αυτά διατυπώνονται ως εξής σε Prolog:

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

Έτσι, μπορούμε να δίνουμε ερωτήσεις και να παίρνουμε απαντήσεις, όπως:

```
?- append([a,b], [c,d,e], [a,b,c,d,e]).
yes
?- append([a], [c,d], [a,b,c,d]).
no.
?- append([a,b,c], [d], L).
L = [a,b,c,d]
?-
```

Αλλά, μπορούμε να θέσουμε και ακόμα πιο ενδιαφέρουσες ερωτήσεις, όπως:

```
?- append(L, [c,d,e], [a,b,c,d,e]).
L = [a,b]     ;
no
?- append([a,b,c], L, [a,b,c,d,e]).
L = [d,e]
?- append(L1, L2, [a,b,c]).
L1 = []
L2 = [a,b,c]  ;
L1 = [a]
L2 = [b,c]    ;
L1 = [a,b]
L2 = [c]      ;
L1 = [a,b,c]
L2 = []       ;
no
?-
```

Δηλαδή, το κατηγόρημα `append/3` μπορεί να χρησιμοποιηθεί όχι μόνο για τη συνένωση λιστών, αλλά και για τη διαφορά λιστών ή, ακόμα, και για τη διάσπαση λιστών σε δύο άλλες, με όλους τους δυνατούς τρόπους, μέσω οπισθοδρόμησης. □

Έχοντας τώρα στη διάθεσή μας και διάφορα χρήσιμα κατηγορήματα διαχείρισης λιστών, όπως τα `member/2` και `append/3` των Παραδειγμάτων 4.3 και 4.4, αντίστοιχα, μπορούμε να υποβάλουμε στη νέα έκδοση του γενεαλογικού κόσμου μας, αυτήν του Παραδείγματος 4.2, διάφορες ερωτήσεις, όπως:

- «*Ποιος είναι ο πατέρας του `jack`;*»
  ```
  ?- family(X, _, C), member(jack, C).
  ```
- «*Ποια είναι συνολικά τα παιδιά της `margaret` και του `chris`;*»
  ```
  family(_, margaret, C1), family(chris, _, C2), append(C1, C2, C).
  ```

- «*Ποια είναι μητέρα μιας γυναίκας που δεν έχει παιδιά;*»
  `family(_, X, []), family(_, Y, C), member(X, C).`

Άσκηση 4.4

Μερικές φορές, κάποιοι ισχυρίζονται ότι ο όρος `[X,L]` είναι ακριβώς ο ίδιος με τον όρο `[X|L]`. Είναι σωστό αυτό ή λάθος; Αν ρωτήσουμε τους ίδιους τι είναι το `[[]]`, θα απαντήσουν ότι είναι η κενή λίστα. Τι έχετε να πείτε γι' αυτήν την απάντηση; Είναι σωστή ή λάθος;

Άσκηση 4.5

Ένα κλασικό κατηγόρημα διαχείρισης λιστών είναι το `delete/3`. Οι προδιαγραφές του κατηγορήματος αυτού ορίζουν ότι ο στόχος `delete(X, L, R)` επιτυγχάνει όταν το `X` ανήκει στη λίστα `L` και το `R` είναι η λίστα που απομένει όταν από την `L` διαγραφεί το `X`. Συμπληρώστε τα κενά στον ορισμό του `delete/3`, που ακολουθεί:

```
delete(X, [      ], L).
delete(X, [Y|L1], [    |L2]) :-                    .
```

Άσκηση 4.6

Πολλές φορές είναι χρήσιμο να γράψουμε ένα κατηγόρημα `sublist/2` το οποίο να επιτυγχάνει όταν το πρώτο όρισμά του είναι μια λίστα που συνιστά τμήμα της λίστας η οποία είναι δεύτερο όρισμά του. Δηλαδή, θα επιθυμούσαμε την εξής συμπεριφορά:

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

Στον ημιτελή ορισμό του `sublist/2`:

```
sublist(L1, L2) :- append(    A    ), append(    B    ).
```

ποια είναι η σωστή επιλογή ή οι σωστές επιλογές, για να συμπληρώσουμε στα **A** και **B**, ώστε το `sublist/2` να έχει την επιθυμητή λειτουργικότητα;

1. **A**: `L1, L3, _` **B**: `_, L2, L3`
2. **A**: `_, L3, L2` **B**: `L1, _, L3`
3. **A**: `L2, _, L1` **B**: `L3, L1, _`
4. **A**: `_, L1, L2` **B**: `L1, _, L2`
5. **A**: `L3, _, L2` **B**: `_, L1, L3`

Άσκηση 4.7
Στην Άσκηση 1.3, σας ζητήθηκε να συμπληρώσετε τα αξιώματα για την αντιστροφή μιας λίστας. Ορίστε σε Prolog ένα κατηγόρημα `reverse/2` το οποίο να υλοποιεί τα αξιώματα αυτά. Φυσικά, αν δείτε πάλι την απάντηση της άσκησης αυτής, θα διαπιστώσετε ότι απαιτείται και ο ορισμός ενός κατηγορήματος, ας το ονομάσουμε `add_at_end/3`, για την τοποθέτηση ενός στοιχείου στο τέλος μιας λίστας.

Άσκηση 4.8
Ορίστε τα κατηγορήματα `is_list/1`, `pref/2`, `suff/2`, `consecutive/3` και `equals/2`, με τις προδιαγραφές:

- Το `is_list(L)` αληθεύει όταν η `L` είναι αποδεκτή λίστα Prolog.
- Το `pref(L1, L2)` αληθεύει όταν η `L1` είναι υπολίστα της λίστας `L2` στην αρχή της.
- Το `suff(L1, L2)` αληθεύει όταν η `L1` είναι υπολίστα της λίστας `L2` στο τέλος της.
- Το `consecutive(X, Y, L)` αληθεύει όταν τα `X` και `Y` είναι διαδοχικά στοιχεία στη λίστα `L`.
- Το `equals(L1, L2)` αληθεύει όταν οι λίστες `L1` και `L2` περιέχουν τα ίδια ακριβώς στοιχεία, όχι απαραίτητα με την ίδια σειρά.

Απαντήσεις ασκήσεων

Απάντηση άσκησης 4.1
Οι διατυπώσεις σε Prolog των ερωτήσεων που δόθηκαν είναι οι εξής:

1. «*Υπάρχει κάποιος πρόγονος ενός* `clark`;»
   ```
   ?- ancestor(X, full_name(Y, clark)).
   ```
   Φυσικά, αντί για `Y`, θα μπορούσαμε να είχαμε την ανώνυμη μεταβλητή _.
2. «*Υπάρχει κάποιος του οποίου οι δύο παππούδες να έχουν ίδιο μικρό όνομα;*»
   ```
   ?- grandfather(full_name(X, Y), W),
      grandfather(full_name(X, Z), W), diff(Y, Z).
   ```
   Είναι σημαντικό να εξασφαλίσουμε ότι οι δύο παππούδες έχουν διαφορετικά επώνυμα, ώστε να μην πρόκειται για το ίδιο πρόσωπο και έχουμε πάλι τα προβλήματα που αντιμετωπίσαμε στην Ενότητα 3.1, με τον ορισμό του `sister/2`.
3. «*Ποιο είναι το μικρό όνομα μιας γυναίκας που έχει κόρη με διαφορετικό επώνυμο από το δικό της;*»
   ```
   ?- female(full_name(X, Y)),
      daughter(full_name(_, Z), full_name(X, Y)), diff(Y, Z).
   ```
   Στη θέση του μικρού ονόματος της κόρης θα μπορούσαμε να είχαμε, αντί της ανώνυμης μεταβλητής _, μια άλλη μεταβλητή, που δεν χρησιμοποιείται στην ίδια ερώτηση, όπως `W`.

Απάντηση άσκησης 4.2
Μπορούμε να αναπαραστήσουμε σε Prolog τον κόσμο που διατυπώθηκε σε φυσική γλώσσα μέσω ενός συνόλου από γεγονότα, με ένα γεγονός για κάθε πρόσωπο, έστω με κατηγόρημα `person/5`. Το πρώτο όρισμα του `person/5` περιγράφει το φύλο του προσώπου (`man`

ή `woman`) και το δεύτερο όρισμα είναι το πλήρες όνομά του (σύνθετος όρος, με συναρτησιακό σύμβολο `full_name/2`). Το τρίτο όρισμα δηλώνει την οικογενειακή του κατάσταση (`single` αν είναι ανύπαντρος/η ή σύνθετος όρος, με συναρτησιακό σύμβολο `married_to/1` και όρισμα το πλήρες όνομα της/του συζύγου) και το τέταρτο όρισμα είναι η ημερομηνία γέννησής του (σύνθετος όρος, με συναρτησιακό σύμβολο `date_born/3` και ορίσματα την ημερομηνία, το μήνα και το έτος). Για το πέμπτο όρισμα, που περιγράφει την επαγγελματική κατάσταση του προσώπου, υπάρχουν τρεις περιπτώσεις, είτε δεν εργάζεται (`unemployed`) είτε είναι συνταξιούχος (σύνθετος όρος, με συναρτησιακό σύμβολο `retired/1` και όρισμα τη μηνιαία σύνταξή του) είτε είναι εργαζόμενος (σύνθετος όρος, με συναρτησιακό σύμβολο `works/2` και ορίσματα την εταιρεία στην οποία εργάζεται και το μισθό του). Έχουμε, δηλαδή:

```
person(man, full_name(john, jones),
       married_to(full_name(helen, jones)), date_born(12, apr, 1931),
       retired(1200)).
person(woman, full_name(alice, clark),
       single, date_born(25, aug, 1990),
       unemployed).
person(woman, full_name(ann, smith),
       married_to(full_name(chris, smith)), date_born(17, feb, 1982),
       works(lufthansa, 2500)).
```

Φυσικά, ο προηγούμενος τρόπος αναπαράστασης του κόσμου δεν είναι μοναδικός. Θα μπορούσαμε να είχαμε υιοθετήσει κάποια άλλη μορφή διατύπωσης, όμως δεν θα ήταν ουσιωδώς διαφορετική.

Απάντηση άσκησης 4.3
Ένας τρόπος ορισμού των ζητούμενων κατηγορημάτων είναι ο εξής:

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

myplus(0, X, X) :- mynumber(X).
myplus(s(X), Y, s(Z)) :- myplus(X, Y, Z).

myminus(X, Y, Z) :- myplus(Y, Z, X).

mytimes(0, X, 0) :- mynumber(X).
mytimes(s(X), Y, Z) :- mytimes(X, Y, XY), myplus(XY, Y, Z).

mydiv(X, X, s(0)) :- mynumber(X).
mydiv(X, Y, s(Z)) :- myminus(X, Y, XY), mydiv(XY, Y, Z).

myexp(0, s(X), 0) :- mynumber(X).
myexp(s(X), 0, s(0)) :- mynumber(X).
myexp(X, s(N), Y) :- myexp(X, N, Z), mytimes(Z, X, Y).

myfact(0, s(0)).
myfact(s(X), Y) :- myfact(X, FX), mytimes(s(X), FX, Y).
```

Δείτε και παραδείγματα ερωτήσεων που μπορούμε να υποβάλουμε στο προηγούμενο πρόγραμμα, μαζί με τις απαντήσεις τους:

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

```
?- mynumber(5).
no
?- mynumber(0).
yes
?- myplus(s(s(0)), s(s(s(0))), X).
X = s(s(s(s(s(0)))))
?- myplus(0, 5, X).
no
?- myminus(s(s(s(s(0)))), s(0), X).
X = s(s(s(0)))
?- myminus(s(s(0)), s(s(s(0))), X).
no
?- mytimes(s(s(s(s(0)))), s(s(0)), X).
X = s(s(s(s(s(s(s(s(0))))))))
?- mytimes(3, 0, X).
no
?- mydiv(s(s(s(s(s(s(0)))))), s(s(0)), X).
X = s(s(s(0)))                          ;
no
?- mydiv(s(s(s(0))), s(s(0)), X).
no
?- myexp(s(s(s(0))), s(s(0)), X).
X = s(s(s(s(s(s(s(s(s(0))))))))    ;
no
?- myfact(s(s(s(0))), X).
X = s(s(s(s(s(s(0))))))
?-
```

Απάντηση άσκησης 4.4
Ο όρος `[X,L]` είναι εντελώς διαφορετικός από τον όρο `[X|L]`. Ο πρώτος παριστάνει μια λίστα με δύο στοιχεία, τα `X` και `L`, δηλαδή είναι ουσιαστικά ο όρος `.(X, .(L, []))`. Ο δεύτερος είναι μια λίστα με κεφαλή το `X` και ουρά το `L`, δηλαδή είναι ο όρος `.(X, L)`. Συνεπώς, δεν είναι σωστό ότι αυτοί οι δύο όροι παριστάνουν την ίδια οντότητα. Όσον αφορά το δεύτερο ερώτημα, ο όρος `[[]]` δεν είναι η κενή λίστα, αλλά μια λίστα με ένα στοιχείο, την κενή λίστα. Είναι, δηλαδή, ίδιος με τον όρο `.([], [])`. Αυτό είναι, βέβαια, εντελώς διαφορετικό από την κενή λίστα `[]`. Άρα, και αυτή η απάντηση είναι λάθος.

Απάντηση άσκησης 4.5
Ο συμπληρωμένος ορισμός για το κατηγόρημα `delete/3` είναι ο εξής:

```
delete(X, [X|L], L).
delete(X, [Y|L1], [Y|L2]) :- delete(X, L1, L2).
```

Η πρώτη πρόταση δηλώνει ότι μπορούμε να διαγράψουμε την κεφαλή από μια λίστα και να μείνει η ουρά της. Η δεύτερη πρόταση δηλώνει ότι μπορούμε να διαγράψουμε ένα στοιχείο από την ουρά μιας λίστας, οπότε απομένει μια λίστα που έχει κεφαλή την κεφαλή της αρχικής και ουρά τη λίστα που προκύπτει μετά τη διαγραφή του συγκεκριμένου στοιχείου από την ουρά της αρχικής. Παρατηρήστε ότι το `delete/3` δεν είναι τίποτε άλλο από μια γενίκευση του `member/2`, που είδαμε στο Παράδειγμα 4.3.

Απάντηση άσκησης 4.6
Οι σωστές επιλογές είναι οι 2 και 5. Με την επιλογή 2, διασπάται με την πρώτη `append`

η λίστα `L2` σε δύο λίστες, με τη δεύτερη να είναι η λίστα `L3`, η οποία διασπάται με τη δεύτερη `append` σε δύο λίστες, εκ των οποίων η πρώτη είναι η λίστα `L1`. Με την επιλογή 5, η διάσπαση της `L2` δίνει πρώτο τμήμα την `L3`, της οποίας η διάσπαση δίνει δεύτερο τμήμα τη λίστα `L1`.

Απάντηση άσκησης 4.7
Μπορούμε να ορίσουμε το κατηγόρημα `reverse/2` ως εξής:

```
reverse([], []).
reverse([X|L], R) :- reverse(L, RL), add_at_end(X, RL, R).
```

Η πρώτη πρόταση δηλώνει ότι η αντιστροφή της κενής λίστας δίνει την κενή λίστα, ενώ η δεύτερη πρόταση καλύπτει την αντιστροφή μιας μη κενής λίστας. Σύμφωνα με την πρόταση αυτή, για να αντιστρέψουμε μια μη κενή λίστα, αρκεί να την ξεχωρίσουμε στην κεφαλή και στην ουρά της, να αντιστρέψουμε την ουρά και να τοποθετήσουμε την κεφαλή στο τέλος της αντεστραμμένης ουράς. Μπορούμε να ορίσουμε το κατηγόρημα `add_at_end/3` είτε μέσω της `append/3`:

```
add_at_end(X, L1, L2) :- append(L1, [X], L2).
```

είτε, απευθείας, μέσω αναδρομής:

```
add_at_end(X, [], [X]).
add_at_end(X, [Y|L1], [Y|L2]) :- add_at_end(X, L1, L2).
```

Απάντηση άσκησης 4.8
Μπορούμε να ορίσουμε τα ζητούμενα κατηγορήματα ως εξής:

```
is_list([]).
is_list([_|L]) :- is_list(L).

pref([],_).
pref([X|L1], [X|L2]) :- pref(L1, L2).

suff(L,L).
suff(L1, [_|L2]) :- suff(L1, L2).

consecutive(X, Y, [X,Y|_]).
consecutive(X, Y, [_|L]) :- consecutive(X, Y, L).

equals([], []).
equals([X|L1], L2) :- delete(X, L2, L3), equals(L1, L3).
```

Προβλήματα

Πρόβλημα 4.1
Αν θέλουμε να αναπαραστήσουμε σε Prolog ένα συγκεκριμένο σημείο στον δισδιάστατο χώρο, μπορούμε να το κάνουμε χρησιμοποιώντας ένα συναρτησιακό σύμβολο βαθμού 2, έστω το `point/2`, για να κατασκευάσουμε έναν όρο με ορίσματα τις συντεταγμένες του σημείου. Για παράδειγμα, ο όρος Prolog `point(3,5)` ορίζει μονοσήμαντα το σημείο `(3,5)` του επιπέδου. Με παρόμοιο τρόπο, επειδή ένα τρίγωνο καθορίζεται πλήρως από τις τρεις

κορυφές του, ο όρος `triangle(point(2,-1),point(0,7),point(3,10))` ορίζει μονοσήμαντα ένα τρίγωνο στο επίπεδο. Ένα κύκλος θα μπορούσε να καθορισθεί πλήρως από τον όρο `circle(point(7,1),radius(6))`, υπονοώντας ότι έχει κέντρο το σημείο `(7,1)` και ακτίνα ίση με `6`.

Στην προηγούμενη λογική, προτείνετε όρους Prolog, εξηγώντας τη σημασία των συστατικών τους, για τη μονοσήμαντη αναπαράσταση στον δισδιάστατο χώρο i) τετραπλεύρου, ii) παραλληλογράμμου, iii) ρόμβου, iv) ορθογωνίου παραλληλογράμμου και v) τετραγώνου.

Πρόβλημα 4.2
Μερικές φορές, στο σώμα ενός κανόνα (ή σε μια ερώτηση) Prolog, θέλοντας να ελέγξουμε αν ένα στοιχείο περιέχεται σε μια λίστα και να το διαγράψουμε από αυτήν, τότε γράφουμε:

```
..... :- ....., member(X, L1), delete(X, L1, L2), .....
```

όπου τα `member/2` και `delete/3` είναι τα κατηγορήματα του Παραδείγματος 4.3 και της Άσκησης 4.5, αντίστοιχα. Μπορείτε να σχολιάσετε αυτό το τμήμα προγράμματος; Είναι «σωστός», «λάθος», «καλός», «κακός» προγραμματισμός σε Prolog και γιατί;

Πρόβλημα 4.3
Ορίστε σε Prolog το κατηγόρημα `sel/3`, έτσι ώστε ο στόχος `sel(X, L, R)` να αληθεύει όταν το `X` είναι στοιχείο της λίστας `L` και `R` είναι η λίστα των στοιχείων της `L` που βρίσκονται μετά το `X`. Το κατηγόρημα αυτό να μπορεί να καλείται και με μεταβλητές `X` και `R`. Για παράδειγμα:

```
?- sel(X,[a,b,c],R).
X = a
R = [b,c]    ;
X = b
R = [c]      ;
X = c
R = []
```

Πρόβλημα 4.4
Ορίστε σε Prolog ένα κατηγόρημα `emo/2`, το οποίο, όταν καλείται ως `emo(L1, L2)`, με δεδομένη τη λίστα `L1`, να επιστρέφει στη μεταβλητή `L2` μια αναδιάταξη της `L1`, στην οποία πρώτο να είναι το πρώτο στοιχείο της `L1`, δεύτερο το τελευταίο στοιχείο της `L1`, μετά το δεύτερο στοιχείο της `L1`, μετά το προτελευταίο της κ.ο.κ. Τα παρακάτω παραδείγματα εκτέλεσης αποσαφηνίζουν πλήρως τα ζητούμενα:

```
?- emo([1,2,3,4,5,6,7], L).
L = [1,7,2,6,3,5,4]
?- emo([a,b,c,d,e,f,g,h,i,j], L).
L = [a,j,b,i,c,h,d,g,e,f]
```

Πρόβλημα 4.5
Υλοποιήστε σε Prolog το κατηγόρημα `permutation/2`, έτσι ώστε, όταν αυτό καλείται ως `permutation(L1, L2)`, να επιστρέφει στην `L2` μέσω οπισθοδρόμησης όλες τις δυνατές μεταθέσεις των στοιχείων της δεδομένης λίστας `L1`. Ένα παράδειγμα εκτέλεσης:

```
?- permutation([red,blue,green], P).
P = [red,blue,green]     ;
P = [red,green,blue]     ;
P = [blue,red,green]     ;
P = [blue,green,red]     ;
P = [green,red,blue]     ;
P = [green,blue,red]
no
```

Πρόβλημα 4.6
Θεωρήστε ότι ένας πίνακας μπορεί να αναπαρασταθεί στην Prolog ως μια λίστα από τις γραμμές του, καθεμία εκ των οποίων είναι μια λίστα από τα στοιχεία της. Έχοντας αυτό ως δεδομένο, ορίστε το κατηγόρημα `matr_transp/2`, έτσι ώστε το `matr_transp(M1, M2)` να επιστρέφει στο `M2` τον ανάστροφο πίνακα (αυτόν που προκύπτει με εναλλαγή γραμμών και στηλών) του `M1`. Για παράδειγμα:

```
?- matr_transp([[5,8,9,7,2],[3,6,1,1,4],[2,4,2,8,0]],M).
   M = [[5,3,2],[8,6,4],[9,1,2],[7,1,8],[2,4,0]]
```

Πρόβλημα 4.7
Ορίστε σε Prolog ένα κατηγόρημα `cartprod/2`, το οποίο, όταν καλείται με πρώτο όρισμα ένα σύνολο συνόλων (στη μορφή μιας λίστας λιστών), να επιστρέφει στο δεύτερο όρισμα το καρτεσιανό γινόμενο των συνόλων, ως ένα σύνολο (λίστα) πλειάδων, καθεμία από τις οποίες είναι μια λίστα στοιχείων. Ένα παράδειγμα εκτέλεσης είναι το εξής:

```
?- cartprod([[a,b,c],[1,2],[x,y,z,w]], CP).
CP = [[a,1,x],[a,1,y],[a,1,z],[a,1,w],[a,2,x],[a,2,y],[a,2,z],[a,2,w],
      [b,1,x],[b,1,y],[b,1,z],[b,1,w],[b,2,x],[b,2,y],[b,2,z],[b,2,w],
      [c,1,x],[c,1,y],[c,1,z],[c,1,w],[c,2,x],[c,2,y],[c,2,z],[c,2,w]]
```

Πρόβλημα 4.8
Ορίστε σε Prolog τα παρακάτω κατηγορήματα, με τις προδιαγραφές που δίνονται:

- Το `last(Item, List)` επιστρέφει στο `Item` το τελευταίο στοιχείο της λίστας `List`.
- Το `evenlength(List)` αληθεύει όταν η λίστα `List` έχει άρτιο πλήθος στοιχείων. Προσέξτε ότι δεν γνωρίζετε ακόμη περί ενσωματωμένων κατηγορημάτων αριθμητικής.
- Το `oddlength(List)` αληθεύει όταν η λίστα `List` έχει περιττό πλήθος στοιχείων. Και πάλι, δεν έχετε στη διάθεσή σας ενσωματωμένα κατηγορήματα αριθμητικής.
- Το `palindrome(List)` αληθεύει όταν η λίστα `List` είναι παλινδρομική, δηλαδή ταυτίζεται με την αντίστροφή της.

Βιβλιογραφικές αναφορές

[1] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison Wesley (4th Edition), 2011.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer (5th Edition), 2003.

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

Κεφάλαιο 5
Ενσωματωμένες δυνατότητες και επεκτάσεις της Prolog

Σύνοψη

Στο κεφάλαιο αυτό εξετάζονται αρχικά οι τελεστές, που δεν είναι τίποτε άλλο από συναρτησιακά σύμβολα και κατηγορήματα, τα οποία, αφού γίνει η κατάλληλη δήλωση, μπορούμε να χρησιμοποιήσουμε και με έναν περισσότερο ευανάγνωστο συντακτικά τρόπο από αυτόν με τις παρενθέσεις, που ακολουθούμε, κατά κανόνα, στην Prolog. Επίσης, παρουσιάζονται οι δυνατότητες που παρέχει η Prolog για αριθμητική επεξεργασία, μέσω κατάλληλων ενσωματωμένων κατηγορημάτων, οι αριθμητικοί τελεστές, που είναι συναρτησιακά σύμβολα, όπως επίσης ένα ενσωματωμένο κατηγόρημα για τον υπολογισμό της τιμής αριθμητικών εκφράσεων, και οι τελεστές σύγκρισης, που είναι κατηγορήματα. Ακόμα, γίνεται αναφορά στην αποκοπή, ένα ενσωματωμένο κατηγόρημα που μας επιτρέπει να αποκτήσουμε στοιχειώδη έλεγχο στον μηχανισμό της οπισθοδρόμησης, όπως επίσης στην άρνηση και στον τρόπο με τον οποίο αυτή υποστηρίζεται στην Prolog. Τέλος, μελετάται μια σειρά από άλλα ενσωματωμένα κατηγορήματα, τα οποία είναι χρήσιμα σε περιπτώσεις στις οποίες θέλουμε να εκτελέσουμε κάποιες εργασίες που θα ήταν πολύ δύσκολο ή ακόμα και αδύνατον να υλοποιηθούν απευθείας σε Prolog.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει μελετήσει προσεκτικά τα Κεφάλαια 3 και 4.

5.1 Τελεστές

Μας είναι πλέον γνωστό ότι, με τη βοήθεια συναρτησιακών συμβόλων, μπορούμε να αναπαραστήσουμε στην Prolog σύνθετες οντότητες [1, 2, 3]. Ακολούθως, βλέπουμε ένα χαρακτηριστικό παράδειγμα μιας τέτοιας οντότητας:

Παράδειγμα 5.1
Ας θεωρήσουμε μια αριθμητική έκφραση, όπως «*τη διαφορά του γινομένου του 2 με το 4 από το γινόμενο του 3 με το 5*», που θα την αναπαριστούσαμε με έναν σύνθετο όρο, τον `-(*(3, 5), *(2, 4))`. Έχουμε, δηλαδή, το συναρτησιακό σύμβολο «–», με το οποίο κατασκευάζουμε έναν όρο που παριστάνει μια διαφορά, το πρώτο όρισμα μείον το δεύτερο. Στο συγκεκριμένο παράδειγμα, και τα ορίσματα της διαφοράς συνιστούν σύνθετους όρους, που είναι γινόμενα με συναρτησιακό σύμβολο το «*». □

Το πρόβλημα με την αναπαράσταση της αριθμητικής έκφρασης του Παραδείγματος 5.1

σε Prolog εντοπίζεται στο γεγονός ότι είναι αρκετά δυσανάγνωστη, το οποίο θα ήταν σίγουρα εντονότερο αν η έκφραση ήταν πιο πολύπλοκη. Είναι προφανές ότι θα ήταν πιο χρήσιμο αν μπορούσαμε να γράψουμε την έκφραση ως εξής: `3*5-2*4`. Ευτυχώς, αυτό μπορεί να γίνει, γιατί η Prolog έχει ορίσει ότι τα συναρτησιακά σύμβολα «-» και «*» (αλλά και τα «+» και «/») είναι **τελεστές**, και μάλιστα **ενδοθεματικοί τελεστές**. Ένας ενδοθεματικός τελεστής op είναι πάντα βαθμού 2. Έτσι, για να τον εφαρμόσουμε στα ορίσματα a και b, εκτός από την παραδοσιακή γραφή $op(a, b)$, μπορούμε να χρησιμοποιήσουμε και την $a\ op\ b$.

Εδώ θα μπορούσατε να ρωτήσετε γιατί η αριθμητική έκφραση `3*5-2*4` ερμηνεύεται ως `(3*5)-(2*4)` και όχι ως `3*(5-2*4)` ή ως `(3*5-2)*4`. Η απάντηση βρίσκεται σε μια ιδιότητα που έχει κάθε τελεστής, την **προτεραιότητά** του. Για τη συγκεκριμένη περίπτωση, στην Prolog έχει οριστεί ότι ο τελεστής «-» έχει προτεραιότητα 500, ενώ ο τελεστής «*» έχει προτεραιότητα 400. Αν ένας σύνθετος όρος είναι διατυπωμένος με τη βοήθεια περισσοτέρων του ενός τελεστών, για την αποσαφήνιση της πραγματικής δομής του όρου, εφαρμόζεται ο εξής κανόνας:

Ο κύριος τελεστής ενός σύνθετου όρου είναι αυτός που έχει τη μεγαλύτερη προτεραιότητα. Αυτός ο τελεστής είναι το συναρτησιακό σύμβολο με το οποίο δομείται ο όρος στην παραδοσιακή του γραφή με τις παρενθέσεις. Ενδέχεται και τα ορίσματά του να είναι σύνθετοι όροι με τελεστές, οπότε και γι' αυτά εφαρμόζεται αναδρομικά ο ίδιος κανόνας.

Οπότε, καταλαβαίνουμε γιατί ο σύνθετος όρος `3*5-2*4` είναι μια διαφορά και όχι ένα γινόμενο ($500 > 400$). Ο προηγούμενος όμως κανόνας δεν μπορεί να εφαρμοστεί, όταν στον σύνθετο όρο, μέγιστη προτεραιότητα έχουν περισσότεροι του ενός τελεστές. Ας δούμε το εξής παράδειγμα:

Παράδειγμα 5.2
Ποια είναι η πραγματική δομή του όρου `2*3*5`; Είναι η `2*(3*5)` ή η `(2*3)*5`; Δηλαδή, ποιος είναι ο κύριος τελεστής του όρου, το πρώτο «*» ή το δεύτερο; Ας σημειώσουμε ότι η ερμηνεία που αποδίδουμε στο σύμβολο «*», αυτή του πολλαπλασιασμού, και το γεγονός ότι η πράξη αυτή είναι προσεταιριστική δεν αφορά την αποσαφήνιση της πραγματικής δομής του όρου. Αυτή πρέπει, ούτως ή άλλως, να γίνει, ανεξάρτητα από την ιδιότητα της προσεταιριστικότητας στη συγκεκριμένη περίπτωση. Εξάλλου, σε κάποια άλλη περίπτωση τελεστή, θα μπορούσε η ιδιότητα αυτή να μην υπάρχει, οπότε θα είχαμε πραγματικά πρόβλημα. □

Απάντηση στο ερώτημα του Παραδείγματος 5.2 δίνει μια ιδιότητα του τελεστή, που είναι ο **τύπος** του. Για τους ενδοθεματικούς τελεστές, υπάρχουν τρεις τύποι, οι `yfx`, `xfy` και `xfx`. Το «`f`» στα σύμβολα αυτά παριστάνει τον τελεστή, ενώ οι χαρακτήρες πριν από και μετά το «`f`» παριστάνουν το πρώτο και το δεύτερο όρισμα του τελεστή, αντίστοιχα. Ο κανόνας που καθορίζει πώς αποσαφηνίζεται η πραγματική δομή ενός όρου, όταν αυτός έχει περισσότερους του ενός τελεστές με τη μεγαλύτερη προτεραιότητα, είναι ο εξής:

Αν ένα όρισμα ενός τελεστή «f» είναι τύπου «x» και το όρισμα αυτό είναι σύνθετος όρος με τελεστές, τότε ο κύριος τελεστής του ορίσματος πρέπει να έχει προτεραιότητα μικρότερη από την προτεραιότητα του τελεστή. Αν το όρισμα είναι τύπου «y», τότε ο κύριος τελεστής του πρέπει να έχει προτεραιότητα μικρότερη ή ίση από την προτεραιότητα του τελεστή. Αν το όρισμα είναι όρος, απλός ή σύνθετος, χωρίς όμως τελεστές, τότε μπορούμε να θεωρήσουμε ότι ο κύριος τελεστής του έχει προτεραιότητα 0, οπότε δεν υπάρχει κανένα πρόβλημα, είτε το όρισμα είναι τύπου «x» είτε τύπου «y». Προτεραιότητα 0 θεωρούμε ότι έχει κάθε όρος, ακόμα και αν περιέχει τελεστές, όταν περιβάλλεται από παρενθέσεις.

Ο προηγούμενος κανόνας μπορεί να χρησιμοποιηθεί για την αποσαφήνιση της δομής του όρου `2*3*5` ως εξής: Στην Prolog, ο τελεστής «`*`» (αλλά και οι «`+`», «`-`» και «`/`») είναι ορισμένος με τύπο `yfx`. Αυτό σημαίνει ότι το δεύτερο «`*`» στον όρο `2*3*5` είναι ο κύριος τελεστής του, γιατί, αν ήταν το πρώτο, τότε το δεύτερο όρισμά του (το `3*5`) θα είχε κύριο τελεστή με προτεραιότητα 400, ίση με την προτεραιότητα του τελεστή, κάτι που είναι αδύνατον, επειδή το όρισμα αυτό είναι τύπου «`x`». Συνεπώς, ο όρος `2*3*5` έχει δομή την `(2*3)*5`. Με άλλα λόγια, μπορούμε να πούμε ότι τελεστές τύπου `yfx` είναι αριστερά προσεταιριστικοί. Αντίστοιχα, τελεστές τύπου `xfy` είναι δεξιά προσεταιριστικοί, ενώ, αν ένας τελεστής op είναι τύπου `xfx`, τότε η έκφραση $a\ op\ b\ op\ c$ είναι συντακτικά λάθος.

Εκτός από τους ενδοθεματικούς τελεστές, υπάρχουν επίσης οι **προθεματικοί** και οι **μεταθεματικοί**. Ένας προθεματικός τελεστής op (πάντα βαθμού 1) μπορεί να είναι τύπου `fy` ή `fx` και ο όρος $op\ a$ να έχει δομή την $op(a)$. Αν ο τελεστής είναι τύπου `fy`, τότε ο όρος $op\ op\ a$ επιτρέπεται και σημαίνει τον $op(op(a))$, όπως και οποιοσδήποτε αριθμός επαναλήψεων του τελεστή. Αν ο τελεστής είναι τύπου `fx`, τότε ο όρος $op\ op\ a$ είναι συντακτικά λάθος. Ένας μεταθεματικός τελεστής op (επίσης, πάντα βαθμού 1) μπορεί να είναι τύπου `yf` ή `xf` και ο όρος $a\ op$ να έχει δομή την $op(a)$. Όπως ισχύει και για τους προθεματικούς τελεστές, επαναλήψεις στους μεταθεματικούς `yf` τελεστές επιτρέπονται (και το $a\ op\ op$ είναι το $op(op(a))$), ενώ στους μεταθεματικούς `xf` τελεστές απαγορεύονται.

Σε αυτό το σημείο πρέπει να διευκρινίσουμε ότι, αν και τα παραδείγματα που αναφέραμε και οι κανόνες που διατυπώσαμε αφορούν τελεστές οι οποίοι είναι συναρτησιακά σύμβολα, επιτρέπεται να έχουμε και κατηγορήματα που να έχουν δηλωθεί κατάλληλα και να συντάσσονται ως τελεστές. Για να δηλώσουμε ένα συναρτησιακό σύμβολο ή κατηγόρημα $oper$ ως τελεστή τύπου $type$, με προτεραιότητα $prec$, αρκεί να ζητήσουμε την ικανοποίηση του στόχου `op(`$prec$`,`$type$`,`$oper$`)`. Το `op/3` είναι ένα ενσωματωμένο κατηγόρημα, που χρησιμοποιείται για τη δήλωση τελεστών. Μπορούμε να το χρησιμοποιήσουμε είτε απευθείας σε μια ερώτηση, οπότε η δήλωση του τελεστή ισχύει από εκείνη τη χρονική στιγμή και μετά, είτε να βάλουμε το στόχο αυτό υπό μορφή οδηγίας στην αρχή ενός αρχείου που περιέχει ένα πρόγραμμα Prolog, στο οποίο χρησιμοποιείται ο τελεστής αυτός. Για παράδειγμα, αν θέλουμε να ορίσουμε τον τελεστή `my_op` ως ενδοθεματικό `xfy` τελεστή με προτεραιότητα 150, μπορούμε στην αρχή του αρχείου στο οποίο θα τον χρησιμοποιήσουμε να βάλουμε την οδηγία:

```
:- op(150, xfy, my_op).
```

Το `:-` είναι μια υπόδειξη προς το μεταγλωττιστή, ότι αυτό που ακολουθεί δεν είναι ένα γεγονός ή ένας κανόνας, αλλά μια ερώτηση προς εκτέλεση, η οποία είναι βέβαια στη συγκεκριμένη περίπτωση μια δήλωση/οδηγία για τον ορισμό ενός τελεστή. Η δυνατότητα όμως να βάζουμε και ερωτήσεις μέσα σε αρχεία προς μεταγλώττιση είναι γενικότερη και πολλές φορές πάρα πολύ χρήσιμη.

Ένα άλλο ενσωματωμένο κατηγόρημα, το `current_op/3`, μας πληροφορεί για τις ιδιότητες τελεστών που έχουν οριστεί, είτε από εμάς είτε από την ίδια την Prolog. Ας δούμε τη χρήση του:

```
?- op(150, xfy, my_op).
yes
?- current_op(P, T, *).
P = 400
T = yfx       ;
no
?- current_op(P, T, my_op).
P = 150
T = xfy       ;
no
?-
```

Τέλος, θα πρέπει να σημειώσουμε ότι σε κάθε σύστημα Prolog είναι ορισμένο ένα πλήθος από τελεστές, συμπεριλαμβανομένων των «`:-`» (ως `xfx` και ως `fx` τελεστής με προτεραιότητα 1200) και «`,`» (ως `xfy` με προτεραιότητα 1000). Οι υπόλοιποι τελεστές, που είναι ήδη ορισμένοι, έχουν συνήθως προτεραιότητα ≥ 200. Οι συνηθέστεροι από τους εκ των προτέρων ορισμένους τελεστές στα συστήματα Prolog φαίνονται στον Πίνακα 5.1. Στους τελεστές αυτούς θα αναφερθούμε σε επόμενες ενότητες. Ωστόσο, μπορούμε να τους επαναορίσουμε, αν θέλουμε, με διαφορετικό τρόπο. Κάτι τέτοιο, όμως, πρέπει να γίνεται εν γνώσει του κινδύνου που ενέχει, ο οποίος δεν είναι άλλος από την πιθανή μεταβολή της σημασίας ενός προγράμματος. Ούτως ή άλλως, βέβαια, δεν είναι καλή ιδέα να ορίζουμε εμείς τελεστές με προτεραιότητα ≥ 1000, γιατί τότε επεμβαίνουμε ουσιαστικά στη σύνταξη των κανόνων. Εν πάση περιπτώσει, αν θέλουμε οι τελεστές μας να μην παρεμβάλλονται, ως προς την προτεραιότητά τους, στους ήδη ορισμένους τελεστές, θα πρέπει να τους ορίζουμε με προτεραιότητα < 200. Μερικές όμως φορές είναι χρήσιμο να ορίζουμε τελεστές με προτεραιότητα μεταξύ 200 και 1000.

Προτεραιότητα	Τύπος	Τελεστής
1200	`xfx`	`:-`
1200	`fx`	`:-`
1100	`xfy`	`;`
1000	`xfy`	`,`
900	`fy`	`\+`, `not`
700	`xfx`	`is`, `=`, `\=`, `<`, `=<`, `>`, `>=`, `=:=`, `=\=`, `=..`
500	`fx`	`+`, `-`
500	`yfx`	`+`, `-`
400	`yfx`	`*`, `/`, `//`
300	`xfx`	`mod`
200	`xfy`	`^`

Πίνακας 5.1: Οι συνηθέστεροι εκ των προτέρων ορισμένοι τελεστές.

Στο σημείο αυτό, πρέπει να διευκρινιστεί ότι η δυνατότητα που παρέχει η Prolog για ορισμό και χρήση τελεστών συμβάλλει απλώς στη συγγραφή ευανάγνωστων προγραμμάτων και τίποτα περισσότερο. Δηλαδή, ακόμα και αν δεν είχαμε στη διάθεσή μας τους τελεστές, η Prolog θα μας παρείχε τις ίδιες ουσιαστικά εκφραστικές δυνατότητες.

Άσκηση 5.1
Έστω ότι έχουμε ορίσει κάποιους τελεστές με τις οδηγίες που ακολουθούν:

```
:- op(180, xfx, likes).
:- op(180, xfx, hate).
:- op(190, fy, doesnt).
:- op(180, xfx, equals_to).
:- op(140, yfx, plus).
:- op(140, yfx, minus).
:- op(120, xfy, or).
:- op(110, xfy, and).
:- op(160, fy, father_of).
:- op(100, yf, ++).
```

Μετά τις προηγούμενες δηλώσεις, ακολουθεί ένα πρόγραμμα Prolog διατυπωμένο ως εξής:

```
tarzan likes jane.
father_of john likes eggs and fish and meat and nothing_else.
father_of father_of mary likes books or tv and cinema.
X likes Y :- doesnt X hate Y.
X ++ equals_to X plus 1.
X ++ ++ equals_to X plus 2.
1 plus 2 plus 3 equals_to 6.
X minus Y equals_to Z :- Y plus Z equals_to X.
```

Πώς θα έπρεπε να είχε γραφεί το πρόγραμμα αν δεν είχαμε δηλώσει τους τελεστές που χρησιμοποιούμε σε αυτό;

Άσκηση 5.2
Δώστε κατάλληλες δηλώσεις τελεστών και χρησιμοποιήστε τες για να ορίσετε το κατηγόρημα `member/2` του Παραδείγματος 4.3.

5.2 Αριθμητική

Στην Άσκηση 4.3 ζητήθηκαν οι ορισμοί μιας σειράς από κατηγορήματα, τα οποία υλοποιούσαν σε Prolog τις τέσσερις πράξεις της αριθμητικής, καθώς και κάποιες άλλες μαθηματικές συναρτήσεις. Για τον ορισμό των κατηγορημάτων αυτών, ήταν απαραίτητο να αναπαρασταθούν οι ακέραιοι αριθμοί όχι με τα κλασικά αραβικά σύμβολα 0, 1, 2, 3, . . ., αλλά με τους όρους Prolog `0`, `s(0)`, `s(s(0))`, `s(s(s(0)))`, . . ., αντίστοιχα. Αντιλαμβανόμαστε όμως ότι, παρότι έχουμε τη δυνατότητα με αυτόν τον τρόπο να χρησιμοποιούμε στα προγράμματά μας αριθμούς και να κάνουμε πράξεις μεταξύ τους, η μέθοδος αυτή δεν είναι ιδιαίτερα εξυπηρετική. Θα προτιμούσαμε να υπήρχε η δυνατότητα να αναπαριστούμε τους αριθμούς με τον κλασικό τρόπο, αλλά και να μπορούμε να υπολογίζουμε την τιμή μιας αριθμητικής έκφρασης διατυπωμένης ως ένας σύνθετος όρος με τελεστές, μέσω των κλασικών συμβόλων των πράξεων. Τα προηγούμενα σημαίνουν ότι, για παράδειγμα, αν είχαμε την αριθμητική έκφραση `3*5-2*4`, θα θέλαμε να μας προσφέρεται από την Prolog ένας τρόπος να υπολογίσουμε την τιμή αυτής της έκφρασης, δηλαδή το `7`. Ευτυχώς, ο τρόπος αυτός υπάρχει και παρέχεται από το ενσωματωμένο κατηγόρημα `is/2`, το οποίο είναι ορισμένο και ως ενδοθεματικός `xfx` τελεστής με προτεραιότητα 700. Η ικανοποίηση ενός στόχου με κατηγόρημα το `is/2`, πρώτο όρισμα μια μεταβλητή και δεύτερο όρισμα μια αριθμητική έκφραση έχει αποτέλεσμα τον υπολογισμό της τιμής της έκφρασης και της ενοποίησης της τιμής αυτής με τη μεταβλητή. Δηλαδή:

```
?- X is 3*5-2*4.
X = 7
?-
```

Σε μια αριθμητική έκφραση μπορούμε να έχουμε ακέραιους αριθμούς και, σχεδόν σε όλα τα συστήματα Prolog μπορούμε να έχουμε ακόμα και πραγματικούς (οι πραγματικοί ξεχωρίζουν από τους ακέραιους, επειδή περιλαμβάνουν και υποδιαστολή, π.χ. `2.35`). Οι πράξεις σε μια αριθμητική έκφραση παριστάνονται με τους αριθμητικούς τελεστές `+`, `-`, `*` και `/`. Όλοι είναι `yfx` τελεστές, οι δύο πρώτοι με προτεραιότητα 500 και οι δύο τελευταίοι με προτεραιότητα 400. Επίσης, οι `+` και `-` είναι δηλωμένοι και ως `fx` προθεματικοί τελεστές με προτεραιότητα 500, έτσι ώστε να μπορούν να παίξουν και το ρόλο προσήμου. Ο τελεστής `/` χρησιμοποιείται για την πραγματική διαίρεση, δηλαδή το `21/8` ισούται με `2.625`. Πολλά συστήματα Prolog προσφέρουν και τον τελεστή `//` (ενδοθεματικό `yfx` με προτεραιότητα 400), που χρησιμοποιείται για το πηλίκο της ακέραιας διαίρεσης, δηλαδή το `21//8` ισούται με `2`, τον τελεστή `mod` (ενδοθεματικό `xfx` με προτεραιότητα 300), που χρησιμοποιείται για το υπόλοιπο της ακέραιας διαίρεσης, δηλαδή το `21 mod 8` ισούται με `5`, καθώς και τον τελεστή `^` (ενδοθεματικό `xfy` με προτεραιότητα 200), που χρησιμοποιείται για την ύψωση σε δύναμη, δηλαδή το `3^4` ισούται με `81`. Οι αριθμητικοί τελεστές έχουν περιληφθεί στον Πίνακα 5.1.

Εκτός από το ενσωματωμένο κατηγόρημα `is/2` και τους αριθμητικούς τελεστές, η Prolog παρέχει και ορισμένους **τελεστές σύγκρισης**, που είναι ενσωματωμένα κατηγορήματα. Τα κατηγορήματα αυτά, ορισμένα και ως ενδοθεματικοί `xfx` τελεστές με προτεραιότητα 700, χρησιμοποιούνται για τη σύγκριση των τιμών των αριθμητικών εκφράσεων. Είναι τα `>/2`, `</2`, `>=/2`, `=</2`, `=:=/2` και `=\=/2`, που σημαίνουν, αντίστοιχα, **μεγαλύτερο**, **μικρότερο**, **μεγαλύτερο ή ίσο**, **μικρότερο ή ίσο**, **ίσο** και **διάφορο** (Πίνακας 5.1). Ο στόχος $expr_1\ compop\ expr_2$ επιτυγχάνει, αν οι τιμές των αριθμητικών εκφράσεων $expr_1$ και $expr_2$, αφού υπολογιστούν και συγκριθούν, έχουν τη σχέση που δηλώνει ο τελεστής σύγκρισης $compop$. Δείτε πώς εφαρμόζονται αυτά στην πράξη:

```
?- 3*5-2*4 > 21 mod 8.
yes
?- 4*7//9 =:= 8-4.
no
?- 5+2.4 =< 25/3.
yes
?-
```

Πρέπει να σημειώσουμε ότι, τόσο η αριθμητική έκφραση που είναι δεύτερο όρισμα σε ένα `is/2`, όσο και οι αριθμητικές εκφράσεις που συμμετέχουν σε ένα στόχο με τελεστή σύγκρισης μπορεί να περιλαμβάνουν όχι μόνο αριθμούς, αλλά και μεταβλητές. Πρέπει όμως κατά τη στιγμή ικανοποίησης του στόχου οι μεταβλητές να έχουν πάρει τιμή έναν αριθμό. Σε αντίθετη περίπτωση, το σύστημα θα επιστρέψει μήνυμα λάθους.

Με τις αριθμητικές δυνατότητες που παρέχει η Prolog και περιγράψαμε σε αυτήν την ενότητα, έχουμε πλέον μεγάλη ευελιξία στην υλοποίηση διαφόρων χρήσιμων λειτουργιών.

Παράδειγμα 5.3

Στο Παράδειγμα 1.2 είδαμε πώς μπορούμε με δηλωτικό τρόπο να ορίσουμε τι σημαίνει μήκος μιας λίστας. Εκεί δώσαμε και την υλοποίηση των κατάλληλων αξιωμάτων, τόσο σε Prolog όσο και σε Haskell, χωρίς όμως να εξηγήσουμε τίποτα περισσότερο. Τώρα όμως

μπορούμε να καταλάβουμε τον τρόπο με τον οποίο είναι υλοποιημένο σε Prolog το κατηγόρημα `length/2`, που είναι κλασικό κατηγόρημα διαχείρισης λιστών και χρήσιμο σε πάρα πολλές περιπτώσεις. Ας επαναλάβουμε εδώ τον ορισμό του:

```
length([], 0).
length([X|L], N) :- length(L, M), N is M+1.
```

Υποβάλλοντας σχετικές ερωτήσεις, παίρνουμε και τις κατάλληλες απαντήσεις, όπως:

```
?- length([a,b,c,d], N).
N = 4
?-
```

Παράδειγμα 5.4
Ένας εναλλακτικός τρόπος ορισμού του μήκους λίστας είναι αυτός που βασίζεται στη χρήση του λεγόμενου **συσσωρευτή**. Η ιδέα είναι να αρχικοποιήσουμε ένα συσσωρευτή με τιμή 0 και μετά να διατρέξουμε τη λίστα και να προσθέσουμε σε αυτόν 1, για κάθε στοιχείο που συναντάμε. Το `length/2`, το οποίο ζητάμε, ορίζεται μέσω ενός κατηγορήματος `length/3`, με ορίσματα όχι μόνο τη λίστα της οποίας το μήκος μάς ενδιαφέρει, αλλά επίσης την τρέχουσα τιμή του συσσωρευτή και την τελική τιμή, η οποία θα είναι το μήκος που θα υπολογίσουμε. Δείτε αυτήν την εναλλακτική υλοποίηση:

```
length(L, N) :- length(L, 0, N).

length([], N, N).
length([_|L], N1, N) :- N2 is N1+1, length(L, N2, N).
```

Ο συγκεκριμένος ορισμός διαφέρει από εκείνον του Παραδείγματος 5.3, επειδή είναι λιγότερο δηλωτικός. Είναι όμως περισσότερο οικονομικός σε μνήμη, επειδή ο αναδρομικός κανόνας του έχει αναδρομή ουράς, δηλαδή ο στόχος στο σώμα του που έχει κατηγόρημα ίδιο με αυτό της κεφαλής είναι ο τελευταίος. Αυτό όμως δεν συμβαίνει στον αναδρομικό κανόνα του Παραδείγματος 5.3.

Παράδειγμα 5.5
Ας ορίσουμε ένα κατηγόρημα `sumlist/2`, το οποίο, όταν του δίνουμε στο πρώτο όρισμα μια λίστα από αριθμούς, να μας επιστρέφει στο δεύτερο όρισμα το άθροισμά τους.

Ο εντελώς δηλωτικός ορισμός του `sumlist/2` είναι ο εξής:

```
sumlist([], 0).
sumlist([X|L], S) :- sumlist(L, S1), S is X+S1.
```

Θα μπορούσαμε όμως να ορίσουμε το `sumlist/2` και μέσω συσσωρευτή, ως εξής:

```
sumlist(L, S) :- sumlist(L, 0, S).

sumlist([], S, S).
sumlist([X|L], S1, S) :- S2 is X+S1, sumlist(L, S2, S).
```

□

Θα κλείσουμε αυτήν την ενότητα με μια σειρά από ασκήσεις, που θα σας βοηθήσουν να εμπεδώσετε καλύτερα το περιεχόμενό της.

Άσκηση 5.3
Πώς πρέπει να συμπληρωθεί ο ορισμός του κατηγορήματος `max/3` που ακολουθεί, έτσι ώστε αυτό να επιστρέφει στο τρίτο όρισμά του το μέγιστο μεταξύ του πρώτου και του δεύτερου ορίσματός του;

```
max(X, Y, [     ] ) :- X >= Y.
max(X, Y, [     ] ) :- X [     ] Y.
```

Άσκηση 5.4
Γνωρίζουμε ότι ο μέγιστος κοινός διαιρέτης δύο αριθμών είναι ο μέγιστος κοινός διαιρέτης του μικρότερου από αυτούς και της διαφοράς του από τον μεγαλύτερο και ότι ο μέγιστος κοινός διαιρέτης ενός αριθμού και του εαυτού του είναι ο ίδιος αριθμός. Εκμεταλλευτείτε αυτήν τη γνώση, για να συμπληρώσετε τον ορισμό του κατηγορήματος `gcd/3` που ακολουθεί, το οποίο επιστρέφει στο τρίτο όρισμά του τον μέγιστο κοινό διαιρέτη του πρώτου και του δεύτερου ορίσματός του:

```
gcd( [     ] , [     ] , X).
gcd(X, Y, [     ] ) :- X [     ] Y, Y1 is Y-X, gcd(X, [     ] , D).
gcd(X, Y, D) :- [     ] > [     ] , [                    ] .
```

Άσκηση 5.5
Με βάση την απάντησή σας στην Άσκηση 4.2, διατυπώστε σε Prolog τις εξής ερωτήσεις:

1. «*Υπάρχει κάποιος άνεργος που να έχει γεννηθεί μετά το 1980;*»
2. «*Υπάρχει κάποιος άνδρας που να παίρνει σύνταξη μεταξύ 1000 και 1300 ευρώ;*»
3. «*Υπάρχει κάποια γυναίκα ανύπαντρη που να έχει γεννηθεί ανήμερα τα Χριστούγεννα και ο μισθός της να είναι τουλάχιστον 2000 ευρώ;*»

Άσκηση 5.6
Ορίστε σε Prolog τα εξής κατηγορήματα:

- `prod/2`: Το `prod(L, P)` επιστρέφει στο `P` το γινόμενο των αριθμών που είναι στοιχεία της λίστας `L`.
- `factorial/2`: Το `factorial(N, F)` επιστρέφει στο `F` το παραγοντικό του μη αρνητικού ακεραίου `N`.
- `maxlist/2`: Το `maxlist(L, M)` επιστρέφει στο `M` το μέγιστο στοιχείο της λίστας `L`.
- `n_th/3`: Το `n_th(N, L, X)` επιστρέφει στο `X` το `N`-οστό στοιχείο της λίστας `L`.
- `first_n/3`: Το `first_n(N, L1, L2)` επιστρέφει στο `L2` τη λίστα που περιέχει τα `N` πρώτα στοιχεία της λίστας `L1`.
- `ordered/1`: Το `ordered(L)` επιτυγχάνει όταν η λίστα `L` είναι ταξινομημένη σε αύξουσα σειρά.

Άσκηση 5.7
Ορίστε ένα κατηγόρημα `primes/2`, έτσι ώστε το `primes(N, L)` να επιστρέφει στο `L` τη λίστα όλων των πρώτων αριθμών που είναι μικρότεροι ή ίσοι του `N`. Μπορείτε να εφαρμόσετε

το κόσκινο του Ερατοσθένη, δηλαδή, αφού κατασκευάσετε τη λίστα όλων των ακέραιων από το 2 έως το N, να διαγράψετε από αυτήν όλους τους αριθμούς που είναι πολλαπλάσια κάποιου άλλου αριθμού στη λίστα (και, άρα, δεν μπορεί να είναι πρώτοι).

5.3 Έλεγχος οπισθοδρόμησης

Στην Άσκηση 5.4 σας ζητήθηκε να συμπληρώσετε τον ορισμό ενός κατηγορήματος gcd/3 που θα ήταν σε θέση να υπολογίζει τον μέγιστο κοινό διαιρέτη δύο θετικών ακέραιων. Αφού επαναλάβουμε σε αυτό το σημείο τη σωστή απάντηση, ας τη σχολιάσουμε λίγο:

```
                % gcd/3: Έκδοση 1
gcd(X, X, X).
gcd(X, Y, D) :- X < Y, Y1 is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- X > Y, gcd(Y, X, D).
```

Η δεύτερη πρόταση του gcd/3, για την εύρεση του μέγιστου κοινού διαιρέτη των X και Y, καλύπτει την περίπτωση στην οποία το X είναι μικρότερο του Y. Οπότε, αρκεί να βρεθεί ο μέγιστος κοινός διαιρέτης του X και του Y-X. Για την περίπτωση αυτή, δεν υπάρχει λόγος να δοκιμαστεί έπειτα από οπισθοδρόμηση και η τρίτη πρόταση, γιατί, ακόμα και αν γίνει τέτοια απόπειρα, τελικά δεν θα ευδοκιμήσει, αφού ο πρώτος στόχος του σώματος της πρότασης αυτής (X > Y) θα αποτύχει. Βέβαια, αυτό δεν μπορεί να το γνωρίζει η Prolog εκ των προτέρων. Συνεπώς, σε κάθε περίπτωση κατά την οποία θα επιλέγεται η δεύτερη πρόταση για το gcd/3 και αυτή θα επιτυγχάνει, έπειτα από οπισθοδρόμηση θα δοκιμάζεται και η τρίτη πρόταση. Παρότι αυτή η πρόταση δεν θα καταφέρει ποτέ να επιτύχει, υπάρχει σίγουρα κάποια επίπτωση στην απόδοση του προγράμματος. Βέβαια, αν η δεύτερη πρόταση αποτύχει, επειδή δεν ισχύει X < Y, τότε θα πρέπει να επιλεγεί η τρίτη πρόταση, για την οποία είμαστε σίγουροι ότι είναι η κατάλληλη.

Για να αντιμετωπιστεί το πρόβλημα της περιττής οπισθοδρόμησης, συνεπώς και της μείωσης της απόδοσης ενός προγράμματος, η Prolog παρέχει ένα ενσωματωμένο κατηγόρημα, το !/0, που λέγεται **αποκοπή**. Με τη βοήθεια της αποκοπής, το πρόγραμμα για το gcd/3 μπορεί, καταρχάς, να βελτιωθεί, ως εξής:

```
                % gcd/3: Έκδοση 2
gcd(X, X, X).
gcd(X, Y, D) :- X < Y, !, Y1 is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- X > Y, gcd(Y, X, D).
```

Το ! που βρίσκεται στο σώμα της δεύτερης πρότασης υποδεικνύει ότι, αν επιλεγεί η πρόταση και ο στόχος X < Y επιτύχει, τότε δεν υπάρχει λόγος να δοκιμαστεί επόμενη πρόταση, για να ικανοποιηθεί ο στόχος που την ενεργοποίησε. Ουσιαστικά, η επιτυχία του στόχου μας έχει δεσμευτεί με την επιτυχία αυτής της πρότασης.

Φυσικά, αν αρχικά επιλεγεί η δεύτερη πρόταση για την ικανοποίηση ενός στόχου, αλλά αποτύχει, εξαιτίας της αποτυχίας του X < Y, δεν υπάρχει άλλος τρόπος ικανοποίησης του στόχου μας από την επιλογή της τρίτης πρότασης. Τότε όμως, ο στόχος X > Y στο σώμα της πρότασης αυτής είναι περιττός, γιατί πάντα θα ισχύει. Συνεπώς, μπορεί να διαγραφεί. Ό,τι συζητήσαμε για τη σχέση της δεύτερης πρότασης με την τρίτη ισχύει και για τη σχέση της πρώτης με τη δεύτερη. Δηλαδή, αν ζητηθεί να βρεθεί ο μέγιστος κοινός διαιρέτης δύο ίσων αριθμών, τότε η πρώτη πρόταση θα είναι, προφανώς, η κατάλληλη να εφαρμοστεί, αλλά κατά την οπισθοδρόμηση, που ίσως προκύψει εξαιτίας της αποτυχίας άλλου μεταγενέστερου στόχου, θα δοκιμαστεί και η επόμενη. Αυτό είναι, προφανώς, άσκοπο, γιατί, παρότι η

πρόταση δεν θα επιτύχει, θα υπάρξει πάλι κάποια επίδραση στην απόδοση. Για να αποφύγουμε αυτήν την περιττή οπισθοδρόμηση, δεν έχουμε παρά να μετατρέψουμε την πρώτη πρόταση, που είναι γεγονός, σε κανόνα και να βάλουμε στο σώμα του ένα `!`. Δηλαδή:

```
                % gcd/3: Έκδοση 3
gcd(X, X, X) :- !.
gcd(X, Y, D) :- X < Y, !, Y1 is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- gcd(Y, X, D).
```

Δίνουμε τον κανόνα λειτουργίας της αποκοπής, ως εξής:

Η αποκοπή `!/0` είναι ένα ενσωματωμένο κατηγόρημα που πάντοτε επιτυγχάνει κατά την εμπρόσθια φορά του ελέγχου. Κατά την οπισθοδρόμηση όμως, η λειτουργία της επιβάλλει την αποτυχία του στόχου που ενεργοποίησε την πρόταση στο σώμα της οποίας βρίσκεται η αποκοπή. Με άλλα λόγια, αν το κατηγόρημα της κεφαλής της πρότασης αυτής είναι `pred`, δεν γίνεται οπισθοδρόμηση ούτε στους στόχους του σώματος που βρίσκονται πριν από την αποκοπή, ούτε σε επόμενες προτάσεις για το `pred`.

Αν συγκρίνουμε τις εκδόσεις 1 και 3 του ορισμού για το `gcd/3`, μπορούμε να σχολιάσουμε ότι η έκδοση 1 είναι πολύ πιο δηλωτική από την 3, αν και, όπως εξετάσαμε ήδη, λιγότερο αποδοτική. Είναι σαφές ότι η έκδοση 3 είναι λίγο δυσανάγνωστη. Φυσικά, αυτό σχετίζεται άμεσα με το γεγονός ότι είναι περισσότερο διαδικαστική, αλλά και πιο αποδοτική. Πραγματικά, αυτή είναι η παγίδα στην οποία μπορεί να μας παρασύρει η αποκοπή. Στην προσπάθειά μας να βελτιώσουμε την απόδοση των προγραμμάτων μας, μερικές φορές κάνουμε υπερβολική χρήση της αποκοπής, με συνέπεια να χάνουμε ένα από τα βασικά πλεονεκτήματα του λογικού προγραμματισμού, τον δηλωτικό του χαρακτήρα. Από την άλλη πλευρά, σίγουρα η αποκοπή είναι σε πάρα πολλές περιπτώσεις χρήσιμη για την αποφυγή περιττών οπισθοδρομήσεων, αλλά και σε άλλες περιπτώσεις είναι άκρως απαραίτητη για την υλοποίηση κάποιας λειτουργίας, όπως θα δούμε στο Παράδειγμα 5.7.

Πάντως, για να έχουμε μια εκτίμηση της πιθανής επικινδυνότητας των αποκοπών, μπορούμε να τις διακρίνουμε σε δύο κατηγορίες. Η πρώτη κατηγορία περιλαμβάνει τις αποκοπές που απλώς έχουν προστεθεί σε ένα πρόγραμμα για να βελτιώσουν την απόδοσή του, μέσω της αποφυγής περιττών οπισθοδρομήσεων. Σε αυτήν την περίπτωση, οι απαντήσεις που δίνει το πρόγραμμα σε κάποια ερώτηση είναι οι ίδιες, ανεξάρτητα αν έχουμε ή δεν έχουμε αποκοπές, οι οποίες ονομάζονται **πράσινες αποκοπές**. Μπορούμε όμως να έχουμε αποκοπές που, αν τις βγάλουμε, το πρόγραμμα να αλλάζει συμπεριφορά, οι οποίες ονομάζονται **κόκκινες αποκοπές**. Η αποκοπή στην έκδοση 2 του προγράμματος για το κατηγόρημα `gcd/3` είναι πράσινη, επειδή η ύπαρξή της βοηθά να αποφευχθεί η διερεύνηση της τρίτης πρότασης, όταν έχει ήδη χρησιμοποιηθεί η δεύτερη και ο έλεγχος `X < Y` ήταν επιτυχής. Εάν δεν υπήρχε αυτή η αποκοπή, τότε θα είχαμε απλώς μια επίπτωση στην απόδοση του προγράμματος. Όμως αυτό θα ήταν σωστό. Αντίθετα, οι αποκοπές στην έκδοση 3 είναι κόκκινες, επειδή η ύπαρξή τους είναι ουσιαστική για την ορθότητα του προγράμματος. Εάν δεν υπήρχαν, τότε η τρίτη πρόταση θα εφαρμοζόταν έπειτα από οπισθοδρόμηση σε κάθε περίπτωση, ακόμα και αν δεν ίσχυε το `X > Y`. Αυτό είναι, προφανώς, λάθος, όταν έχει ήδη εφαρμοστεί επιτυχώς για το δεδομένο στόχο είτε η πρώτη είτε η δεύτερη πρόταση. Μεταξύ των δύο ειδών αποκοπών, οι κόκκινες αποκοπές είναι αυτές που επηρεάζουν σημαντικά την αναγνωσιμότητα των προγραμμάτων και μειώνουν το δηλωτικό τους χαρακτήρα. Υπ' αυτήν την έννοια, οι συγκεκριμένες αποκοπές πρέπει να χρησιμοποιούνται με φειδώ, ενώ οι πράσινες μπορούν να χρησιμοποιούνται με άνεση, γιατί είναι περισσότερο αβλαβείς.

Παράδειγμα 5.6
Μπορείτε να δώσετε τον ορισμό του `max/3`, που σας ζητήθηκε να τον συμπληρώσετε στην Άσκηση 5.3, εκμεταλλευόμενοι την αποκοπή, ως εξής:

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

Δηλαδή, η δεύτερη πρόταση θα χρησιμοποιηθεί αν αποτύχει η πρώτη, εξαιτίας αποτυχίας της συνθήκης `X >= Y`, χωρίς να χρειάζεται να γίνει ο έλεγχος `X < Y`, και δεν θα γίνει οπισθοδρόμηση σε αυτήν αν επιτύχει η πρώτη.

Παράδειγμα 5.7
Έστω ότι θέλουμε να ορίσουμε ένα κατηγόρημα `flatten/2`, το οποίο, όταν καλείται με πρώτο όρισμα μια λίστα που περιέχει απλά στοιχεία, αλλά και άλλες λίστες, που και αυτές, με τη σειρά τους, είναι δυνατόν να περιέχουν άλλες λίστες κ.ο.κ., να επιστρέφει στο δεύτερο όρισμά του μια λίστα η οποία περιλαμβάνει όλα τα στοιχεία που περιέχονται στο πρώτο όρισμα, ανεξάρτητα από το επίπεδο, σε ένα επίπεδο. Το κατηγόρημα αυτό θα μπορούσε να οριστεί ως εξής:

```
flatten([], []) :- !.
flatten([X|L], F) :- !, flatten(X, FX), flatten(L, FL),
                     append(FX, FL, F).
flatten(X, [X]).
```

Οι αποκοπές στην πρώτη και στη δεύτερη πρόταση είναι απολύτως απαραίτητες, επειδή αποκλείουν την περίπτωση να χρησιμοποιηθεί η τρίτη πρόταση, όταν το πρώτο όρισμα, με το οποίο καλείται η `flatten/2`, είναι λίστα. Κάτι τέτοιο θα ήταν ανεπιθύμητο, γιατί τότε η τρίτη πρόταση θα επέστρεφε μια λίστα με ένα στοιχείο, τη λίστα που της δόθηκε ως πρώτο όρισμα. Βέβαια, αυτό θα συνέβαινε σε οπισθοδρόμηση, αφού πρώτα είχε χρησιμοποιηθεί σωστά η κατάλληλη από τις δύο προηγούμενες προτάσεις. Δηλαδή, θα παίρναμε ως πρώτη απάντηση στην ερώτησή μας τη σωστή, αλλά η Prolog θα μας έδινε και ένα πλήθος από εναλλακτικές απαντήσεις μετά, το οποίο θα ήταν όμως λάθος. Η τρίτη πρόταση χρειάζεται γιατί, όταν φτάσουμε σε ένα απλό στοιχείο, θα πρέπει στην `append/3` να δοθεί η μονομελής λίστα που περιλαμβάνει αυτό το στοιχείο, για να δουλέψει σωστά. Έτσι, μπορούμε να ρωτήσουμε:

```
?- flatten([[a,b],c,d,[[[e,f],g],[h]],[],[i],j], L).
L = [a,b,c,d,e,f,g,h,i,j]
?-
```

Άσκηση 5.8
Θεωρήστε τα κατηγορήματα `intersection/3` και `union/3` με τις εξής προδιαγραφές:

- Το `intersection(L1, L2, L3)` επιστρέφει στο `L3` τη λίστα που περιέχει τα κοινά στοιχεία των λιστών `L1` και `L2`. Υποθέτουμε ότι τόσο η `L1` όσο και η `L2` δεν περιέχουν διπλές εμφανίσεις στοιχείων.
- Το `union(L1, L2, L3)` επιστρέφει στο `L3` τη λίστα που περιέχει τα κοινά και μη κοινά στοιχεία των λιστών `L1` και `L2`. Υποθέτουμε ότι τόσο η `L1` όσο και η `L2` δεν περιέχουν διπλές εμφανίσεις στοιχείων και ότι το ίδιο πρέπει να συμβαίνει στο αποτέλεσμα `L3`.

Συμπληρώστε τους ορισμούς του `intersection/3` και του `union/3` που ακολουθούν και προσθέστε, όπου χρειάζεται, αποκοπές, για να είναι σωστοί οι ορισμοί:

```
intersection( [          ] ).
intersection([X|L1], L2, [X|L3]) :- member(X, L2),
                                    intersection(L1, L2, L3).
intersection([_|L1], L2, L3) :- intersection(L1, L2, L3).

union( [          ] ).
union([X|L1], L2, L3) :- member(X, L2),
                         union(L1, L2, L3).
union([X|L1], L2, [X|L3]) :- union(L1, L2, L3).
```

Άσκηση 5.9
Δείτε πάλι την απάντηση που δώσαμε στην Άσκηση 5.7, για την εύρεση των πρώτων αριθμών που είναι μικρότεροι ή ίσοι, δεδομένου θετικού ακέραιου. Πιστεύετε ότι θα μπορούσε ο ορισμός του κατηγορήματος `filter/3` να ήταν λίγο διαφορετικά διατυπωμένος, συγκεκριμένα με χρήση της αποκοπής;

Άσκηση 5.10
Ορίστε σε Prolog τα εξής κατηγορήματα:

- `delete_one/3`: Το `delete_one(X, L1, L2)` επιστρέφει στο `L2` τη λίστα που προκύπτει αν διαγραφεί η πρώτη εμφάνιση του στοιχείου `X` στη λίστα `L1`. Αν η `L1` δεν περιέχει το `X`, τότε η `L2` είναι η ίδια με την `L1`.
- `delete_all/3`: Το `delete_all(X, L1, L2)` επιστρέφει στο `L2` τη λίστα που προκύπτει αν διαγραφούν όλες οι εμφανίσεις του στοιχείου `X` στη λίστα `L1`. Αν η `L1` δεν περιέχει το `X`, τότε η `L2` είναι η ίδια με την `L1`.
- `substitute/4`: Το `substitute(X, L1, Y, L2)` επιστρέφει στο `L2` τη λίστα που προκύπτει αν αντικατασταθούν όλες οι εμφανίσεις του στοιχείου `X` στη λίστα `L1` με το στοιχείο `Y`. Και εδώ θα θέλαμε, αν η `L1` δεν περιέχει το `X`, τότε η `L2` να είναι η ίδια με την `L1`.
- `rem_dupls/2`: Το `rem_dupls(L1, L2)` επιστρέφει στο `L2` τη λίστα που προκύπτει αν από τη λίστα `L1` διαγραφούν όλες οι πολλαπλές εμφανίσεις στοιχείων της.
- `difference/3`: Το `difference(L1, L2, L3)` επιστρέφει στο `L3` τη λίστα που περιέχει όσα στοιχεία της λίστας `L1` δεν ανήκουν στη λίστα `L2`. Υποθέστε ότι τόσο η `L1` όσο και η `L2` δεν περιέχουν διπλές εμφανίσεις στοιχείων.

5.4 Άρνηση

Πολλές φορές είναι χρήσιμο να υποβάλουμε κάποια ερώτηση σε ένα πρόγραμμα Prolog ή να έχουμε ένα στόχο στο σώμα ενός κανόνα που να ικανοποιείται όταν κάτι δεν ισχύει. Δηλαδή, θα θέλαμε να είχαμε τη δυνατότητα να εκφράσουμε την **άρνηση** στην Prolog. Αυτή η δυνατότητα υπάρχει μέσω ενός ενσωματωμένου κατηγορήματος που προσφέρεται, του `\+/1` (τα περισσότερα συστήματα Prolog παρέχουν και το συνώνυμό του `not/1`), το οποίο θα περιγράψουμε στην ενότητα αυτή. Καθώς η λειτουργία της άρνησης στην Prolog

είναι αρκετά ιδιόρρυθμη, θα πρέπει να δώσετε πολύ μεγάλη προσοχή στην κατανόηση της έννοιας αυτής, για να αποφύγετε προβλήματα και παρεξηγήσεις στο μέλλον.

Η άρνηση στην Prolog βασίζεται στην **υπόθεση του κλειστού κόσμου**. Σύμφωνα με την υπόθεση αυτή, όταν έχουμε διατυπώσει ένα πρόγραμμα, αυτό περιλαμβάνει ό,τι είναι αληθινό, τίποτα λιγότερο από αυτό. Δηλαδή, αν κάτι δεν περιέχεται στο πρόγραμμα, υποθέτουμε ότι δεν ισχύει. Αν το σκεφτούμε λίγο, αυτή είναι μια πάρα πολύ «σκληρή» υπόθεση. Είναι αρκετά δύσκολο, αν όχι αδύνατον, σε κάποια δεδομένη κατάσταση του περιβάλλοντος κόσμου να έχουμε εξασφαλίσει την αλήθεια της υπόθεσης αυτής, δηλαδή να έχουμε καταγράψει πλήρως τι ισχύει στον κόσμο αυτό. Εν πάση περιπτώσει, για να μπορέσουμε να εκφράσουμε την άρνηση στην Prolog, πρέπει να δεχτούμε την ισχύ αυτής της υπόθεσης.

Το κατηγόρημα `\+/1` που παρέχει η Prolog λειτουργεί με τον εξής τρόπο: Ο στόχος `\+(Goal)` επιτυγχάνει αν δεν μπορεί να αποδειχθεί ο στόχος `Goal` και, αντίστροφα, ο στόχος `\+(Goal)` αποτυγχάνει αν ο στόχος `Goal` αποδεικνύεται. Σύμφωνα με τη λογική αυτή, αν κάτι δεν αποδεικνύεται, η άρνησή του αληθεύει. Για παράδειγμα, αν σε ένα πρόγραμμα έχουμε καταχωρήσει πληροφορίες για διάφορους ανθρώπους, συγκεκριμένα για το επάγγελμά τους μέσω του κατηγορήματος `profession/2`, αλλά δεν έχουμε καταγράψει ρητά ότι ο *Αισχύλος* ήταν συγγραφέας τραγωδιών και ρωτήσουμε «*είναι αλήθεια ότι ο Αισχύλος δεν ήταν συγγραφέας τραγωδιών;*», θα πάρουμε ως απάντηση ένα μεγαλοπρεπές `yes` (με ό,τι επιπτώσεις μπορεί να έχει αυτό στην πίστη μας για τις καλλιτεχνικές δραστηριότητες του *Αισχύλου*)!

Το `\+/1` είναι ενσωματωμένο κατηγόρημα, δηλαδή ορισμένο εσωτερικά στην Prolog, αλλά, αν δεν ήταν, θα μπορούσαμε να το ορίσουμε και εμείς μέσω του `!/0` και δύο άλλων ενσωματωμένων κατηγορημάτων του `call/1` και του `fail/0`, ως εξής:

```
\+(Goal) :- call(Goal), !, fail.
\+(Goal).
```

Το κατηγόρημα `call/1` δέχεται όρισμα ένα στόχο τον οποίο καλεί. Αν ο στόχος αυτός επιτύχει, τότε θα επιτύχει και το κατηγόρημα. Αν ο στόχος αποτύχει, τότε θα αποτύχει και το `call/1`. Το κατηγόρημα `fail/0` πάντα αποτυγχάνει. Συνεπώς, ο προηγούμενος ορισμός λειτουργεί ως εξής: Αν κληθεί ένας στόχος `\+(Goal)`, γίνεται απόπειρα να ικανοποιηθεί με την πρώτη πρόταση για το `\+/1`. Καλείται ο στόχος `Goal` με το `call(Goal)` και, αν επιτύχει, περνάμε μετά το `!`, το `fail` προκαλεί οπισθοδρόμηση, αλλά, λόγω του `!`, το `\+(Goal)` αποτυγχάνει. Αν το `call(Goal)` αποτύχει, γίνεται οπισθοδρόμηση στη δεύτερη πρόταση για το `\+/1`, οπότε το `\+(Goal)` επιτυγχάνει. Έτσι, με αυτήν την υλοποίηση του `\+/1`, έχουμε τη λειτουργικότητα που περιγράψαμε προηγουμένως.

Ας σημειώσουμε επίσης ότι το κατηγόρημα `\+/1` είναι ορισμένο και ως προθεματικός `fy` τελεστής με προτεραιότητα 900, οπότε είναι δυνατόν να χρησιμοποιείται και με τη σύνταξη `\+ Goal`, εκτός από την `\+(Goal)`.

Πολλές φορές, η άρνηση μπορεί να μας βοηθήσει να απαλλαγούμε από αποκοπές που έχουμε σε ένα πρόγραμμα. Αυτή βέβαια η απαλλαγή είναι λίγο εικονική, επειδή η άρνηση ορίζεται ουσιαστικά μέσω του `!/0`. Παρ' όλα αυτά όμως, με τη χρήση της άρνησης, αντί για το `!/0`, τα προγράμματά μας γίνονται περισσότερο δηλωτικά. Δείτε το επόμενο παράδειγμα:

Παράδειγμα 5.8
Στην Άσκηση 5.10 ζητήθηκε να υλοποιήσετε κάποια κατηγορήματα, μεταξύ των οποίων και το `rem_dupls/2`. Η υλοποίηση που προτάθηκε στην απάντηση της άσκησης ήταν η εξής:

```
rem_dupls([], []).
rem_dupls([X|L1], L2) :- member(X, L1), !, rem_dupls(L1, L2).
rem_dupls([X|L1], [X|L2]) :- rem_dupls(L1, L2).
```

Μπορούμε να απαλλαγούμε από το `!` στο σώμα της δεύτερης πρότασης, αν εξασφαλίσουμε ότι η τρίτη θα αφεθεί να λειτουργήσει μόνο αν το `X` δεν ανήκει στην `L1`. Δηλαδή:

```
rem_dupls([], []).
rem_dupls([X|L1], L2) :- member(X, L1), rem_dupls(L1, L2).
rem_dupls([X|L1], [X|L2]) :- \+ member(X, L1), rem_dupls(L1, L2).
```

Το πρόγραμμα αυτό είναι περισσότερο δηλωτικό από το αρχικό, αλλά έχει τον εξής πλεονασμό στη λειτουργία του: Αν κάποια στιγμή η δεύτερη πρόταση αποτύχει, επειδή δεν ισχύει το `member(X, L1)`, θα γίνει απόπειρα να αποδειχθεί το ζητούμενο με την τρίτη πρόταση. Οπότε, στην προσπάθεια απόδειξης του `\+ member(X, L1)`, θα κληθεί πάλι ο στόχος `member(X, L1)`, ο οποίος γνωρίζουμε βέβαια ότι θα αποτύχει, αλλά αυτό θα αποδειχθεί για άλλη μία φορά. Έτσι, η άρνησή του θα επιτύχει και θα συνεχίσει η εφαρμογή της τρίτης πρότασης. Ακόμα και όταν ο στόχος `member(X, L1)` στη δεύτερη πρόταση επιτυγχάνει, σε οπισθοδρόμηση θα δοκιμάζεται και η τρίτη πρόταση, η οποία δεν θα καταφέρει όμως ποτέ να φτάσει στο τέλος, γιατί ο στόχος `\+ member(X, L1)` θα αποτυγχάνει, αφού για άλλη μία φορά θα αποδεικνύεται η αλήθεια του `member(X, L1)`. Συνοψίζοντας, χρησιμοποιώντας την άρνηση, πληρώνουμε τη δηλωτικότητα του προγράμματός μας με τη μειωμένη απόδοσή του.

Παράδειγμα 5.9
Έστω ότι έχουμε το εξής πολύ απλό πρόγραμμα Prolog:

```
man(john).
woman(mary).
```

Γνωρίζουμε, δηλαδή, ότι ο *Γιάννης* είναι άνδρας και ότι η *Μαίρη* είναι γυναίκα. Έστω ότι θέλουμε να μάθουμε αν «*υπάρχει κάποιος που να μην είναι άνδρας*», οπότε μας φαίνεται πολύ λογικό να υποβάλουμε την ερώτηση αυτή όπως φαίνεται στη συνέχεια. Βλέπετε επίσης και την απάντηση που μας δίνει η Prolog:

```
?- \+ man(X).
no
?-
```

Προφανώς, δεν περιμέναμε τη συγκεκριμένη απάντηση, γιατί υπάρχει σίγουρα η *Μαίρη*, που δεν είναι άνδρας. Πού εντοπίζεται το πρόβλημα; Στο γεγονός ότι η ερώτηση που υποβάλαμε δεν αντιπροσωπεύει αυτήν την οποία θέλαμε να υποβάλουμε, αλλά το αν «*είναι αλήθεια ότι δεν υπάρχει κάποιος που να είναι άνδρας*». Η απάντηση στην ερώτηση αυτή είναι, βέβαια, αρνητική, αφού υπάρχει ο *Γιάννης*, που είναι άνδρας. □

Από το Παράδειγμα 5.9 αποκομίζουμε το εξής δίδαγμα: χρειάζεται πολλή προσοχή στη χρήση της άρνησης με στόχους που περιέχουν μεταβλητές. Ενώ με οποιοδήποτε στόχο χωρίς άρνηση που περιέχει μεταβλητές ρωτάμε αν υπάρχει συνδυασμός τιμών των μεταβλητών, τέτοιος ώστε ο στόχος να είναι αληθής, αν ο στόχος έχει άρνηση και είναι με μεταβλητές, τότε ρωτάμε αν για οποιονδήποτε συνδυασμό τιμών των μεταβλητών ο στόχος είναι ψευδής. Αυτό είναι τελείως διαφορετικό από το αν υπάρχει συνδυασμός τιμών των μεταβλητών,

τέτοιος ώστε ο στόχος να είναι ψευδής. Επειδή σίγουρα θα μας ενδιέφερε να γνωρίζουμε πώς θα μπορούσαμε να υποβάλουμε σωστά την ερώτηση που μας ενδιέφερε στο Παράδειγμα 5.9, ας δούμε το επόμενο παράδειγμα:

Παράδειγμα 5.10
Για να μπορέσουμε να διατυπώσουμε σωστά την ερώτηση «*υπάρχει κάποιος που να μην είναι άνδρας;*», θα πρέπει να έχουμε και άλλο ένα κατηγόρημα, έστω το `human/1`, με το οποίο να έχουμε δηλώσει ποιο είναι το πεδίο ορισμού επάνω στο οποίο θέλουμε να υποβάλουμε την ερώτησή μας. Έτσι, έχουμε το εξής πρόγραμμα:

```
human(john).
human(mary).
man(john).
woman(mary).
```

Οπότε, υποβάλλοντας τη σωστή ερώτηση, παίρνουμε και τη σωστή απάντηση:

```
?- human(X), \+ man(X).
X = mary
?-
```

Ουσιαστικά, αν τα προγράμματά μας έχουν στόχους με άρνηση, πρέπει να είμαστε προσεκτικοί τη στιγμή της ικανοποίησης αυτών των στόχων ώστε οι μεταβλητές που τυχόν περιέχουν να έχουν πάρει τιμή. Είναι πολύ απίθανο, όχι όμως και αδύνατον, να μας ενδιαφέρει να ζητάμε την ικανοποίηση ενός στόχου με άρνηση ο οποίος περιέχει μεταβλητές χωρίς τιμή.

Άσκηση 5.11
Συμπληρώστε τον ημιτελή ορισμό του κατηγορήματος `disjoint/2` που ακολουθεί, έτσι ώστε ο στόχος `disjoint(L1, L2)` να επιτυγχάνει όταν οι λίστες `L1` και `L2` δεν έχουν κοινά στοιχεία:

```
disjoint( [   ] , _).
disjoint([X|L1], L2) :- [          ](X, L2), [                ] .
```

Άσκηση 5.12
Επαναδιατυπώστε τους ορισμούς για τα κατηγορήματα `intersection/3` και `union/3` που σας ζητήθηκε να συμπληρώσετε στην Άσκηση 5.8, χρησιμοποιώντας την άρνηση, αντί για την αποκοπή.

Άσκηση 5.13
Ορίστε σε Prolog ένα κατηγόρημα `ifthenelse/3`, το οποίο να συμπεριφέρεται ως δομή ελέγχου **if-then-else** των διαδικαστικών γλωσσών προγραμματισμού. Δηλαδή, ο στόχος `ifthenelse(Cond, ThenGoal, ElseGoal)` να καλεί το στόχο `ThenGoal`, αν η συνθήκη `Cond` είναι αληθής, αλλιώς να καλεί το στόχο `ElseGoal`. Ορίστε το `ifthenelse/3` με δύο τρόπους, με αποκοπή και με άρνηση.

5.5 Ενσωματωμένα κατηγορήματα

Στις προηγούμενες ενότητες αυτού του κεφαλαίου είδαμε ένα πλήθος από ενσωματωμένα κατηγορήματα της Prolog, που υποστηρίζουν διάφορες χρήσιμες λειτουργίες. Είδαμε το κατηγόρημα `op/3`, για τη δήλωση τελεστών, και το `current_op/3`, για την ανάκτηση των χαρακτηριστικών δηλωμένων τελεστών. Επίσης, είδαμε το `is/2`, για τον υπολογισμό της τιμής μιας αριθμητικής έκφρασης, και τα κατηγορήματα-τελεστές, για τη σύγκριση των τιμών των αριθμητικών εκφράσεων `>/2`, `</2`, `>=/2`, `=</2`, `=:=/2` και `=\=/2`. Για τον έλεγχο της οπισθοδρόμησης, αναφερθήκαμε στο `!/0` και τέλος στο κατηγόρημα της άρνησης `\+/1` και, παρεμπιπτόντως, στα ενσωματωμένα κατηγορήματα `call/1` και `fail/0`.

Στην ενότητα αυτή θα δούμε ένα πλήθος από άλλα ενσωματωμένα κατηγορήματα της Prolog. Η αναφορά μας δεν θα είναι εξαντλητική, τουναντίον μάλιστα. Θα περιγράψουμε μόνο ενσωματωμένα κατηγορήματα που ανήκουν στο πρότυπο της γλώσσας, κοινά σε όλα τα συστήματα Prolog, τα οποία είναι εξαιρετικά χρήσιμα. Για να πάρετε μια ιδέα σχετικά με την πληθώρα των ενσωματωμένων κατηγορημάτων τα οποία υποστηρίζονται από το σύστημα Prolog που χρησιμοποιείτε για την εξάσκησή σας, δεν έχετε παρά να ρίξετε μια ματιά στο συνοδευτικό εγχειρίδιο.

`;/2`

Το ενσωματωμένο κατηγόρημα `;/2` είναι μια ευκολία που μας προσφέρεται για να εκφράσουμε διαζευγμένους στόχους (είτε σε μια ερώτηση που υποβάλλουμε είτε στο σώμα ενός κανόνα). Δηλαδή, ενώ η ερμηνεία του κανόνα «`a :- b, c.`» ορίζει ότι «*αν ισχύει το `b` και το `c`, τότε ισχύει το `a`*», ο κανόνας «`a :- b; c.`» δηλώνει ότι «*αν ισχύει το `b` ή το `c`, τότε ισχύει το `a`*».

Όπως καταλαβαίνουμε από τη σύνταξή του, το κατηγόρημα `;/2` είναι ορισμένο και ως ενδοθεματικός τελεστής, και μάλιστα τύπου `xfy` και προτεραιότητας 1100. Συνεπώς, αφού έχει προτεραιότητα μεγαλύτερη από τον τελεστή `,/2` της σύζευξης (προτεραιότητας 1000), τότε ο κανόνας «`a :- b, c; d, e.`» σημαίνει τον «`a :- (b, c) ; (d, e).`».

Το κατηγόρημα `;/2` διευκολύνει τη σύνταξη των προγραμμάτων, αλλά η ύπαρξή του δεν είναι απολύτως απαραίτητη. Αυτό συμβαίνει γιατί ο κανόνας «`a :- b; c.`» μπορεί εντελώς ισοδύναμα να γραφεί ως δύο κανόνες, τους «`a :- b.`» και «`a :- c.`».

Πάντως, πρέπει να επισημανθεί ότι η κατάχρηση της διάζευξης κάνει τα προγράμματα πολύ δυσανάγνωστα και γι' αυτό πρέπει να υπάρχει κάποιο μέτρο στην εκμετάλλευσή της. Συνήθως, είναι προτιμότερο να γράφουμε περισσότερους και απλούστερους κανόνες, παρά λιγότερους και πιο πολύπλοκους.

`=/2` `\=/2`

Το ενσωματωμένο κατηγόρημα `=/2` προκαλεί την ενοποίηση των δύο ορισμάτων του. Φυσικά, αν τα ορίσματά του δεν ενοποιούνται, αποτυγχάνει. Είναι ορισμένο και ως ενδοθεματικός `xfx` τελεστής με προτεραιότητα 700, με αποτέλεσμα να μπορεί να χρησιμοποιηθεί ως εξής:

```
?- full_name(ann, X) = full_name(Y, smith).
X = smith
Y = ann
?- [a,b,c,d,e] = [X,Y|Z].
X = a
Y = b
Z = [c,d,e]
```

```
?- p([]) = p([[]]).
no
?-
```

Το κατηγόρημα `\=/2` είναι η άρνηση του `=/2`, δηλαδή αποτυγχάνει όταν τα δύο ορίσματά του ενοποιούνται. Η ενοποίηση όμως δε γίνεται τελικά, αφού έχουμε αποτυχία. Το `\=/2` επιτυγχάνει όταν τα ορίσματά του δεν ενοποιούνται. Είναι και αυτό ορισμένο ως ενδοθεματικός `xfx` τελεστής με προτεραιότητα 700. Δείτε κάποια παραδείγματα χρήσης του:

```
?- [a,b,c] \= [X,Y,Z].
no
?- p([]) \= p([[]]).
yes
?- date_born(17, jan, X) \= date_born(4, Y, 1964).
X = _
Y = _
?-
```

Στην τελευταία ερώτηση, μη σας ξενίζουν οι τιμές των δύο μεταβλητών `X` και `Y`. Επειδή τα δύο ορίσματα του `\=/2` δεν ήταν ενοποιήσιμα, ο στόχος έπρεπε να επιτύχει, όμως οι μεταβλητές δεν ήταν δυνατόν να πάρουν κάποια τιμή.[1]

Το κατηγόρημα `\=/2` δεν είναι άλλο από το `diff/2`, στο οποίο αναφερθήκαμε για πρώτη φορά στην Ενότητα 3.1. Θα μπορούσαμε εκείνο το κατηγόρημα να το είχαμε ορίσει όχι εξαντλητικά, αλλά μέσω του `\=/2`. Δηλαδή:

```
diff(X, Y) :- X \= Y.
```

`assert/1` `retract/1`

Η Prolog προσφέρει μια δυνατότητα, αλλά ταυτόχρονα κρύβει μια πολύ μεγάλη παγίδα, την οποία είναι απίθανο να συναντήσουμε σε κάποια άλλη γλώσσα προγραμματισμού. Αναφερόμαστε στη δυνατότητα τροποποίησης του προγράμματος κατά τη φάση της εκτέλεσής του ή, για να ακριβολογούμε σε σχέση με την Prolog, κατά τη φάση υπολογισμού της απάντησης σε μια ερώτηση.

Με το κατηγόρημα `assert/1` προστίθεται στο πρόγραμμα, κατά την εκτέλεση του αντίστοιχου στόχου, η πρόταση που του έχει δοθεί ως όρισμα. Σε σχέση με τις ήδη υπάρχουσες προτάσεις που έχουν το ίδιο κατηγόρημα στην κεφαλή τους, η προστιθέμενη με το `assert/1` εισάγεται έπειτα από αυτές. Αν θέλουμε για κάποιο λόγο να προστεθεί πριν από τις υπάρχουσες, μπορούμε να χρησιμοποιήσουμε μια παραλλαγή του, το κατηγόρημα `asserta/1`.[2]

Με το κατηγόρημα `retract/1`, διαγράφεται από το πρόγραμμα η πρώτη πρόταση, η οποία ενοποιείται με το όρισμα που δόθηκε στο `retract/1`. Η ενοποίηση αυτή γίνεται πραγματικά, οπότε μεταβλητές που τυχόν υπάρχουν στο στόχο με το `retract/1` ενδέχεται να έχουν αποτιμηθεί. Σε οπισθοδρόμηση, αν υπάρχει και άλλη πρόταση που να ενοποιείται με το όρισμα του `retract/1`, διαγράφεται και αυτή. Αυτό θα συνεχιστεί έως ότου δεν υπάρχουν άλλες προτάσεις για διαγραφή, οπότε το `retract/1` αποτυγχάνει. Δείτε μια σειρά από ερωτήσεις και τα αποτελέσματά τους:

[1] Η μορφή των απαντήσεων που δίνουν τα διάφορα συστήματα Prolog όταν μια ερώτηση που υποβάλλουμε επιτυγχάνει, αλλά παραμένουν μεταβλητές χωρίς να έχουν πάρει τιμή, διαφέρει από περίπτωση σε περίπτωση.

[2] Υπάρχει και το `assertz/1`, που είναι συνώνυμο του `assert/1`.

```
?- assert(p(1)).
yes
?- assert(p(2)).
yes
?- p(X).
X = 1       ;
X = 2
?- retract(p(X)).
X = 1       ;
X = 2       ;
no
?- p(X)
no
?-
```

Εκτός από γεγονότα, μπορούμε να εισαγάγουμε και να διαγράψουμε δυναμικά και κανόνες, μόνο που πρέπει, όταν σε κάποιο `assert/1` ή `retract/1` βάζουμε όρισμα έναν κανόνα, να τον περικλείουμε σε παρενθέσεις. Για παράδειγμα:

```
?- assert((q :- r)).
yes
?- assert(r).
yes
?- q.
yes
?- retract((q :- r)).
yes
?- q.
no
?- retract(r).
yes
?-
```

Στην αρχή της περιγραφής μας για τα ενσωματωμένα κατηγορήματα `assert/1` και `retract/1` χαρακτηρίσαμε παγίδα τη δυνατότητα που προσφέρει η Prolog για δυναμική τροποποίηση του προγράμματος. Πράγματι, η δυνατότητα αυτή προσφέρει μεγάλη ευελιξία, αλλά ταυτόχρονα είναι και ένας επικίνδυνος πειρασμός, στον οποίο, αν υποκύπτουμε συχνά, θα καταφέρουμε απλώς τα προγράμματά μας να μην μπορούν να γίνουν κατανοητά από κανέναν, ούτε από εμάς τους ίδιους. Αν για τη χρήση της αποκοπής και, σε μικρότερο βαθμό, της διάζευξης, σας επιστήσαμε την προσοχή, για να μην κάνετε κατάχρησή τους, το ίδιο ισχύει, με πολλαπλάσια ένταση, για τα ενσωματωμένα κατηγορήματα δυναμικής τροποποίησης του προγράμματος. Προσέξτε λοιπόν πολύ καλά τη χρήση αυτών των κατηγορημάτων, γιατί και αυτά «*βλάπτουν πολύ σοβαρά τον καλό προγραμματισμό σε Prolog*».

`write/1` `nl/0`

Μέχρι τώρα, βλέπαμε τα αποτελέσματα από την εκτέλεση των ερωτήσεων που υποβάλλαμε σε προγράμματα Prolog ως τιμές μεταβλητών, που μας δίνει το ίδιο το σύστημα της Prolog, εκείνων των μεταβλητών οι οποίες περιέχονταν στην ερώτησή μας. Μερικές φορές όμως, επιθυμούμε να μαθαίνουμε την τιμή κάποιας μεταβλητής, όχι κατ' ανάγκη μόνο της αρχικής μας ερώτησης, στη φάση ικανοποίησης της ερώτησης αυτής. Για το σκοπό αυτό, κάθε σύστημα Prolog προσφέρει ένα πλούσιο ρεπερτόριο από ενσωματωμένα κατηγορήματα εξόδου, τα πιο διάσημα εκ των οποίων είναι το `write/1` και το `nl/0`.

Το κατηγόρημα `write/1` εκτυπώνει στην έξοδο το όρισμά του. Αν το όρισμα είναι μεταβλητή, εκτυπώνεται η τιμή αυτής της μεταβλητής. Με το `nl/0`, κάνουμε στην έξοδο αλλαγή γραμμής. Ιδού σχετικά παραδείγματα:

```
?- write('Hello world!'), nl.
Hello world!
yes
?- X = 'Hello ', Y = world, Z = !, W = ' How are you?',
   write(X), write(Y), write(Z), write(W), nl.
Hello world! How are you?
X = 'Hello '
Y = world
Z = !
W = ' How are you?'
?-
```

Μέσω των προηγούμενων ερωτήσεων, μας δίνεται η ευκαιρία να πούμε δύο λόγια για τους συντακτικούς κανόνες που πρέπει να διέπουν τις σταθερές. Μια σταθερά (το ίδιο ισχύει για τα συναρτησιακά σύμβολα και τα κατηγορήματα) κατασκευάζεται ως μια ακολουθία από χαρακτήρες, που μπορεί να είναι γράμματα,[3] αριθμοί ή το σύμβολο _, με τον πρώτο χαρακτήρα να είναι πεζό γράμμα. Εναλλακτικά, μπορεί όλοι οι χαρακτήρες μιας σταθεράς να είναι ειδικά σύμβολα, από τα +, −, *, /, <, >, =, :, ., & ή ~. Τέλος, μια σταθερά μπορεί να κατασκευαστεί από οποιαδήποτε ακολουθία χαρακτήρων, χωρίς περιορισμούς, αρκεί να την περικλείσουμε σε απλά εισαγωγικά `'`. Αυτήν την τελευταία εκδοχή χρησιμοποιήσαμε στις προηγούμενες ερωτήσεις. Όσον αφορά τη σύνταξη των μεταβλητών, αυτές μπορούν να κατασκευαστούν ως ακολουθίες από χαρακτήρες που μπορεί να είναι γράμματα, αριθμοί ή το σύμβολο _, με τον πρώτο χαρακτήρα να είναι κεφαλαίο γράμμα ή το _.

`findall/3`

Όπως έχουμε δει, όταν για ένα στόχο υπάρχουν περισσότερες της μιας απαντήσεις, η Prolog μπορεί να μας δώσει μέσω οπισθοδρόμησης μία προς μία τις απαντήσεις αυτές. Ας θεωρήσουμε το εξής πρόγραμμα:

```
likes(john, beans).
likes(john, mary).
likes(jack, mary).
```

Υποβάλλοντας ερωτήσεις, παίρνουμε τις αναμενόμενες απαντήσεις:

```
?- likes(X, mary).
X = john     ;
X = jack
?-
```

Πολλές φορές όμως θα θέλαμε, αντί να πάρουμε μία-μία τις απαντήσεις για ένα στόχο μέσω οπισθοδρόμησης, να τις έχουμε όλες μαζί σε μια λίστα. Αυτή η δυνατότητα μας παρέχεται με το ενσωματωμένο κατηγόρημα `findall/3`. Το κατηγόρημα αυτό, όταν καλείται ως `findall(T, G, L)`, επιστρέφει στο `L` τη λίστα των αποτιμήσεων του όρου `T` που προκύπτουν από επιτυχείς ικανοποιήσεις του στόχου `G`. Υποτίθεται ότι, για να έχει πρακτικό νόημα αυτή η χρήση, ο όρος `T` πρέπει να περιλαμβάνει μεταβλητές από το στόχο `G` (στις συνηθέστερες βέβαια περιπτώσεις ο `T` είναι μια μεταβλητή του `G`). Δείτε σχετικά παραδείγματα ερωτήσεων που απευθύνονται στο προηγούμενο πρόγραμμα:

[3] Τα περισσότερα συστήματα Prolog χρησιμοποιούν γράμματα του λατινικού αλφαβήτου.

```
?- findall(X, likes(X, mary), L).
X = _
L = [john, jack]
?- findall(likedby(X, Y), likes(Y, X), L).
X = _
Y = _
L = [likedby(beans, john), likedby(mary, john), likedby(mary, jack)]
?- findall(X, likes(X, X), L).
X = _,
L = []
?-
```

var/1 nonvar/1 atom/1 integer/1

Το ενσωματωμένο κατηγόρημα `var/1` επιτυγχάνει αν το όρισμά του είναι μεταβλητή που δεν έχει πάρει τιμή, αλλιώς αποτυγχάνει. Το `nonvar/1` συμπεριφέρεται αντίστροφα, δηλαδή είναι η άρνηση του `var/1`. Το `atom/1` επιτυγχάνει όταν το όρισμά του είναι σταθερά (ή μεταβλητή που έχει πάρει τιμή σταθερά) και το `integer/1` συμπεριφέρεται αντίστοιχα για ακέραιους. Ακολουθούν παραδείγματα χρήσης των κατηγορημάτων αυτών:

```
?- var(X).
X = _
?- var(something).
no
?- var(Y), Y = 42.
Y = 42
?- Y = 42, var(Y).
no
?- nonvar(X).
no
?- nonvar(const).
yes
?- atom(<==>), integer(3456).
yes
?- atom(X_23)
no
?-
```

name/2 =../2

Πολλές φορές, χρειάζεται να επέμβουμε στη δομή μιας σταθεράς ή ενός αριθμού στο επίπεδο των χαρακτήρων του. Αυτή η δυνατότητα μας παρέχεται με το ενσωματωμένο κατηγόρημα `name/2`, το οποίο, όταν καλείται ως `name(S, L)`, αντιστοιχεί το σύμβολο `S` (σταθερά ή αριθμό) με τη λίστα `L` των ASCII κωδικών των χαρακτήρων από τους οποίους αποτελείται το `S`. Δείτε τα παραδείγματα:

```
?- name(abcd, L).
L = [97,98,99,100]
?- name('good point', [_,_,_,_|L1]), name(bad, L2),
   append(L2, L1, L3), name(S, L3).
L1 = [32,112,111,105,110,116]
L2 = [98,97,100]
L3 = [98,97,100,32,112,111,105,110,116]
S = 'bad point'
?-
```

Το ενσωματωμένο κατηγόρημα `=../2` χρησιμοποιείται για τη διείσδυση στα συστατικά (συναρτησιακό σύμβολο και ορίσματα) ενός σύνθετου όρου. Είναι ορισμένο και ως ενδοθεματικός `xfx` τελεστής με προτεραιότητα 700. Ο στόχος `T =.. L` αντιστοιχεί τον όρο `T` με τη λίστα `L` που έχει στοιχεία, κατά σειρά, το συναρτησιακό σύμβολο του `T` και τα ορίσματά του. Παραδείγματα χρήσης του είναι τα εξής:

```
?- f(a, b) =.. L.
L = [f,a,b]
?- g(1, 2, 3) =.. [_|L], T =.. [h|L].
L = [1,2,3]
T = h(1, 2, 3)
?-
```

`consult/1` `[.....]` `halt/0`

Δεδομένου ότι η μελέτη σας μέχρι αυτό το σημείο βασίστηκε και σε παράλληλη πρακτική εξάσκηση σε συγκεκριμένο σύστημα Prolog (ή μήπως όχι;), σίγουρα έχετε βρει τρόπο να φορτώνετε τα προγράμματα που γράφετε στο σύστημα. Σε πολλά συστήματα, αυτό γίνεται και μέσω της διεπαφής με το χρήστη που παρέχουν, αλλά, αν δεν συμβαίνει κάτι τέτοιο, υπάρχει το ενσωματωμένο κατηγόρημα `consult/1`, το οποίο, καλώντας το με όρισμα ένα όνομα αρχείου, φορτώνει στο σύστημα το πρόγραμμα Prolog που είναι αποθηκευμένο στο αρχείο αυτό. Εναλλακτικά, μπορείτε, για να φορτώσετε ένα αρχείο προγράμματος, να καλέσετε ένα στόχο που είναι το όνομα του αρχείου κλεισμένο μέσα σε ορθογώνιες αγκύλες.

Τέλος, ένα σύστημα Prolog τερματίζει την αλληλεπίδρασή του με το χρήστη είτε μέσω της προσφερόμενης διεπαφής, σε αρκετά συστήματα, είτε με την εκτέλεση του στόχου `halt`.

Άσκηση 5.14

Θεωρήστε δύο κατηγορήματα `strangey/1` και `strangen/1`, με τις εξής προδιαγραφές:

- Το `strangey(L)` επιστρέφει στο `L` μια λίστα 10 στοιχείων, με την ιδιότητα να προκύπτει η ίδια λίστα, είτε όταν προστεθούν στην αρχή της `L` τα στοιχεία `a`, `b`, `c`, είτε όταν προστεθούν στο τέλος της `L` τα στοιχεία `b`, `c`, `a`, με αυτήν ακριβώς τη σειρά.
- Το `strangen(L)` επιστρέφει στο `L` μια λίστα 10 στοιχείων, με την ιδιότητα να προκύπτει η ίδια λίστα, είτε όταν προστεθούν στην αρχή της `L` τα στοιχεία `a`, `b`, `c`, είτε όταν προστεθούν στο τέλος της `L` τα στοιχεία `b`, `a`, `c`, με αυτήν ακριβώς τη σειρά.

Ακολουθούν ημιτελείς ορισμοί των κατηγορημάτων αυτών. Συμπληρώστε τα κενά:

```
strangey(L) :- L = [                          ] ,
               append([a,b,c], [    ] , L1), [                          ] .

strangen(L) :- [                              ] ,
               [                              ] , append(L, [    ] , [    ] ).
```

Άσκηση 5.15

Επιλέξτε τη σωστή εκδοχή από αυτές που προτείνονται:

Τα ενσωματωμένα κατηγορήματα `assert/1` και `retract/1`:

1. είναι πάρα πολύ χρήσιμα και πρέπει να τα εκμεταλλευόμαστε πλήρως στα προγράμματά μας.

2. έχουν λάθη στην υλοποίησή τους και είναι προτιμότερο να τα αποφεύγουμε.
3. είναι σε μερικές περιπτώσεις εξυπηρετικά, αλλά πρέπει να τα χρησιμοποιούμε όσο σπανιότερα γίνεται, γιατί κάνουν τα προγράμματά μας πολύ δυσανάγνωστα.
4. αναδεικνύουν τη δηλωτικότητα του λογικού προγραμματισμού.

Άσκηση 5.16
Ορίστε σε Prolog το κατηγόρημα `langford/1`, το οποίο να επιστρέφει στο όρισμά του μια 27μελή λίστα με τους αριθμούς 1, 2, . . ., 9 (από 3 φορές ο καθένας), η οποία να είναι τέτοια ώστε, για κάθε K (K = 1, 2, . . ., 9), μεταξύ της πρώτης και της δεύτερης εμφάνισης του K, όπως και μεταξύ της δεύτερης και τρίτης, να υπάρχουν σε κάθε περίπτωση ακριβώς K στο πλήθος άλλοι αριθμοί. Φυσικά, το πρόγραμμα αυτό θα πρέπει μέσω οπισθοδρόμησης να βρίσκει όλες αυτές τις λίστες, οι οποίες, πληροφοριακά, ονομάζονται ακολουθίες Langford.

Άσκηση 5.17
Ορίστε σε Prolog το κατηγόρημα `whirl/2`, το οποίο, όταν καλείται με ορίσματα δύο θετικούς ακεραίους `A` και `B`, δηλαδή `whirl(A, B)`, να εκτυπώνει στην οθόνη μια ορθογώνια διάταξη πλάτους `B` και ύψους `A` των αριθμών από το `1` έως το `A*B`, με τον ελικοειδή τρόπο που φαίνεται στη συνέχεια. Οι στήλες πρέπει να έχουν στοιχηθεί δεξιά και να καταλαμβάνουν $N + 1$ θέσεις, όπου N είναι ο αριθμός των ψηφίων στη δεκαδική αναπαράσταση του `A*B`:

```
?- whirl(12, 15).
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  50  51  52  53  54  55  56  57  58  59  60  61  62  63  16
  49  92  93  94  95  96  97  98  99 100 101 102 103  64  17
  48  91 126 127 128 129 130 131 132 133 134 135 104  65  18
  47  90 125 152 153 154 155 156 157 158 159 136 105  66  19
  46  89 124 151 170 171 172 173 174 175 160 137 106  67  20
  45  88 123 150 169 180 179 178 177 176 161 138 107  68  21
  44  87 122 149 168 167 166 165 164 163 162 139 108  69  22
  43  86 121 148 147 146 145 144 143 142 141 140 109  70  23
  42  85 120 119 118 117 116 115 114 113 112 111 110  71  24
  41  84  83  82  81  80  79  78  77  76  75  74  73  72  25
  40  39  38  37  36  35  34  33  32  31  30  29  28  27  26
yes
?-
```

Απαντήσεις ασκήσεων

Απάντηση άσκησης 5.1
Με βάση τους ορισμούς των τελεστών της εκφώνησης, το πρόγραμμα που δόθηκε είναι ισοδύναμο με το εξής:

```
likes(tarzan, jane).
likes(father_of(john), and(eggs, and(fish, and(meat, nothing_else)))).
likes(father_of(father_of(mary)), or(books, and(tv, cinema))).
likes(X, Y) :- doesnt(hate(X, Y)).
equals_to(++(X), plus(X, 1)).
equals_to(++(++(X)), plus(X, 2)).
equals_to(plus(plus(1, 2), 3), 6).
equals_to(minus(X, Y), Z) :- equals_to(plus(Y, Z), X).
```

Απάντηση άσκησης 5.2
Αν ορίσουμε κατάλληλα το κατηγόρημα `member/2` και το συναρτησιακό σύμβολο `./2` ως τελεστές, μπορούμε να ορίσουμε πότε ένα στοιχείο είναι μέλος μιας λίστας με έναν συντακτικά διαφορετικό τρόπο από αυτόν του Παραδείγματος 4.3, αλλά, επί της ουσίας, ίδιον ακριβώς:

```
:- op(180, xfx, member).
:- op(150, xfy, .).

X member X._ .
X member _.L :- X member L.
```

Απάντηση άσκησης 5.3
Ο ορισμός του `max/3` πρέπει να συμπληρωθεί ως εξής:

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

Απάντηση άσκησης 5.4
Ο ορισμός του `gcd/3` πρέπει να συμπληρωθεί ως εξής:

```
gcd(X, X, X).
gcd(X, Y, D) :- X < Y, Y1 is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- X > Y, gcd(Y, X, D).
```

Απάντηση άσκησης 5.5
Οι ερωτήσεις που δόθηκαν μπορούν να διατυπωθούν σε Prolog ως εξής:

1. «*Υπάρχει κάποιος άνεργος που να έχει γεννηθεί μετά το 1980;*»
   ```
   ?- person(_, X, _, date_born(_, _, Y), unemployed), Y > 1980.
   ```
2. «*Υπάρχει κάποιος άνδρας που να παίρνει σύνταξη μεταξύ 1000 και 1300 ευρώ;*»
   ```
   ?- person(man, X, _, _, retired(P)), P > 1000, P < 1300.
   ```
3. «*Υπάρχει κάποια γυναίκα ανύπαντρη που να έχει γεννηθεί ανήμερα τα Χριστούγεννα και ο μισθός της να είναι τουλάχιστον 2000 ευρώ;*»
   ```
   ?- person(woman, X, single, date_born(25, dec, _), works(_, S)),
      S >= 2000.
   ```

Απάντηση άσκησης 5.6
Τα κατηγορήματα της εκφώνησης θα μπορούσαν να οριστούν ως εξής:

```
prod([], 1).
prod([X|L], P) :- prod(L, P1), P is X*P1.

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

maxlist([M], M).
```

```
maxlist([X,Y|L], M) :- maxlist([Y|L], M1), max(X, M1, M).

n_th(1, [X|_], X).
n_th(N, [_|L], X) :- N > 1, M is N-1, n_th(M, L, X).

first_n(0, _, []).
first_n(N, [X|L1], [X|L2]) :- N > 0, M is N-1, first_n(M, L1, L2).

ordered([_]).
ordered([X,Y|L]) :- X =< Y, ordered([Y|L]).
```

Για το κατηγόρημα `max/3`, μπορούμε να χρησιμοποιήσουμε την απάντηση της Άσκησης 5.3.

Απάντηση άσκησης 5.7
Το `primes/2` θα μπορούσε να οριστεί ως εξής:

```
primes(N, PL) :-
   N >= 2,                    % οι πρώτοι είναι πάντα >= 2
   integers(2, N, IL),        % οι ακέραιοι από 2 έως N στην IL
   sift(IL, PL).              % "κοσκινίστε" την IL

integers(N, N, [N]).
integers(M, N, [M|L]) :-      % τοποθετούνται αναδρομικά στο τρίτο
   M < N,                     % όρισμα οι ακέραιοι από M έως N
   M1 is M+1,
   integers(M1, N, L).

sift([],[]).                  % τέλος "κοσκινίσματος"
sift([P|IL], [P|PL]) :-       % το πρώτο στοιχείο P είναι πρώτος
   filter(P, IL, NIL),        % βγάλτε από την IL τα πολλαπλάσια του P
   sift(NIL, PL).             % "κοσκινίστε" ό,τι απέμεινε

filter(_, [], []).
filter(P, [N|IL], [N|NIL]) :-
   N mod P =\= 0,             % το N δεν είναι πολλαπλάσιο του P
   filter(P, IL, NIL).        % μπαίνει στο αποτέλεσμα
filter(P, [N|IL], NIL) :-
   N mod P =:= 0,             % το N είναι πολλαπλάσιο του P
   filter(P, IL, NIL).        % δεν μπαίνει στο αποτέλεσμα
```

Απάντηση άσκησης 5.8
Οι σωστοί ορισμοί του `intersection/3` και του `union/3` ακολουθούν. Χρειάστηκε να προσθέσουμε ένα `!` στο σώμα της δεύτερης πρότασης για το `intersection/3`, μετά το στόχο `member`, για να αποκλείσουμε την περίπτωση να δοθούν λύσεις έπειτα από οπισθοδρόμηση, όπου κοινά στοιχεία των δεδομένων λιστών να μην έχουν περιληφθεί στο αποτέλεσμα. Ομοίως, μετά το στόχο `member` της δεύτερης πρότασης για το `union/3` τοποθετήσαμε ένα `!`, για μην υπάρχει περίπτωση να δοθούν λύσεις έπειτα από οπισθοδρόμηση, όπου κοινά στοιχεία των δεδομένων να περιληφθούν δύο φορές στο αποτέλεσμα:

```
intersection([], _, []).
intersection([X|L1], L2, [X|L3]) :- member(X, L2), !,
                                    intersection(L1, L2, L3).
intersection([_|L1], L2, L3) :- intersection(L1, L2, L3).
```

```
union([], L, L).
union([X|L1], L2, L3) :- member(X, L2), !,
                         union(L1, L2, L3).
union([X|L1], L2, [X|L3]) :- union(L1, L2, L3).
```

Απάντηση άσκησης 5.9

Ναι, θα μπορούσαμε στο σώμα της δεύτερης πρότασης του `filter/3`, να βάλουμε ένα `!`, μετά το στόχο `N mod P =\= 0`, και να διαγράψουμε από το σώμα της τρίτης πρότασης το στόχο `N mod P =:= 0`. Δηλαδή:

```
filter(_, [], []).
filter(P, [N|IL], [N|NIL]) :-
   N mod P =\= 0, !,          % το N δεν είναι πολλαπλάσιο του P
   filter(P, IL, NIL).        % μπαίνει στο αποτέλεσμα
filter(P, [N|IL], NIL) :-
   filter(P, IL, NIL).        % αλλιώς δεν μπαίνει στο αποτέλεσμα
```

Απάντηση άσκησης 5.10

Τα κατηγορήματα της εκφώνησης θα μπορούσαν να οριστούν ως εξής:

```
delete_one(_, [], []).
delete_one(X, [X|L], L) :- !.
delete_one(X, [Y|L1], [Y|L2]) :- delete_one(X, L1, L2).

delete_all(_, [], []).
delete_all(X, [X|L1], L2) :- !, delete_all(X, L1, L2).
delete_all(X, [Y|L1], [Y|L2]) :- delete_all(X, L1, L2).

substitute(_, [], _, []).
substitute(X, [X|L1], Y, [Y|L2]) :- !, substitute(X, L1, Y, L2).
substitute(X, [Z|L1], Y, [Z|L2]) :- substitute(X, L1, Y, L2).

rem_dupls([], []).
rem_dupls([X|L1], L2) :- member(X, L1), !, rem_dupls(L1, L2).
rem_dupls([X|L1], [X|L2]) :- rem_dupls(L1, L2).

difference([], _, []).
difference([X|L1], L2, L3) :- member(X, L2), !,
                              difference(L1, L2, L3).
difference([X|L1], L2, [X|L3]) :- difference(L1, L2, L3).
```

Απάντηση άσκησης 5.11

Ο συμπληρωμένος ορισμός του `disjoint/2` είναι ο εξής:

```
disjoint([], _).
disjoint([X|L1], L2) :- \+ member(X, L2), disjoint(L1, L2).
```

Απάντηση άσκησης 5.12

Οι ορισμοί του `intersection/3` και του `union/3` μπορούν να γραφούν χωρίς αποκοπές ως εξής:

```
intersection([], _, []).
intersection([X|L1], L2, [X|L3]) :- member(X, L2),
                                    intersection(L1, L2, L3).
intersection([X|L1], L2, L3) :- \+ member(X, L2),
                                  intersection(L1, L2, L3).

union([], L, L).
union([X|L1], L2, L3) :- member(X, L2),
                         union(L1, L2, L3).
union([X|L1], L2, [X|L3]) :- \+ member(X, L2),
                              union(L1, L2, L3).
```

Απάντηση άσκησης 5.13
Το κατηγόρημα `ifthenelse/3` θα μπορούσε να οριστεί ως εξής μέσω του `!`:

```
ifthenelse(Cond, ThenGoal, _) :- call(Cond), !, call(ThenGoal).
ifthenelse(_, _, ElseGoal) :- call(ElseGoal).
```

Με άρνηση, θα το ορίζαμε έτσι:

```
ifthenelse(Cond, ThenGoal, _) :- call(Cond), call(ThenGoal).
ifthenelse(Cond, _, ElseGoal) :- \+ call(Cond), call(ElseGoal).
```

Απάντηση άσκησης 5.14
Τα κενά στους ημιτελείς ορισμούς των `strangey/1` και `strangen/1` θα πρέπει να συμπληρωθούν ως εξής:

```
strangey(L) :- L = [_,_,_,_,_,_,_,_,_,_],
               append([a,b,c], L, L1), append(L, [b,c,a], L1).

strangen(L) :- L = [_,_,_,_,_,_,_,_,_,_],
               append([a,b,c], L, L1), append(L, [b,a,c], L1).
```

Αν έχετε την περιέργεια να δείτε ποιες είναι οι λίστες που έχουν αυτές τις ιδιόρρυθμες ιδιότητες, να τα αποτελέσματα:

```
?- strangey(L).
L = [a,b,c,a,b,c,a,b,c,a]
?- strangen(L).
no
?-
```

Δηλαδή, η δεύτερη λίστα δεν υπάρχει.

Απάντηση άσκησης 5.15
Η σωστή απάντηση είναι η 3. Η απάντηση 1 είναι λάθος, γιατί, όπως εξηγήσαμε αναλυτικά, η συχνή χρήση αυτών των κατηγορημάτων όχι μόνο εθίζει, αλλά και δεν συνεισφέρει καθόλου στις ιδέες του δηλωτικού προγραμματισμού. Η απάντηση 2 δεν προκύπτει από πουθενά και είναι εμφανώς λανθασμένη. Επίσης, η απάντηση 4 είναι στα όρια της διαστροφής. Ακριβώς το αντίθετο συμβαίνει από αυτό που δηλώνει. Αν δώσατε τη σωστή απάντηση, αυτό σημαίνει ότι έχετε συναίσθηση των κινδύνων του `assert/1` και του `retract/1`.

Απάντηση άσκησης 5.16
Το κατηγόρημα `langford/1` μπορεί να υλοποιηθεί με τον εξής εντυπωσιακό τρόπο:

```
langford(S) :-   % Δημιουργήστε πρώτα τη λίστα με τα 27 στοιχεία
   S = [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_],
   sublist([1,_,1,_,1], S),      % Εξασφαλίστε την ιδιότητα για τα 1
   sublist([2,_,_,2,_,_,2], S),                    % και για τα 2
   sublist([3,_,_,_,3,_,_,_,3], S),                % και για τα 3
   sublist([4,_,_,_,_,4,_,_,_,_,4], S),            % και για τα 4
   sublist([5,_,_,_,_,_,5,_,_,_,_,_,5], S),        %     -"-    5
   sublist([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6], S),    %     -"-    6
   sublist([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7], S),   %    κ.ο.κ.
   sublist([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8], S),
   sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], S).

sublist(Y, XYZ) :-               % Η γνωστή sublist/2
   append(_, YZ, XYZ),
   append(Y, _, YZ).
```

Δείτε τις απαντήσεις:

```
?- langford(S).
S = [1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7]      ;
S = [1,8,1,9,1,5,2,6,7,2,8,5,2,9,6,4,7,5,3,8,4,6,3,9,7,4,3]      ;
S = [1,9,1,6,1,8,2,5,7,2,6,9,2,5,8,4,7,6,3,5,4,9,3,8,7,4,3]      ;
S = [3,4,7,8,3,9,4,5,3,6,7,4,8,5,2,9,6,2,7,5,2,8,1,6,1,9,1]      ;
S = [3,4,7,9,3,6,4,8,3,5,7,4,6,9,2,5,8,2,7,6,2,5,1,9,1,8,1]      ;
S = [7,5,3,8,6,9,3,5,7,4,3,6,8,5,4,9,7,2,6,4,2,8,1,2,1,9,1]      ;
no
?-
```

Το `langford/1` γίνεται πιο αποδοτικό αν βάλουμε τους στόχους `sublist` στο σώμα του κανόνα, έτσι ώστε να προηγείται αυτός για το 9, μετά αυτός για το 8 κ.ο.κ. Γιατί;

Απάντηση άσκησης 5.17
Το κατηγόρημα `whirl/2` θα μπορούσε να υλοποιηθεί ως εξής:

```
whirl(A, B) :-
   N is A*B, whirl_pos(N, A, B, Whirl),
   name(N, NCodes), length(NCodes, MaxDigs),
   gen_lines(1, A, Lines), write_lines(MaxDigs, Lines, Whirl).

% Δημιουργήστε το στρόβιλο Whirl ως μια λίστα από ζευγάρια
% συντεταγμένων και αριθμών, δηλαδή [p(c(1, 1), 1), p(c(2, 1), 2),
% ...., p(c(X, Y), A*B)]. Η αρχική θέση είναι η c(1, 1), η αρχική
% κατεύθυνση είναι προς τα δεξιά (r), το αριστερό περιθώριο είναι στη
% στήλη 1, το δεξιό περιθώριο είναι στη στήλη B, το επάνω περιθώριο
% είναι στη γραμμή 1 και το κάτω περιθώριο είναι στη γραμμή A.

whirl_pos(N, A, B, [p(c(1, 1), 1)|Whirl]) :-
   create_whirl(1, N, c(1, 1), r, 1, B, 1, A, Whirl).

create_whirl(N, N, _, _, _, _, _, _, []).
create_whirl(I, N, Pos, Dir, LBord, RBord, UBord, DBord,
      [p(NewPos, I1)|Whirl]) :-
   I < N,
   new_pos(Pos, Dir, LBord, RBord, UBord, DBord, NewPos, NewDir,
      NewLBord, NewRBord, NewUBord, NewDBord),
   I1 is I+1,
   create_whirl(I1, N, NewPos, NewDir, NewLBord, NewRBord, NewUBord,
```

```
      NewDBord, Whirl).

new_pos(c(CX, CY), r, LBord, RBord, UBord, DBord, c(NCX, NCY), NewDir,
      LBord, RBord, NewUBord, DBord) :-
   (CX+1 =< RBord, NCX is CX+1, NCY is CY,
    NewDir = r,  % Συνεχίστε να κινείστε προς τα δεξιά.
    NewUBord is UBord) ;
   (CX+1 > RBord, NCX is CX, NCY is CY+1,
    NCY =< DBord,
    NewDir = d,  % Βρήκατε δεξιό περιθώριο, κινηθείτε προς τα κάτω.
    NewUBord is UBord+1).
new_pos(c(CX, CY), d, LBord, RBord, UBord, DBord, c(NCX, NCY), NewDir,
      LBord, NewRBord, UBord, DBord) :-
   (CY+1 =< DBord, NCX is CX, NCY is CY+1,
    NewDir = d,  % Συνεχίστε να κινείστε προς τα κάτω.
    NewRBord is RBord) ;
   (CY+1 > DBord, NCX is CX-1, NCY is CY,
    NCX >= LBord,
    NewDir = l,  % Βρήκατε κάτω περιθώριο, κινηθείτε προς τα αριστερά.
    NewRBord is RBord-1).
new_pos(c(CX, CY), l, LBord, RBord, UBord, DBord, c(NCX, NCY), NewDir,
      LBord, RBord, UBord, NewDBord) :-
   (CX-1 >= LBord, NCX is CX-1, NCY is CY,
    NewDir = l,  % Συνεχίστε να κινείστε προς τα αριστερά.
    NewDBord is DBord) ;
   (CX-1 < LBord, NCX is CX, NCY is CY-1,
    NCY >= UBord,
    NewDir = u,  % Βρήκατε αριστερό περιθώριο, κινηθείτε προς τα επάνω.
    NewDBord is DBord-1).
new_pos(c(CX, CY), u, LBord, RBord, UBord, DBord, c(NCX, NCY), NewDir,
      NewLBord, RBord, UBord, DBord) :-
   (CY-1 >= UBord, NCX is CX, NCY is CY-1,
    NewDir = u,  % Συνεχίστε να κινείστε προς τα επάνω.
    NewLBord is LBord) ;
   (CY-1 < UBord, NCX is CX+1, NCY is CY,
    NCX =< RBord,
    NewDir = r,  % Βρήκατε επάνω περιθώριο, κινηθείτε προς τα δεξιά.
    NewLBord is LBord+1).

gen_lines(A, A, [A]).
gen_lines(L, A, [L|Lines]) :-
   L < A, L1 is L+1,
   gen_lines(L1, A, Lines).

write_lines(MaxDigs, Lines, Whirl1) :-
   member(L, Lines),             % Διαλέξτε μια γραμμή L.
   findall(p(c(X, L), I), member(p(c(X, L), I), Whirl1), Whirl2),
   sort_whirl(Whirl2, Whirl3),   % Ταξινομήστε τα στοιχεία της γραμμής.
   write_line(MaxDigs, Whirl3),  % Εκτυπώστε την ταξινομημένη γραμμή.
   nl, fail.
write_lines(_, _, _).

sort_whirl([], []).              % Ταξινόμηση με εισαγωγή.
sort_whirl([Element|Whirl1], Whirl2) :-
   sort_whirl(Whirl1, Whirl3),
   insert(Element, Whirl3, Whirl2).

insert(p(c(X1, Y), I1), [p(c(X2, Y), I2)|Whirl1],
```

```
      [p(c(X2, Y), I2)|Whirl2]) :-
   X1 > X2, !, insert(p(c(X1, Y), I1), Whirl1, Whirl2).
insert(Element, Whirl, [Element|Whirl]).

write_line(MaxDigs, Whirl) :-
   member(p(_, I), Whirl),       % Διαλέξτε ένα στοιχείο στη γραμμή.
   write_index(MaxDigs, I),      % Εκτυπώστε το στοιχείο.
   fail.
write_line(_, _).

write_index(MaxDigs, Index) :-
   name(Index, Codes), length(Codes, NDigs),
   Spaces is MaxDigs-NDigs+1, spaces(Spaces), write(Index).

spaces(0).
spaces(K) :-
   K > 0, write(' '), K1 is K-1, spaces(K1).
```

Προβλήματα

Πρόβλημα 5.1

Έστω ότι, για τους τελεστές `op1`, `op2`, `op3`, `op4` και `op5`, έχουν δηλωθεί τα εξής:

```
:- op(120, fy,  op1).
:- op(150, yf,  op2).
:- op(250, xfx, op3).
:- op(200, fx,  op4).
:- op(100, xfy, op5).
```

Δώστε τον όρο Prolog, στην κλασική γραφή, με τον οποίο αντιστοιχεί η έκφραση:

```
op1 x op2 op3 op4 y op5 z op5 w
```

Πρόβλημα 5.2

Ορίστε σε Prolog το κατηγόρημα `rem_ind/3`, έτσι ώστε η ερώτηση `rem_ind(Is, Xs, Rs)` να επιστρέφει στη λίστα `Rs` τα στοιχεία που απομένουν όταν από τη λίστα `Xs` διαγραφούν όσα βρίσκονται στις θέσεις οι οποίες καθορίζονται από τους δείκτες που περιέχονται στη λίστα `Is`. Ένα παράδειγμα:

```
?- rem_ind([1,3,4,6],[a,b,c,d,e,f,g],L).
L = [b,e,g]
```

Πρόβλημα 5.3

Υλοποιήστε σε Prolog ένα κατηγόρημα `drop(List1, N, List2)`, το οποίο, για δεδομένη λίστα `List1` και θετικό ακέραιο `N`, να επιστρέφει στη μεταβλητή `List2` τη λίστα που προκύπτει αν διαγραφεί από τη `List1` κάθε `N`-οστό στοιχείο της. Για παράδειγμα, αν το `N` ισούται με `3`, από τη λίστα `List1` θα πρέπει να διαγραφούν το 3ο, το 6ο, το 9ο κτλ. στοιχεία της, για να προκύψει η `List2`. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- drop([a,b,c,d,e,f,g,h,i,j,k,l,m,n],4,L).
L = [a,b,c,e,f,g,i,j,k,m,n]
?- drop([a,b,c,d,e,f,g,h,i,j,k,l,m,n],3,L).
L = [a,b,d,e,g,h,j,k,m,n]
?- drop([1,2,3,4,5,6,7],1,L).
L = []
```

Πρόβλημα 5.4

Ορίστε σε Prolog ένα κατηγόρημα `binnumbs/2`, με τις εξής προδιαγραφές: όταν καλείται ως `binnumbs(N, L)`, με δεδομένο θετικό ακέραιο `N`, να επιστρέφει στη μεταβλητή `L` έναν `N`-ψήφιο δυαδικό αριθμό, σε μορφή λίστας των δυαδικών ψηφίων του. Για την ακρίβεια, το κατηγόρημα αυτό θα πρέπει να επιστρέφει μέσω οπισθοδρόμησης όλους τους `N`-ψήφιους δυαδικούς αριθμούς, και μάλιστα σε αύξουσα σειρά μεγέθους. Ένα παράδειγμα εκτέλεσης είναι το εξής:

```
?- binnumbs(4, L).
L = [0,0,0,0]   ;
L = [0,0,0,1]   ;
L = [0,0,1,0]   ;
L = [0,0,1,1]   ;
L = [0,1,0,0]   ;
.................
L = [1,1,0,1]   ;
L = [1,1,1,0]   ;
L = [1,1,1,1]
```

Πρόβλημα 5.5

Ορίστε σε Prolog το κατηγόρημα `subst_one/4`, με τέτοιον τρόπο ώστε, όταν αυτό καλείται ως `subst_one(X, L1, Y, L2)`, να επιστρέφει στο `L2` τη λίστα που προκύπτει αν αντικατασταθεί μια εμφάνιση του στοιχείου `X` στη λίστα `L1` με το στοιχείο `Y`. Φυσικά, αν υπάρχουν πολλές εμφανίσεις του στοιχείου `X` στη λίστα `L1`, το `subst_one/4` να επιστρέφει μέσω οπισθοδρόμησης όλες τις πιθανές λύσεις, σε καθεμία από τις οποίες έχει γίνει μια δυνατή αντικατάσταση. Ένα παράδειγμα ερώτησης:

```
?- subst_one(b, [a,b,c,b,b,d,e], w, L).
L = [a,w,c,b,b,d,e]   ;
L = [a,b,c,w,b,d,e]   ;
L = [a,b,c,b,w,d,e]
no
```

Πρόβλημα 5.6

Ορίστε σε Prolog το κατηγόρημα `mult_subst/4`, έτσι ώστε, όταν υποβάλλεται η ερώτηση `mult_subst(Is, Xs, Ls, Ys)`, να αντικαθιστά τα στοιχεία της λίστας `Xs` που βρίσκονται στις θέσεις οι οποίες καθορίζονται από τους δείκτες που περιέχονται στη λίστα `Is`, με τα στοιχεία τα οποία περιέχονται στις λίστες που ανήκουν στη λίστα `Ls` (τα στοιχεία-δείκτες της `Is` να αντιστοιχούν ένα προς ένα με τα στοιχεία-λίστες της `Ls`). Το αποτέλεσμα να επιστρέφεται στη μεταβλητή `Ys`. Ένα παράδειγμα:

```
?- mult_subst([2,4,5,7], [a,b,c,d,e,f,g,h], [[k,l],[m],[],[n,o,p]],R).
R = [a,k,l,c,m,f,n,o,p,h]
```

Σημειώνεται ότι τα στοιχεία της λίστας `Xs`, καθώς και τα στοιχεία των λιστών που ανήκουν στη λίστα `Ls`, μπορεί να είναι αυθαίρετοι όροι Prolog.

Πρόβλημα 5.7

Υλοποιήστε το κατηγόρημα `mm/4` έτσι ώστε, όταν αυτό καλείται ως `mm(L1, L2, CS, CD)`, όπου `L1` και `L2` είναι δύο δεδομένες ισομήκεις λίστες, να επιστρέφει στο `CS` το πλήθος των

κοινών στοιχείων των δύο λιστών που βρίσκονται στην ίδια θέση, και στο `CD` το πλήθος των κοινών στοιχείων των δύο λιστών, αφού εξαιρεθούν τα προηγούμενα, ανεξάρτητα από τη θέση στην οποία βρίσκονται. Για παράδειγμα:

```
?- mm([a,b,c,d,e],[f,b,g,d,c],CS,CD).
   CS = 2
   CD = 1
?- mm([a,b,b,c,d],[e,a,b,b,b],CS,CD).
   CS = 1
   CD = 2
```

Πρόβλημα 5.8

Υλοποιήστε σε Prolog ένα κατηγόρημα `triangle/1`, το οποίο, όταν παίρνει ως όρισμα μια λίστα από χαρακτήρες, να εκτυπώνει ένα σύνολο ισόπλευρων τριγώνων, το καθένα εκ των οποίων να έχει πλευρές κατασκευασμένες από ένα χαρακτήρα και να περικλείει τα τρίγωνα που κατασκευάζονται με τους χαρακτήρες οι οποίοι ακολουθούν τον δικό του χαρακτήρα στη λίστα εισόδου. Ο τελευταίος χαρακτήρας της λίστας συνθέτει το μικρότερο κεντρικό τρίγωνο, που δεν είναι άλλο από ένα εκφυλισμένο σημείο. Ένα παράδειγμα εκτέλεσης του προγράμματος είναι το εξής:

```
?- triangle([a,b,c,d]).
         a
        aba
       abcba
      abcdcba
     abccccba
    abbbbbbbba
   aaaaaaaaaaaaa
```

Πρόβλημα 5.9

Πολλές φορές, στην καθημερινή μας ζωή λέμε «*δύο αρνήσεις ισούνται με μία κατάφαση*». Ισχύει το ίδιο και στην Prolog; Δηλαδή, είναι απολύτως ισοδύναμα να καλέσουμε το στόχο *Goal* ή το στόχο `\+ \+` *Goal*; Εξηγήστε την άποψή σας, μέσω των απαντήσεων που θα πάρετε στην Prolog, αν υποβάλετε τις παρακάτω ερωτήσεις:

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

Πρόβλημα 5.10

Υλοποιήστε σε Prolog το κατηγόρημα `givensum/3`, έτσι ώστε το `givensum(L1, S, L2)` να επιστρέφει στο `L2` όλες τις δυνατές λίστες, μέσω οπισθοδρόμησης, με στοιχεία από τη λίστα `L1` τα οποία έχουν άθροισμα `S`. Για παράδειγμα:

```
?- givensum([5,2,-1,2,-3], 4, L).
L = [5,2,-3]     ;
L = [5,-1]       ;
L = [5,2,-3]     ;
L = [2,2]
```

Αν γνωρίζατε ότι τα στοιχεία της λίστας `L1` είναι πάντα θετικά, θα μπορούσατε να βελτιώσετε την υλοποίησή σας ή δεν σας βοηθά αυτή η επιπλέον γνώση;

Πρόβλημα 5.11

Ορίστε σε Prolog ένα κατηγόρημα `reduce/2`, το οποίο να δέχεται στο πρώτο όρισμά του μια αλγεβρική έκφραση που είναι γινόμενο θετικών ακέραιων αριθμών και μεταβλητών που πιθανώς έχουν υψωθεί και σε κάποια θετική ακέραια δύναμη, οι οποίες παριστάνονται ως σταθερές (άτομα) Prolog. Το κατηγόρημα αυτό να απλοποιεί την έκφραση που του δόθηκε, κάνοντας όλες τις απαραίτητες αναγωγές, επιστρέφοντας το αποτέλεσμα στο δεύτερο όρισμα. Για παράδειγμα:

```
?- reduce(3 * x * y ^ 2 * z * 5 * y ^ 3 * x ^ 5 * 2, E).
E = 30 * x ^ 6 * y ^ 5 * z
```

Πρόβλημα 5.12

Ορίστε ένα κατηγόρημα `differentiate/2`, το οποίο να δέχεται στο πρώτο όρισμά του μια αλγεβρική έκφραση που να μπορεί να περιλαμβάνει γινόμενα, αθροίσματα και διαφορές μεταξύ πολυωνύμων μιας μεταβλητής (έστω `x`), και να βρίσκει την παράγωγο αυτής της έκφρασης, επιστρέφοντας το αποτέλεσμα στο δεύτερο όρισμα ως το απλούστερο δυνατό πολυώνυμο. Για παράδειγμα:

```
?- differentiate((2*x^2-3*x+2)*(3*x-4)-(4*x^2+18*x-7),P).
P = 18 * x ^ 2 - 42 * x
?- differentiate((2*x-3*x^3)-(3*x-x^2+1)*(-x-1)*2+x,P).
P = -15 * x ^ 2 + 8 * x + 11
```

Πρόβλημα 5.13

Υλοποιήστε σε Prolog ένα κατηγόρημα `compr(List1, List2)`, το οποίο, για δεδομένη λίστα `List1`, να επιστρέφει στη μεταβλητή `List2` τη λίστα που προκύπτει αν αντικατασταθούν στη `List1` όλες οι **συνεχόμενες** εμφανίσεις κάθε στοιχείου από μία μόνο εμφάνισή του. Η σειρά των στοιχείων δεν πρέπει να μεταβληθεί. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- compr([a,a,a,b,c,c,d,d,d,d,e],List).
List = [a,b,c,d,e]
?- compr([f(1),g(2),g(2),g(1),f(1),f(1),g(2)],List).
List = [f(1),g(2),g(1),f(1),g(2)]
```

Η υλοποίησή σας αρκεί να δουλεύει σωστά όταν τα στοιχεία της `List1` δεν περιέχουν μεταβλητές. Παρ' όλα αυτά, ποιες απαντήσεις θα παίρνατε αν στο πρόγραμμα που γράψατε υποβάλατε τις παρακάτω ερωτήσεις;

```
?- compr([f(1),f(X),f(2),g(X),g(1)],List).
..................
?- compr([f(X),f(Y),g(X),g(1),g(2),g(Y)],List).
..................
```

Πρόβλημα 5.14

Θεωρήστε ότι ένας πίνακας μπορεί να αναπαρασταθεί στην Prolog ως μια λίστα από τις γραμμές του, καθεμία εκ των οποίων είναι μια λίστα από τα στοιχεία της. Με δεδομένο αυτό, ορίστε το κατηγόρημα `matr_det/2`, έτσι ώστε το `matr_det(M, D)` να επιστρέφει στο `D` την ορίζουσα του πίνακα `M`. Για παράδειγμα:

```
?- matr_det([[5,3],
             [8,7]],D).
   D = 11
?- matr_det([[4,3,2,1],
             [3,2,1,2],
             [1,4,1,3],
             [2,1,3,2]],D).
   D = 51
?- matr_det([[2,0,0,0,0,0],
             [0,3,0,0,0,0],
             [0,0,4,0,0,0],
             [0,0,0,5,0,0],
             [0,0,0,0,6,0],
             [0,0,0,0,0,7]],D).
   D = 5040
```

Πρόβλημα 5.15

Έστω ότι έχετε στη διάθεσή σας ένα κατηγόρημα `randint/2`, το οποίο, όταν καλείται ως `randint(N, R)`, με το `N` να είναι θετικός ακέραιος, επιστρέφει στη μεταβλητή `R` έναν τυχαίο ακέραιο στο διάστημα `[1,N]`. Με τη βοήθεια αυτού του κατηγορήματος, υλοποιήστε ένα κατηγόρημα `randperm/2`, το οποίο, όταν καλείται ως `randperm(L1, L2)`, με δεδομένη λίστα `L1`, να επιστρέφει στη μεταβλητή `L2` μια λίστα που να είναι μια τυχαία μετάθεση των στοιχείων της `L1`. Για παράδειγμα:

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

Επίσης, με τη βοήθεια του `randperm/2`, υλοποιήστε και το `randpermcont/2`, το οποίο να συμπεριφέρεται όπως το πρώτο, αλλά να επιστρέφει συνεχώς μέσω οπισθοδρόμησης τυχαίες μεταθέσεις της δεδομένης λίστας. Δεν απαιτείται να υπάρχουν επαναλαμβανόμενες απαντήσεις, ούτε είναι απαραίτητο το πλήθος των απαντήσεων να είναι πεπερασμένο. Για παράδειγμα:

```
?- randpermcont([1,2,3,4], L).
L = [3,2,4,1]    ;
L = [2,3,4,1]    ;
L = [3,1,4,2]    ;
L = [1,4,2,3]    ;
L = [2,3,4,1]    ;
.................
```

Πρόβλημα 5.16

Υλοποιήστε σε Prolog το κατηγόρημα `maxrep/2`, με τις εξής προδιαγραφές: όταν αυτό καλείται ως `maxrep(List, Maxrep)`, να βρίσκει τη μέγιστη υπολίστα ίδιων συνεχόμενων στοιχείων στη λίστα `List` και να επιστρέφει το αποτέλεσμα στο `Maxrep`, ως έναν όρο της

μορφής X/N, που σημαίνει ότι το στοιχείο X επαναλαμβάνεται N φορές. Αν υπάρχουν περισσότερες της μιας υπολίστες μέγιστου μήκους με επαναλήψεις ίδιου στοιχείου, να επιστρέφονται όλες οι δυνατές λύσεις μέσω οπισθοδρόμησης. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- maxrep([a,a,b,c,c,c,c,d,d,d,e,e,e,e], Maxrep).
Maxrep = c/4      ;
Maxrep = e/4
?- maxrep([a,b,c], Maxrep).
Maxrep = a/1      ;
Maxrep = b/1      ;
Maxrep = c/1
?- maxrep([x,x,x,x,x,x,x,x,x,x,x,x,x,x], Maxrep).
Maxrep = x/14
```

Πρόβλημα 5.17
Υλοποιήστε σε Prolog το κατηγόρημα subseqs/3, με τις εξής προδιαγραφές: όταν αυτό καλείται ως subseqs(N, List, Subseqs), να επιστρέφει στο Subseqs τη λίστα που περιλαμβάνει όλες τις υπολίστες συνεχόμενων N στοιχείων της λίστας List. Ένα παράδειγμα εκτέλεσης είναι το εξής:

```
?- subseqs(3, [a,b,c,d,e,f,g], Subseqs).
Subseqs = [[a,b,c],[b,c,d],[c,d,e],[d,e,f],[e,f,g]]
```

Πώς θα (πρέπει να) συμπεριφερθεί το κατηγόρημα που ορίσατε όταν κληθεί με τιμή του N μεγαλύτερη από το πλήθος των στοιχείων της λίστας List; Αν κληθεί με N ίσο με 0;

Πρόβλημα 5.18
Υλοποιήστε σε Prolog ένα κατηγόρημα maxcsl(List1, List2, Maxcsl), το οποίο, για δεδομένες λίστες List1 και List2, να επιστρέφει στη μεταβλητή Maxcsl τη μέγιστη (σε μήκος) κοινή υπολίστα (συνεχόμενων στοιχείων) στις List1 και List2. Αν δεν υπάρχει κοινή υπολίστα, το κατηγόρημα να αποτυγχάνει, ενώ αν υπάρχουν περισσότερες της μιας κοινές υπολίστες μέγιστου μήκους, να επιστρέφονται όλες μέσω οπισθοδρόμησης. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- maxcsl([a,b,c,d,e,f,g],[h,f,g,i,b,c,d,j,d,e],Maxcsl).
Maxcsl = [b,c,d]
?- maxcsl([a,b,c,d,e,f,g],[h,f,g,i,b,c,d,j,d,e,f],Maxcsl).
Maxcsl = [b,c,d]   ;
Maxcsl = [d,e,f]
?- maxcsl([a,b,c,d],[e,f],Maxcsl).
no
?- maxcsl([b,c,b,c,b,c],[c,b,c,b,c,b], Maxcsl).
Maxcsl = [b,c,b,c,b]   ;
Maxcsl = [c,b,c,b,c]
```

Πρόβλημα 5.19
Στο Παράδειγμα 5.3 είδαμε την υλοποίηση του κατηγορήματος length/2:

```
length([], 0).
length([X|L], N) :- length(L, M), N is M+1.
```

Όπως γνωρίζετε, το κατηγόρημα αυτό μπορεί να χρησιμοποιηθεί για να υπολογισθεί το μήκος δεδομένης λίστας. Για παράδειγμα:

```
?- length([a, b, [c, d], e], N).
N = 4
```

Θα μας ενδιέφερε όμως να μπορούσε να χρησιμοποιηθεί και αντίστροφα, δηλαδή να κατασκευάζει μια λίστα από μεταβλητές δεδομένου μήκους. Για παράδειγμα:

```
?- length(L, 4).
L = [_194, _197, _200, _203]
```

Εξηγήστε γιατί η προηγούμενη υλοποίηση δεν είναι εντελώς σωστή, αν θέλουμε να χρησιμοποιήσουμε το `length/2` με τον δεύτερο τρόπο. Δώστε μια υλοποίηση του κατηγορήματος, τέτοια ώστε να μπορεί να χρησιμοποιηθεί τόσο για τον υπολογισμό του μήκους δεδομένης λίστας, όσο και για την κατασκευή λίστας από μεταβλητές δεδομένου μήκους.

Πρόβλημα 5.20

Θεωρήστε την αναπαράσταση των θετικών ακέραιων ως μια λίστα των ψηφίων τους. Για παράδειγμα, ο αριθμός `287` παριστάνεται με τη λίστα `[2,8,7]`. Ορίστε σε Prolog ένα κατηγόρημα `sum_numbs/3`, το οποίο, όταν καλείται ως `sum_numbs(L1, L2, L3)`, με δεδομένες τις λίστες `L1` και `L2`, που αναπαριστούν δύο θετικούς ακέραιους, σύμφωνα με το προηγούμενο, να επιστρέφει στη μεταβλητή `L3` τη λίστα που αναπαριστά το άθροισμα των δύο αυτών ακέραιων. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- sum_numbs([3,4,1], [5,7,8], L).
L = [9,1,9]
?- sum_numbs([5,3,8,0,7], [6,4,9], L).
L = [5,4,4,5,6]
?- sum_numbs([8,7], [9,9,9,9,9,9,9,9,2,8], L).
L = [1,0,0,0,0,0,0,0,0,1,5]
```

Πρόβλημα 5.21

Έστω ότι έχετε στη διάθεσή σας ένα «ελλιπές» σύστημα Prolog, στο οποίο δεν υπάρχει το γνωστό σας ενσωματωμένο κατηγόρημα `is/2` για την αποτίμηση αριθμητικών εκφράσεων. Αντί αυτού, υπάρχει μια απλή εκδοχή του, το `smplis/2`, το οποίο είναι σε θέση να κάνει τις τέσσερις στοιχειώδεις πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμό και διαίρεση), αλλά μόνο μεταξύ δύο αριθμών. Δηλαδή, δεν είναι σε θέση να υπολογίσει την τιμή πιο πολύπλοκων αριθμητικών εκφράσεων. Όταν του δοθούν τέτοιες εκφράσεις, απλώς αποτυγχάνει. Κάποια παραδείγματα χρήσης του `smplis/2` είναι τα εξής:

```
?- X smplis 5 + 4.2.
X = 9.2
?- X smplis 14 - 3 * 4.
no
```

Ορίστε σε Prolog ένα κατηγόρημα `newis/2`, το οποίο, με τη βοήθεια του `smplis/2`, να μπορεί να υπολογίζει την τιμή και αριθμητικών εκφράσεων που εμπλέκουν τις τέσσερις βασικές πράξεις. Δηλαδή, να συμπεριφέρεται στο θέμα αυτό όπως το γνωστό `is/2`, το οποίο δεν είναι όμως διαθέσιμο. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- X newis 17 - 11.
X = 6
?- X newis 5 + 3 * 2.
X = 11
?- X newis (20 - 4) / 5 + 7 - 2 * 4.
X = 2.2
```

Πρόβλημα 5.22
Ορίστε σε Prolog ένα κατηγόρημα `repeated/3`, με την εξής συμπεριφορά: όταν καλείται ως `repeated(L1, L2, N)`, να επιτυγχάνει, αν η λίστα `L2` είναι το αποτέλεσμα της συνένωσης `N` φορών της λίστας `L1` με τον εαυτό της. Για παράδειγμα:

```
?- repeated([a,b,c,d], [a,b,c,d,a,b,c,d,a,b,c,d], 3).
yes
```

Υλοποιήστε με τέτοιο τρόπο το κατηγόρημα `repeated/3`, ώστε να είναι δυνατόν να κληθεί, όπως φαίνεται και στα ακόλουθα παραδείγματα:

```
?- repeated(L1, [a,b,c,d,a,b,c,d,a,b,c,d], 3).
L1 = [a,b,c,d]
?- repeated([a,b,c,d], [a,b,c,d,a,b,c,d,a,b,c,d], N).
N = 3
?- repeated(L1, [a,b,c,d,a,b,c,d,a,b,c,d], N).
L1 = [a,b,c,d]
N = 3 -> ;
L1 = [a,b,c,d,a,b,c,d,a,b,c,d]
N = 1
?- repeated([a,b,c,d], L2, 3).
L2 = [a,b,c,d,a,b,c,d,a,b,c,d]
```

Πρόβλημα 5.23
Μελετήστε προσεκτικά τον ορισμό του κατηγορήματος `binrep/3` στη συνέχεια.

```
binrep(0, List, List).
binrep(N, List, RepList) :-
   N > 0,
   append(List, List, NewList),
   N1 is N-1,
   binrep(N1, NewList, RepList).
```

Δεδομένου αυτού του ορισμού, ποια θα είναι η απάντηση ενός συστήματος Prolog στην παρακάτω ερώτηση;

```
?- binrep(4, [a,b,c], List).
```

Περιγράψτε, με μία πρόταση, τη λειτουργικότητα αυτού του ορισμού. Τροποποιήστε κατάλληλα τον ορισμό του `binrep/3`, έτσι ώστε να υλοποιεί, εκτός από τη λειτουργικότητα που αναφέρατε, και την εξής:

```
?- binrep(N, List, [1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2]).
N = 0
List = [1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2]    ;
N = 1
List = [1,2,1,2,1,2,1,2]                    ;
N = 2
List = [1,2,1,2]                            ;
N = 3
List = [1,2]
```

Πρόβλημα 5.24

Υλοποιήστε σε Prolog το κατηγόρημα `substodd/4`, με τις εξής προδιαγραφές: όταν αυτό καλείται ως `substodd(X, L1, Y, L2)`, να επιστρέφει στο `L2` τη λίστα που προκύπτει αν αντικατασταθούν όλες οι εμφανίσεις του στοιχείου `X` σε περιττής τάξης θέσεις (1η, 3η, 5η, . . .) στη λίστα `L1` με το στοιχείο `Y`. Για παράδειγμα:

```
?- substodd(c, [a,b,c,c,d,c,c,e,c,f], z, L).
L = [a,b,z,c,d,c,z,e,z,f]
?- substodd(4, [4,4,5,2,4], 0, L).
L = [0,4,5,2,0]
?- substodd(f, [a,b,c,d,e], g, L).
L = [a,b,c,d,e]
```

Πρόβλημα 5.25

Υλοποιήστε σε Prolog ένα κατηγόρημα `diags(Matrix, DiagsDown, DiagsUp)`, το οποίο να παίρνει ως όρισμα έναν πίνακα `Matrix`, στη μορφή μιας λίστας λιστών (οι εσωτερικές λίστες είναι οι γραμμές του πίνακα), και να επιστρέφει στα `DiagsDown` και `DiagsUp` τις λίστες των στοιχείων των κατιουσών και των ανιουσών διαγωνίων του πίνακα, αντίστοιχα. Ένα παράδειγμα εκτέλεσης είναι το εξής:

```
?- diags([[a,b,c,d],[e,f,g,h],[i,j,k,l]],DiagsDown,DiagsUp).
DiagsDown = [[i],[e,j],[a,f,k],[b,g,l],[c,h],[d]]
DiagsUp = [[a],[b,e],[c,f,i],[d,g,j],[h,k],[l]]
```

Πρόβλημα 5.26

Ορίστε σε Prolog ένα κατηγόρημα `multi_split/3`, το οποίο, όταν καλείται με πρώτο όρισμα μια λίστα `L` και δεύτερο όρισμα έναν θετικό ακέραιο `N`, να επιστρέφει στο τρίτο όρισμα μια λίστα με `N` ακριβώς στοιχεία, τα οποία να είναι **μη κενές** λίστες, τέτοιες ώστε, αν συνενωθούν, με τη σειρά που εμφανίζονται, να προκύπτει η λίστα `L`. Το κατηγόρημα που θα γράψετε να επιστρέφει μέσω οπισθοδρόμησης όλες τις εναλλακτικές λύσεις του προβλήματος. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- multi_split([a,b,c,d,e], 3, SL).
SL = [[a],[b],[c,d,e]]    ;
SL = [[a],[b,c],[d,e]]    ;
SL = [[a],[b,c,d],[e]]    ;
SL = [[a,b],[c],[d,e]]    ;
SL = [[a,b],[c,d],[e]]    ;
SL = [[a,b,c],[d],[e]]    ;
no
```

```
?- multi_split([a,b,c,d,e,f,g,h], 8, SL).
SL = [[a],[b],[c],[d],[e],[f],[g],[h]]    ;
no
?- multi_split([a,b,c,d,e,f,g,h], 9, SL).
no
```

Πρόβλημα 5.27

Ας δούμε αν μπορούμε να βάλουμε στην Prolog ιδέες από τη Logo. Έχουμε μια χελώνα η οποία μπορεί να κινείται σε ένα δισδιάστατο σύστημα αξόνων. Κάθε χρονική στιγμή, η χελώνα βρίσκεται σε ένα σημείο του επιπέδου, που ορίζεται από τις συντεταγμένες του, και έχει συγκεκριμένο προσανατολισμό, σύμφωνα με τα τέσσερα σημεία του ορίζοντα. Η χελώνα μπορεί να εκτελέσει δύο μόνο εντολές, τη `step`, οπότε μετακινείται κατά μία μονάδα μήκους προς την κατεύθυνση που είναι προσανατολισμένη, και τη `rotate`, οπότε στρέφεται κατά 90^o δεξιά. Ορίστε σε Prolog ένα κατηγόρημα `closed_path/2`, το οποίο να καλείται με πρώτο όρισμα ένα «πρόγραμμα» για εκτέλεση από τη χελώνα, υπό τη μορφή μιας λίστας από τις εντολές που αναγνωρίζει, και να επιτυγχάνει στην περίπτωση που, εκτελώντας το πρόγραμμα, η χελώνα επιστρέφει στο σημείο από το οποίο ξεκίνησε. Σε περίπτωση επιτυχίας, στο δεύτερο όρισμα να επιστρέφεται η απόσταση που διένυσε η χελώνα. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- closed_path([step,step,step,rotate,step,step,rotate,
                step,step,step,rotate,step,step], D).
D = 10
?- closed_path([step,step,rotate,rotate,step], D).
no
```

Πρόβλημα 5.28

Ορίστε σε Prolog δύο κατηγορήματα `longestdecsub/2` και `longestincsub/2`, τα οποία να καλούνται ως `longestdecsub(L1, L2)` και `longestincsub(L1, L2)`, αντίστοιχα, δίνοντας στη μεταβλητή `L1` μια λίστα από αριθμούς. Το `longestdecsub/2` να επιστρέφει στη μεταβλητή `L2`, μέσω οπισθοδρόμησης, όλες τις λίστες μέγιστου μήκους με στοιχεία από τη λίστα `L1`, οι οποίες είναι γνησίως φθίνουσες, και τα στοιχεία τους εμφανίζονται με τη σειρά που εμφανίζονται και στη λίστα `L1`, όχι όμως κατ' ανάγκη συνεχόμενα. Το `longestincsub/2` διαφέρει από το `longestdecsub/2`, μόνο επειδή η λίστα που επιστρέφει πρέπει να είναι γνησίως αύξουσα. Παραδείγματα εκτέλεσης είναι τα εξής:[4]

```
?- longestdecsub([3,2,7,4,6,1,5], Sub).
Sub = [3,2,1]   ;
Sub = [7,4,1]   ;
Sub = [7,6,1]   ;
Sub = [7,6,5]   ;
no
?- longestincsub([2,4,6,1,3,5,7], Sub).
Sub = [2,4,6,7]   ;
Sub = [2,4,5,7]   ;
```

[4] Το θεώρημα των Erdös-Szekeres (1935) ορίζει ότι κάθε ακολουθία από $m \cdot n+1$ **διαφορετικούς** αριθμούς περιέχει οπωσδήποτε είτε κάποια γνησίως φθίνουσα ακολουθία από $m+1$ αριθμούς είτε κάποια γνησίως αύξουσα ακολουθία από $n+1$ αριθμούς. Στα παραδείγματα της εκφώνησης, για $m = 2$ και $n = 3$, βρίσκουμε, στο μεν πρώτο, γνησίως φθίνουσες ακολουθίες μήκους $2 + 1 = 3$, στο δε δεύτερο, γνησίως αύξουσες ακολουθίες μήκους $3 + 1 = 4$. Άρα, το θεώρημα επαληθεύεται για τα δεδομένα αυτά.

```
Sub = [2,3,5,7]   ;
Sub = [1,3,5,7]   ;
no
```

Πρόβλημα 5.29

Υπάρχει μια παρέα φίλων, κάποιοι από τους οποίους λένε πάντα ψέμματα, ενώ οι υπόλοιποι λένε πάντα την αλήθεια. Όλοι τους γνωρίζουν ποιος είναι ψεύτης και ποιος όχι. Κάποια ημέρα, για να περάσουν την ώρα τους, παίζουν το εξής παιγνίδι: ο καθένας κάνει μια δήλωση της μορφής «*στην παρέα υπάρχουν τουλάχιστον* `K` *ψεύτες*». Για παράδειγμα, έστω ότι υπάρχουν 5 άτομα και οι δηλώσεις τους είναι οι εξής:

1. Ο 1ος δηλώνει: «*Υπάρχουν τουλάχιστον 3 ψεύτες ανάμεσά μας.*»
2. Ο 2ος δηλώνει: «*Υπάρχουν τουλάχιστον 2 ψεύτες ανάμεσά μας.*»
3. Ο 3ος δηλώνει: «*Υπάρχει τουλάχιστον 1 ψεύτης ανάμεσά μας.*»
4. Ο 4ος δηλώνει: «*Υπάρχουν τουλάχιστον 4 ψεύτες ανάμεσά μας.*»
5. Ο 5ος δηλώνει: «*Υπάρχουν τουλάχιστον 2 ψεύτες ανάμεσά μας.*»

Σχετικά εύκολα μπορούμε να διαπιστώσουμε ότι, με βάση τα παραπάνω δεδομένα, στην παρέα το 1ο και το 4ο άτομο, και μόνο αυτοί, πρέπει να είναι ψεύτες.

Υλοποιήστε σε Prolog ένα κατηγόρημα `liars/2`, το οποίο, όταν καλείται με πρώτο όρισμα τη λίστα των αριθμών που δηλώνονται από κάθε άτομο ως ελάχιστο πλήθος ψευτών στην παρέα, να επιστρέφει στο δεύτερο όρισμα μια λίστα η οποία να δείχνει τι είναι το κάθε άτομο της παρέας, ψεύτης ή όχι, μέσω κατάλληλης τιμής, 1 ή 0. Δύο παραδείγματα εκτέλεσης είναι τα εξής:

```
?- liars([3, 2, 1, 4, 2], Liars).
Liars = [1, 0, 0, 1, 0]
?- liars([4, 9, 1, 12, 14, 8, 1, 17, 3, 6, 5, 6, 18,
          20, 0, 8, 7, 9, 4, 16], Liars).
Liars = [0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
```

Πρόβλημα 5.30

Ορίστε σε Prolog το κατηγόρημα `cycle/3`, το οποίο, όταν καλείται δίνοντας στα δύο πρώτα ορίσματά του δύο θετικούς ακέραιους `M` και `N`, να επιστρέφει στο τρίτο όρισμά του μια λίστα από δεκαδικά ψηφία τα οποία αποτελούν την περίοδο του δεκαδικού αριθμού που προκύπτει από τη διαίρεση `M/N`. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- cycle(3,4,C).
C = [0]      %  3/4 = 0.75000...
?- cycle(4,3,C).
C = [3]      %  4/3 = 1.3333...
?- cycle(1,7,C).
C = [1,4,2,8,5,7]
             %  1/7 = 0.142857142857...
?- cycle(129,55,C).
C = [4,5]    %  129/55 = 2.3454545...
?- cycle(13,43,C).
C = [3,0,2,3,2,5,5,8,1,3,9,5,3,4,8,8,3,7,2,0,9]
             %  13/43 = 0.3023255813953488372093023 25...
```

Πρόβλημα 5.31

Το πρόβλημα της κατάστρωσης σχεδίου στην Τεχνητή Νοημοσύνη συνίσταται στην εύρεση μιας ακολουθίας ή ενός συνόλου από ενέργειες, οι οποίες, αν εκτελεσθούν, οδηγούν στην επίτευξη ενός στόχου. Υπάρχουν μεθοδολογίες γραμμικής κατάστρωσης σχεδίου, των οποίων το αποτέλεσμα είναι μια συγκεκριμένη ακολουθία από ενέργειες, δηλαδή ένα γραμμικό πλάνο, που πρέπει να εκτελεσθούν με τη δεδομένη σειρά, για να επιτευχθεί ο επιθυμητός στόχος. Υπάρχουν όμως και μεθοδολογίες μη γραμμικής κατάστρωσης σχεδίου, των οποίων το αποτέλεσμα είναι ένα σύνολο από ζευγάρια διατεταγμένων ενεργειών, δηλαδή ένα μη γραμμικό πλάνο, στις οποίες η απαίτηση, για την επίτευξη του στόχου, ορίζει όπως η πρώτη ενέργεια κάθε ζευγαριού να εκτελεσθεί πριν από τη δεύτερη ενέργεια. Ένα μη γραμμικό πλάνο δεν ορίζει συγκεκριμένη ακολουθία εκτέλεσης των ενεργειών, απλώς περιγράφει εμμέσως όλες τις ακολουθίες οι οποίες σέβονται τις επιμέρους διατάξεις ενεργειών που περιέχονται στο πλάνο.

Υλοποιήστε σε Prolog ένα κατηγόρημα `linearize/2`, με τις εξής προδιαγραφές: όταν καλείται ως `linearize(NonLinearPlan, LinearPlan)`, με δεδομένο ένα μη γραμμικό πλάνο `NonLinearPlan`, που αναπαρίσταται ως μια λίστα από διατάξεις ενεργειών, όπως είναι όροι της μορφής `bef(Act1, Act2)`, όπου `Act1` και `Act2` είναι ενέργειες, να κατασκευάζει μέσω οπισθοδρόμησης όλα τα δυνατά γραμμικά πλάνα που αντιστοιχούν στο δοθέν μη γραμμικό πλάνο. Υποθέστε ότι δεν υπάρχουν ενέργειες που δεν συνδέονται με τουλάχιστον άλλη μία ενέργεια με συγκεκριμένη διάταξη στο δοθέν μη γραμμικό πλάνο. Παραδείγματα εκτέλεσης του `linearize/2` είναι τα εξής:

```
?- linearize([bef(a,b), bef(b,c), bef(c,d),
              bef(a,e), bef(e,d), bef(d,f)], LinearPlan).
LinearPlan = [a,e,b,c,d,f]    ;
LinearPlan = [a,b,e,c,d,f]    ;
LinearPlan = [a,b,c,e,d,f]    ;
no
?- linearize([bef(a,b), bef(b,c), bef(c,a)], LinearPlan).
no
```

Δεδομένου ότι σε πραγματικά προβλήματα το πλήθος των ενεργειών ενός πλάνου μπορεί να είναι ιδιαίτερα μεγάλο, η υλοποίηση που θα προτείνετε για το `linearize/2` δεν θα πρέπει να έχει εκθετική πολυπλοκότητα ως προς το πλήθος των ενεργειών.

Βιβλιογραφικές αναφορές

[1] I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison Wesley (4th Edition), 2011.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer (5th Edition), 2003.

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

Κεφάλαιο 6
Λογικός προγραμματισμός με περιορισμούς

Σύνοψη

Σε αυτό το κεφάλαιο παρουσιάζονται τα προβλήματα ικανοποίησης περιορισμών και ο τρόπος αντιμετώπισής τους με τη βοήθεια του λογικού προγραμματισμού με περιορισμούς. Επίσης, περιγράφονται οι δυνατότητες της βιβλιοθήκης `ic` *της γλώσσας* ECL^iPS^e*, για την επίλυση προβλημάτων ικανοποίησης περιορισμών σε μεταβλητές πεπερασμένων πεδίων. Τέλος, γίνεται αναφορά στη βιβλιοθήκη* `branch_and_bound` *της* ECL^iPS^e*, για την αντιμετώπιση προβλημάτων ικανοποίησης περιορισμών, που είναι και προβλήματα βελτιστοποίησης.*

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει μελετήσει τα Κεφάλαια 3, 4 και 5.

6.1 Προβλήματα ικανοποίησης περιορισμών

Ο **λογικός προγραμματισμός με περιορισμούς** [1] είναι μια επέκταση του λογικού προγραμματισμού, η οποία παρέχει τη δυνατότητα δηλωτικής και ταυτόχρονα αποδοτικής αντιμετώπισης προβλημάτων, που μπορούν να διατυπωθούν ως **προβλήματα ικανοποίησης περιορισμών**. Ένα πρόβλημα ικανοποίησης περιορισμών μπορεί να μοντελοποιηθεί από ένα σύνολο μεταβλητών και ένα σύνολο περιορισμών μεταξύ των μεταβλητών αυτών. Οι μεταβλητές μπορούν να παίρνουν τιμές από προκαθορισμένα σύνολα δυνατών τιμών. Το ζητούμενο είναι να βρεθούν εκείνοι οι συνδυασμοί τιμών των μεταβλητών που να ικανοποιούν όλους τους περιορισμούς, δηλαδή οι λύσεις του προβλήματος. Πολλές φορές, σε κάθε υποψήφια λύση αντιστοιχεί ένα κόστος, που είναι συνάρτηση των μεταβλητών. Οπότε, αυτό που ενδιαφέρει είναι η εύρεση εκείνης της λύσης που έχει το μικρότερο δυνατό κόστος. Τότε, το πρόβλημα είναι και πρόβλημα βελτιστοποίησης.

Ο φυσιολογικός τρόπος επίλυσης ενός προβλήματος ικανοποίησης περιορισμών είναι ίσως η μέθοδος «**γέννα και δοκίμασε**». Δηλαδή, «*γέννα συστηματικά τους δυνατούς συνδυασμούς τιμών των μεταβλητών και, για καθέναν από αυτούς, έλεγξε αν ισχύουν οι περιορισμοί*». Όμως, η μέθοδος αυτή μπορεί να δώσει αποτελέσματα μόνο όταν το πρόβλημα είναι σχετικά μικρού μεγέθους (λίγες μεταβλητές και λίγες δυνατές τιμές γι' αυτές). Στον λογικό προγραμματισμό με περιορισμούς χρησιμοποιείται η μέθοδος «**περιόρισε και γέννα**». Δηλαδή, «*διατύπωσε τους περιορισμούς του προβλήματος, έτσι ώστε να χρησιμοποιηθούν αυτοί ενεργά, για να αποκόψουν πιθανές τιμές από τις μεταβλητές, και, στη συνέχεια, ξεκίνησε μια συστηματική διαδικασία γέννησης τιμών για τις μεταβλητές, αλλά σε κάθε βήμα να ενεργοποιείς πάλι τους περιορισμούς, προκειμένου να φροντίσουν για άλλες δυνατές αποκοπές*

τιμών».

Παράδειγμα 6.1
Έστω ότι πρέπει να λύσουμε τον εξής γρίφο: Με ποια αριθμητικά ψηφία πρέπει να αντικατασταθούν τα γράμματα της πρόσθεσης που ακολουθεί, ώστε να έχουμε διαφορετικό ψηφίο για κάθε διαφορετικό γράμμα και, φυσικά, η πρόσθεση να είναι σωστή;

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

Αυτό είναι ένα πρόβλημα ικανοποίησης περιορισμών, γιατί μπορεί να μοντελοποιηθεί από το σύνολο των μεταβλητών $\{S, E, N, D, M, O, R, Y\}$, καθεμία εκ των οποίων μπορεί να πάρει τιμή από το σύνολο $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, με τους περιορισμούς:

- Οι τιμές των μεταβλητών να είναι διαφορετικές μεταξύ τους
- Να ισχύει η σχέση:

$$(1000 \times S + 100 \times E + 10 \times N + D) + (1000 \times M + 100 \times O + 10 \times R + E) =$$

$$10000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y$$

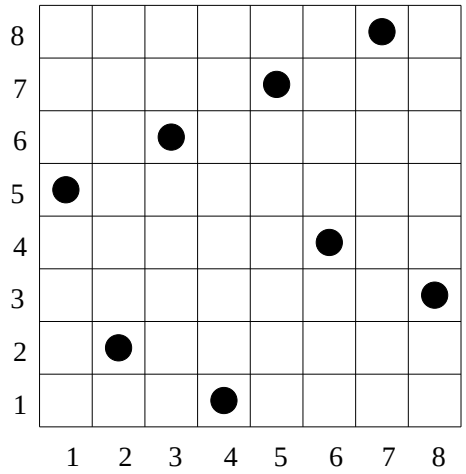
Με λίγη σκέψη, μπορούμε να βρούμε ότι οι σωστές αναθέσεις τιμών στις μεταβλητές είναι $S = 9$, $E = 5$, $N = 6$, $D = 7$, $M = 1$, $O = 0$, $R = 8$ και $Y = 2$, δηλαδή η σωστή πρόσθεση είναι η:

$$\begin{array}{cccccc} & & 9 & 5 & 6 & 7 \\ + & & 1 & 0 & 8 & 5 \\ \hline & 1 & 0 & 6 & 5 & 2 \end{array}$$

Πώς όμως θα μπορούσε να λυθεί το πρόβλημα αυτό μέσω λογικού προγραμματισμού με περιορισμούς; Η απάντηση βρίσκεται στο Παράδειγμα 6.4.

Παράδειγμα 6.2
Ένα κλασικό πρόβλημα ικανοποίησης περιορισμών είναι το πρόβλημα των 8 βασιλισσών. Στο πρόβλημα αυτό, το ζητούμενο είναι να τοποθετηθούν 8 βασίλισσες σε μια σκακιέρα (πλαίσιο 8×8), έτσι ώστε, με βάση τους κανόνες που ισχύουν στο σκάκι για την κίνηση των διαφόρων κομματιών στη σκακιέρα, οι βασίλισσες να μην απειλούνται μεταξύ τους. Σημειώνεται ότι στο σκάκι μια βασίλισσα μπορεί να κινηθεί οριζόντια, κατακόρυφα ή διαγώνια για οποιονδήποτε αριθμό τετραγώνων. Μια λύση του προβλήματος των 8 βασιλισσών φαίνεται στο Σχήμα 6.1.

Πώς θα μπορούσαμε να λύσουμε το πρόβλημα αυτό σε Prolog; Καταρχάς, πρέπει να επιλέξουμε κάποια αναπαράσταση για τις υποψήφιες λύσεις. Μια καλή ιδέα είναι να θεωρήσουμε ότι μια λύση παριστάνεται από μια μετάθεση, σε μορφή λίστας, των ακέραιων αριθμών από το 1 έως το 8. Το i-οστό στοιχείο της λίστας είναι η γραμμή στην οποία πρέπει να τοποθετηθεί η βασίλισσα στη στήλη i. Η αναπαράσταση αυτή θεωρεί δεδομένο ότι σε κάθε στήλη έχουμε ακριβώς μία βασίλισσα (αφού το i-οστό στοιχείο της λίστας έχει μοναδική τιμή) και ότι δεν υπάρχει γραμμή με δύο βασίλισσες, αφού η λίστα περιέχει διαφορετικά στοιχεία. Αν, λοιπόν, γεννήσουμε συστηματικά όλες τις πιθανές μεταθέσεις της



Σχήμα 6.1: Μια λύση του προβλήματος των 8 βασιλισσών.

λίστας $[1, 2, 3, 4, 5, 6, 7, 8]$, εκείνες στις οποίες δεν υπάρχει έστω ένα ζευγάρι βασιλισσών στην ίδια διαγώνιο είναι λύσεις του προβλήματος (μέθοδος «γέννα και δοκίμασε»). Η λύση που φαίνεται στο Σχήμα 6.1 είναι η $[5, 2, 6, 1, 7, 4, 8, 3]$. Το πρόγραμμα Prolog που ακολουθεί βρίσκει τις λύσεις για το γενικευμένο πρόβλημα των N βασιλισσών:

```
gtnqueens(N, Solution) :-
   choices(1, N, Choices),   % Κατασκευή της λίστας [1,2,3,...,N]
   permutation(Choices, Solution),   % Δημιουργία υποψήφιας λύσης
   safe(Solution).           % Έλεγχος ορθότητας λύσης

choices(N, N, [N]).
choices(M, N, [M|Ns]) :-
   M < N,
   M1 is M+1,
   choices(M1, N, Ns).

permutation([], []).         % Δημιουργία μεταθέσεων λίστας
permutation([Head|Tail], PermList) :-
   permutation(Tail, PermTail),
   delete(Head, PermList, PermTail).

safe([]).                    % Έλεγχος για όλες τις βασίλισσες
safe([Queens|Others]) :-
   safe(Others),
   noattack(Queens, Others, 1).

noattack(_, [], _).          % Έλεγχος για μία βασίλισσα
noattack(Y, [Y1|Ylist], Xdist) :-
   Y1-Y =\= Xdist,
   Y-Y1 =\= Xdist,
   NewXdist is Xdist+1,
   noattack(Y, Ylist, NewXdist).

gogt(N, NSols, Time) :-      % Εύρεση πλήθους λύσεων και χρόνου εκτέλεσης
   cputime(T1),
   findall(Sol, gtnqueens(N, Sol), Sols),
   cputime(T2),
   length(Sols, NSols),
   Time is T2-T1.
```

Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- gtnqueens(8, Solution).
Solution = [5, 2, 6, 1, 7, 4, 8, 3]     ;
Solution = [6, 3, 5, 7, 1, 4, 2, 8]     ;
Solution = [6, 4, 7, 1, 3, 5, 2, 8]     ;
.....................................
?- gogt(4, NSols, Time).
NSols = 2
Time = 0.0
?- gogt(5, NSols, Time).
NSols = 10
Time = 0.0
?- gogt(6, NSols, Time).
NSols = 4
Time = 0.0
?- gogt(7, NSols, Time).
NSols = 40
Time = 0.0
?- gogt(8, NSols, Time).
NSols = 92
Time = 0.016000000000000014
?- gogt(9, NSols, Time).
NSols = 352
Time = 0.15600000000000003
?- gogt(10, NSols, Time).
NSols = 724
Time = 1.656
?- gogt(11, NSols, Time).
NSols = 2680
Time = 19.219
?- gogt(12, NSols, Time).
NSols = 14200
Time = 247.141
?-
```

Παράδειγμα 6.3

Μια διαφορετική, και κάπως βελτιωμένη, προσέγγιση για την επίλυση του προβλήματος των N βασιλισσών θα μπορούσε να ήταν η εξής. Αντί να ελέγχεται η νομιμότητα μιας υποψήφιας λύσης, αφού αυτή κατασκευασθεί πλήρως, να ελέγχονται οι απειλές μεταξύ βασιλισσών στη διάρκεια της δημιουργίας της. Έτσι, μια μερικώς κατασκευασμένη υποψήφια λύση, στην οποία υπάρχει ζευγάρι απειλούμενων βασιλισσών, δεν θα επεκταθεί περισσότερο. Πρόκειται για τη **μέθοδο της οπισθοδρόμησης**, η οποία είναι μια παραλλαγή της «γέννα και δοκίμασε». Στη συνέχεια, φαίνεται ένα πρόγραμμα Prolog που αντιμετωπίζει το γενικευμένο πρόβλημα των N βασιλισσών με τη μέθοδο της οπισθοδρόμησης. Μια δευτερεύουσα διαφορά με το πρόγραμμα που είδαμε στο Παράδειγμα 6.2 αφορά την αναπαράσταση της λύσης ως μιας λίστας X/Y συντεταγμένων. Με την αναπαράσταση αυτή, η λύση που φαίνεται στο Σχήμα 6.1 είναι η $[1/5, 2/2, 3/6, 4/1, 5/7, 6/4, 7/8, 8/3]$:

```
btnqueens(N, Solution) :-
   make_tmpl(1, N, Solution), % Δημιουργία προτύπου λύσης
   solution(N, Solution).     % Σταδιακή κατασκευή λύσης

solution(_, []).
```

```
solution(N, [X/Y|Others]) :-
   solution(N, Others),
   between(1, N, Y),      % Τοποθέτηση βασίλισσας στη στήλη X
   noattack(X/Y, Others). % Έλεγχος απειλής με τις ήδη τοποθετημένες

noattack(_, []).
noattack(X/Y, [X1/Y1|Others]) :-
   Y =\= Y1,              % Βασίλισσες όχι στην ίδια γραμμή
   Y1-Y =\= X1-X,         % Βασίλισσες όχι στην ίδια ανιούσα διαγώνιο
   Y1-Y =\= X-X1,         % Βασίλισσες όχι στην ίδια κατιούσα διαγώνιο
   noattack(X/Y, Others).

make_tmpl(N, N, [N/_]).   % Πρότυπο λύσης: Λίστα X/Y συντεταγμένων
make_tmpl(I, N, [I/_|Rest]) :-
   I < N,
   I1 is I+1,
   make_tmpl(I1, N, Rest).

between(I, J, I) :-       % Γέννηση δυνατών ακέραιων στο διάστημα [I,J]
   I =< J.
between(I, J, X) :-
   I < J,
   I1 is I+1,
   between(I1, J, X).

gobt(N, NSols, Time) :-   % Εύρεση πλήθους λύσεων και χρόνου εκτέλεσης
   cputime(T1),
   findall(Sol, btnqueens(N, Sol), Sols),
   cputime(T2),
   length(Sols, NSols),
   Time is T2-T1.
```

Και παραδείγματα εκτέλεσης:

```
?- btnqueens(8, Solution).
Solution = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1]     ;
Solution = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1]     ;
Solution = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1]     ;
.........................................
?- gobt(4, NSols, Time).
NSols = 2
Time = 0.0
?- gobt(5, NSols, Time).
NSols = 10
Time = 0.0
?- gobt(6, NSols, Time).
NSols = 4
Time = 0.0
?- gobt(7, NSols, Time).
NSols = 40
Time = 0.0
?- gobt(8, NSols, Time).
NSols = 92
Time = 0.0
?- gobt(9, NSols, Time).
NSols = 352
Time = 0.016000000000000014
?- gobt(10, NSols, Time).
```

```
NSols = 724
Time = 0.1090000000000001
?- gobt(11, NSols, Time).
NSols = 2680
Time = 0.57799999999999985
?- gobt(12, NSols, Time).
NSols = 14200
Time = 3.5630000000000006
?- gobt(13, NSols, Time).
NSols = 73712
Time = 27.86
?- gobt(14, NSols, Time).
NSols = 365596
Time = 154.04700000000003
?-
```

Παρατηρήστε ότι οι χρόνοι εκτέλεσης με τη μέθοδο της οπισθοδρόμησης είναι 1-2 τάξεις μεγέθους μικρότεροι από αυτούς της μεθόδου «γέννα και δοκίμασε». □

Υπάρχουν διάφορες δυνατότητες υποστήριξης προγραμματισμού με περιορισμούς από γλώσσες λογικού προγραμματισμού, ανάλογα με το είδος των δυνατών τιμών για τις μεταβλητές και το είδος των περιορισμών που καλύπτονται. Μια συνηθισμένη περίπτωση αφορά την κάλυψη μεταβλητών πεπερασμένων πεδίων και διαφόρων ειδών περιορισμών σε αυτές (αριθμητικών, συμβολικών, λογικών κτλ.), όπως γίνεται στη γλώσσα λογικού προγραμματισμού με περιορισμούς **ECL^{i}PS^e** [2]. Η μεθοδολογία επίλυσης ενός προβλήματος ικανοποίησης περιορισμών στην ECL^{i}PS^e, αλλά και σε άλλες παρόμοιες γλώσσες, συνίσταται σε:

1. Ορισμό των μεταβλητών που χρειάζονται για το πρόβλημα, καθώς και των πεδίων τους.
2. Διατύπωση των περιορισμών που μοντελοποιούν το πρόβλημα.
3. Συστηματική γέννηση τιμών για τις μεταβλητές, με στόχο την εύρεση μιας ή όλων των λύσεων.

Στην περίπτωση που το πρόβλημα ικανοποίησης περιορισμών είναι ταυτόχρονα και πρόβλημα βελτιστοποίησης, η προηγούμενη διαδικασία προσαρμόζεται ως εξής:

1. Ορισμός των μεταβλητών που χρειάζονται για το πρόβλημα, καθώς και των πεδίων τους.
2. Διατύπωση των περιορισμών που μοντελοποιούν το πρόβλημα.
3. Ορισμός αντικειμενικής συνάρτησης κόστους λύσης.
4. Εφαρμογή της μεθόδου «διακλάδωσε και φράξε», σε συνδυασμό με τη συστηματική γέννηση τιμών για τις μεταβλητές, με στόχο την εύρεση της βέλτιστης λύσης, δηλαδή αυτής που ελαχιστοποιεί τη συνάρτηση κόστους.

Η ιδέα του προγραμματισμού με περιορισμούς, αν και γεννήθηκε στο περιβάλλον του λογικού προγραμματισμού, έχει ενσωματωθεί πλέον και σε άλλες προγραμματιστικές φιλοσοφίες, όπως αυτήν του αντικειμενοστραφούς προγραμματισμού, για παράδειγμα μέσω του Naxos Solver [3], που είναι μια C++ βιβλιοθήκη για προγραμματισμό με περιορισμούς. Επίσης, υπάρχουν και βιβλιοθήκες Java (Jacop, Cream, Choco), βιβλιοθήκες Python

(python-constraint) ή και αυτόνομα συστήματα ή γλώσσες για προγραμματισμό με περιορισμούς (Picat, MiniZinc, MINION).

Οι περιοχές εφαρμογής του προγραμματισμού με περιορισμούς είναι πάρα πολλές, κυρίως σε προβλήματα συνδυαστικής αναζήτησης, τα οποία έχουν αντιμετωπιστεί στο παρελθόν με μεθόδους επιχειρησιακής έρευνας, όπως η κατασκευή ωρολόγιων προγραμμάτων, ο προγραμματισμός προσωπικού, η σχεδίαση παραγωγής, η διανομή αγαθών κτλ.

Άσκηση 6.1
Γράψτε πρόγραμμα Prolog που να επιλύει το πρόβλημα του γρίφου του Παραδείγματος 6.1 με τη μέθοδο «γέννα και δοκίμασε».

6.2 Η βιβλιοθήκη `ic` της ECL^{i}PS^e

Η ECL^{i}PS^e είναι ένα πολύ ισχυρό σύστημα λογικού προγραμματισμού, το οποίο παρέχει μια σειρά από βιβλιοθήκες ειδικού σκοπού. Μία από αυτές τις βιβλιοθήκες είναι η `ic`, η οποία μπορεί να υποστηρίξει τη διατύπωση και επίλυση προβλημάτων ικανοποίησης περιορισμών σε μεταβλητές πεπερασμένων πεδίων. Στη συνέχεια, παρουσιάζονται τα βασικότερα κατηγορήματα που παρέχονται από τη βιβλιοθήκη:

`#::/2`

Χρησιμοποιείται για τον ορισμό μεταβλητών που μπορούν να πάρουν τιμές από πεπερασμένα πεδία, οι οποίες ονομάζονται **μεταβλητές πεδίου**. Ορίζεται και ως ενδοθεματικός τελεστής. Παραδείγματα χρήσης:

```
X #:: 5..20
[Y,Z,W] #:: [3,6,8,13]
[U,V] #:: [2,5..10,17,23..30]
```

Με τους προηγούμενους στόχους, ορίστηκαν η μεταβλητή `X` με πεδίο τους ακέραιους αριθμούς από το `5` έως το `20`, οι μεταβλητές `Y`, `Z` και `W`, με πεδίο τους ακέραιους `3`, `6`, `8` και `13`, και οι μεταβλητές `U` και `V`, με πεδίο τους ακέραιους `2`, από το `5` έως και το `10`, το `17` και από το `23` έως και το `30`.

`#=/2` `#\=/2` `#>/2` `#</2` `#>=/2` `#=</2`

Οι αριθμητικοί περιορισμοί (ίσο, όχι ίσο, μεγαλύτερο, μικρότερο, μεγαλύτερο ή ίσο, μικρότερο ή ίσο) είναι ορισμένοι και ως ενδοθεματικοί τελεστές. Τα ορίσματά τους μπορεί να είναι αριθμητικές εκφράσεις που εμπλέκουν μεταβλητές πεδίων, αριθμητικές σταθερές και καθιερωμένους αριθμητικούς τελεστές (`+`, `-`, `*`, `/`). Επίσης, μπορούν να χρησιμοποιηθούν συναρτήσεις που εφαρμόζονται σε μεταβλητές πεδίων ή σε λίστες μεταβλητών πεδίων, όπως οι:

- `abs(`E`)`: Η απόλυτη τιμή της έκφρασης E.
- `min(`$E1$`,`$E2$`)`: Το ελάχιστο των εκφράσεων $E1$ και $E2$.
- `max(`$E1$`,`$E2$`)`: Το μέγιστο των εκφράσεων $E1$ και $E2$.
- `min(`$List$`)`: Το ελάχιστο των εκφράσεων στη λίστα $List$.
- `max(`$List$`)`: Το μέγιστο των εκφράσεων στη λίστα $List$.

- `sum(`$List$`)`: Το άθροισμα των εκφράσεων στη λίστα $List$.

Κάποια παραδείγματα χρήσης είναι τα εξής:

```
X+Y #= Z
2*X-3*Y*Y #>= Z*W+max(Z,W)
U #\= abs(V)
```

`alldifferent/1`

Δέχεται ως όρισμα μια λίστα από μεταβλητές πεδίου και επιβάλλει όλες να πάρουν τελικά διαφορετικές τιμές. Παράδειγμα:

```
alldifferent([X,Y,Z,W,U,V])
```

`element/3`

Όταν καλείται ως `element(`I`,`$List$`,`$Value$`)`, με τα I και $Value$ να είναι μεταβλητές πεδίου και το $List$ μια λίστα ακέραιων, επιβάλλει το I-οστό στοιχείο της $List$ να ισούται με $Value$. Παράδειγμα:

```
element(I, [10,20,30,40], Value)
```

`and/2` `or/2` `=>/2` `neg/1`

Τα `and/2` και `or/2` χρησιμοποιούνται, και σε ενδοθεματική σύνταξη, για να εκφράσουν σύζευξη και διάζευξη μεταξύ περιορισμών, αντίστοιχα. Το `=>/2` χρησιμοποιείται, επίσης, ενδοθεματικά, για τη συνεπαγωγή μεταξύ περιορισμών και το `neg/1`, ως προθεματικός τελεστής, για την άρνηση περιορισμού.

Θα πρέπει να σημειωθεί ότι, στη βιβλιοθήκη `ic` της ECL^{i}PS^e, ένας περιορισμός ισοδυναμεί και με μια μεταβλητή πεδίου που μπορεί να πάρει κάποια από τις τιμές `1` ή `0`. Η τιμή `1` αντιστοιχεί στο γεγονός ότι ο περιορισμός αληθεύει, ενώ η τιμή `0` ότι δεν αληθεύει. Δηλαδή, μπορούμε να γράψουμε και τα εξής ως περιορισμούς:

```
B #= (X #> 3 => Y #\= Z)
C #= (X #= Y or Y #= Z) + (neg W #<8)
```

`search/6`

Αποδίδει τιμές σε μεταβλητές πεδίου μέσα από τα πεδία τους. Μέσω οπισθοδρόμησης, επιστρέφει όλους τους επιτρεπτούς συνδυασμούς τιμών. Φυσικά, ο μηχανισμός διαχείρισης των περιορισμών εξασφαλίζει ότι, όταν μια μεταβλητή πάρει κάποια τιμή, τότε από τα πεδία άλλων μεταβλητών διαγράφονται τιμές που δεν είναι συμβατές με αυτήν. Το κατηγόρημα καλείται ως:

`search(`$Vars$`,`Arg`,`$Select$`,`$Choice$`,`$Method$`,`$Options$`)`

Το $Vars$ είναι, στη συνήθη χρήση του κατηγορήματος, η λίστα των μεταβλητών που θα αποτιμηθούν. Στο Arg δίνεται τιμή το `0`, όταν το $Vars$ είναι λίστα μεταβλητών (συνήθης χρήση). Η τιμή του $Select$ καθορίζει τη σειρά αποτίμησης των μεταβλητών. Συνήθεις τιμές είναι οι `input_order` (η σειρά τους στο $Vars$), `first_fail` (προηγούνται οι μεταβλητές με τα μικρότερα πεδία), `smallest` (προηγούνται οι μεταβλητές με τις μικρότερες τιμές στα πεδία τους), `largest` (προηγούνται οι μεταβλητές με τις μεγαλύτερες τιμές στα πεδία τους) και `occurrence` (προηγούνται οι μεταβλητές που εμπλέκονται στους περισσότερους περιορισμούς). Η τιμή του $Choice$ καθορίζει τη σειρά επιλογής τιμών για τη μεταβλητή που θα αποτιμηθεί. Συνήθεις τιμές είναι οι `indomain`, που αντιστοιχεί στη σειρά των τιμών μέσα στο πεδίο, στην πράξη από τη μικρότερη προς τη μεγαλύτερη, η `indomain_middle`, που αντιστοιχεί στην προτεραιότητα τιμών από το μέσον του πεδίου, και η `indomain_split`, με την οποία οι τιμές τελικά επιλέγονται έπειτα από μια διαδικασία υποδιπλασιασμού των πεδίων. Το $Method$ καθορίζει τη μέθοδο αναζήτησης. Συνήθως, το θέτουμε στην τιμή `complete`. Το $Options$ είναι μια λίστα από προχωρημένες επιλογές. Κατά κανόνα, το θέτουμε στην κενή λίστα `([])`. Ένα παράδειγμα κλήσης του `search/6` είναι το εξής:

```
search([X,Y,Z,W], 0, input_order, indomain, complete, [])
```

□

Μπορείτε να βρείτε την πλήρη περιγραφή της βιβλιοθήκης `ic` στο εγχειρίδιο βιβλιοθηκών της γλώσσας [4]. Για την ECL^{i}PS^e, μπορείτε να ανατρέξετε στο εγχειρίδιό της [5], ενώ μια εξαντλητική περιγραφή όλων των ενσωματωμένων κατηγορημάτων της γλώσσας μπορείτε να βρείτε στο εγχειρίδιο αναφοράς [6]. Η πλήρης τεκμηρίωση της ECL^{i}PS^e παρουσιάζεται στο `http://www.eclipseclp.org/doc`.

Παράδειγμα 6.4
Ας δούμε πώς θα μπορούσαμε να αντιμετωπίσουμε το πρόβλημα του Παραδείγματος 6.1, μέσω λογικού προγραμματισμού με περιορισμούς. Ένα πρόγραμμα ECL^{i}PS^e που επιλύει το πρόβλημα μέσω της βιβλιοθήκης `ic` είναι το εξής:

```
:- lib(ic).                          % Φορτώστε τη βιβλιοθήκη ic

cpsendmory(List) :-
   List = [S, E, N, D, M, O, R, Y], % Λίστα μεταβλητών
   List :: 0..9,                    % Πεδίο των μεταβλητών
   alldifferent(List),              % Διαφορετικές τιμές στις μεταβλητές
   1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E #=
      10000*M+1000*O+100*N+10*E+Y,  % Περιορισμός πρόσθεσης
   M #\= 0,                 % Να μην αρχίζουν από 0 τα MORE και MONEY
   S #\= 0,                 % Να μην αρχίζει από 0 το SEND
   search(List, 0, input_order, indomain, complete, []). % Αναζήτηση
```

Και η εκτέλεση:

```
?- cpsendmory(List).
List = [9, 5, 6, 7, 1, 0, 8, 2]   ;
no
?-
```

Παράδειγμα 6.5
Ακολουθεί η υλοποίηση του γενικευμένου προβλήματος των N βασιλισσών σε ECL^{i}PS^e, μέσω της βιβλιοθήκης `ic`:

```
:- lib(ic).                % Φορτώστε τη βιβλιοθήκη ic

cpnqueens(N, Solution) :-
   length(Solution, N),    % Δημιουργία προτύπου λύσης
   Solution #:: 1..N,      % Πεδίο μεταβλητών της λύσης
   constrain(Solution),    % Δήλωση περιορισμών
   generate(Solution).     % Αποτίμηση μεταβλητών

constrain([]).
constrain([X|Xs]) :-
   noattack(X, Xs, 1),     % Μη απειλή βασίλισσας με τις επόμενες
   constrain(Xs).

noattack(_, [], _).
noattack(X, [Y|Ys], M) :-
   X #\= Y,                % Βασίλισσες όχι στην ίδια γραμμή
   X #\= Y-M,              % Βασίλισσες όχι στην ίδια ανιούσα διαγώνιο
   X #\= Y+M,              % Βασίλισσες όχι στην ίδια κατιούσα διαγώνιο
   M1 is M+1,
   noattack(X, Ys, M1).

generate(Sol) :-           % Αναζήτηση όλων των λύσεων
   search(Sol, 0, first_fail, indomain, complete, []).

gocp(N, NSols, Time) :-    % Εύρεση πλήθους λύσεων και χρόνου εκτέλεσης
   cputime(T1),
   findall(Sol, cpnqueens(N, Sol), Sols),
   cputime(T2),
   length(Sols, NSols),
   Time is T2-T1.
```

Ενδεικτικές εκτελέσεις:

```
?- cpnqueens(8, Solution).
Solution = [1, 5, 8, 6, 3, 7, 2, 4]     ;
Solution = [1, 6, 8, 3, 7, 4, 2, 5]     ;
Solution = [1, 7, 4, 6, 8, 2, 5, 3]     ;
.........................................
?- gocp(4, NSols, Time).
NSols = 2
Time = 0.0
?- gocp(5, NSols, Time).
NSols = 10
Time = 0.0
?- gocp(6, NSols, Time).
NSols = 4
Time = 0.0
?- gocp(7, NSols, Time).
NSols = 40
Time = 0.0
?- gocp(8, NSols, Time).
NSols = 92
Time = 0.015000000000000013
?- gocp(9, NSols, Time).
```

```
NSols = 352
Time = 0.032000000000000028
?- gocp(10, NSols, Time).
NSols = 724
Time = 0.14
?- gocp(11, NSols, Time).
NSols = 2680
Time = 0.68800000000000006
?- gocp(12, NSols, Time).
NSols = 14200
Time = 3.4529999999999994
?- gocp(13, NSols, Time).
NSols = 73712
Time = 19.172
?- gocp(14, NSols, Time).
NSols = 365596
Time = 106.578
?-
```

Άσκηση 6.2
Κάποιος πηγαίνει σε ένα κατάστημα ψιλικών για να αγοράσει τέσσερα συγκεκριμένα προϊόντα. Αφού ζητήσει τι ακριβώς θέλει, ο καταστηματάρχης του τα δίνει, του κάνει το λογαριασμό, με τη βοήθεια ενός επιτραπέζιου calculator, και του λέει: «*7,11 ευρώ, παρακαλώ*». Ο πελάτης πληρώνει το ποσό και τη στιγμή που βρίσκεται στην έξοδο ο καταστηματάρχης τού φωνάζει: «*Συγγνώμη, κύριε! Αντί να κάνω πρόσθεση, έκανα, κατά λάθος, πολλαπλασιασμό! Ελάτε πάλι, σας παρακαλώ*». Αφού κάνει τώρα τη σωστή πράξη, διαπιστώνει με έκπληξη ότι το αποτέλεσμα είναι το ίδιο, 7,11 ευρώ. Ποιο μπορεί να είναι το κόστος του κάθε προϊόντος που αγόρασε ο πελάτης;

Γράψτε ένα πρόγραμμα σε ECL^{i}PS^e (μέσω της βιβλιοθήκης `ic`) που να βρίσκει το κόστος του κάθε προϊόντος. Σημειώστε ότι, παρότι στο πρόβλημα εμπλέκονται, προφανώς, αριθμοί που δεν είναι ακέραιοι, εσείς θα πρέπει να το αντιμετωπίσετε μέσω ακέραιων και μόνο.

Άσκηση 6.3
Δίνεται η σχέση:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

στην οποία το ζητούμενο είναι να αντικατασταθούν τα εμφανιζόμενα σύμβολα με τα ψηφία από το 1 έως το 9 (διαφορετικό ψηφίο για κάθε σύμβολο), ώστε να είναι αληθής. Γράψτε ένα πρόγραμμα ECL^{i}PS^e (μέσω της βιβλιοθήκης `ic`) που να το λύνει. Είναι προφανές ότι, για κάθε λύση του προβλήματος, υπάρχουν και άλλες πέντε συμμετρικές με αυτήν, που προκύπτουν κάνοντας αντιμεταθέσεις στα κλάσματα της σχέσης. Μπορείτε να διατυπώσετε έτσι τους περιορισμούς, ώστε να μην προκύπτουν συμμετρικές λύσεις;

6.3 Η βιβλιοθήκη `branch_and_bound` της ECL^{i}PS^e

Στα προβλήματα ικανοποίησης περιορισμών που είναι ταυτόχρονα και προβλήματα βελτιστοποίησης, έχουμε ορίσει μια συνάρτηση των μεταβλητών η οποία ποσοτικοποιεί το κόστος κάθε λύσης. Αυτό που ζητάμε είναι η εύρεση λύσης ελάχιστου κόστους. Αυτό μπορεί να γίνει μέσω μιας διαδικασίας «**διακλάδωσε και φράξε**», στην οποία, αφού βρεθεί η

πρώτη λύση, τίθεται ο περιορισμός το κόστος της επόμενης λύσης να είναι μικρότερο από το κόστος αυτής που βρέθηκε ήδη. Στη συνέχεια, όταν βρεθεί επόμενη καλύτερη λύση, τίθεται ο περιορισμός οι λύσεις να έχουν κόστος μικρότερο από της τελευταίας που βρέθηκε. Η διαδικασία αυτή συνεχίζεται μέχρις ότου να μην μπορεί να βρεθεί καλύτερη λύση από την πιο πρόσφατη. Οπότε, αυτή είναι η βέλτιστη.

Τα προβλήματα αυτά μπορούν να αντιμετωπισθούν από τη συνέργεια της βιβλιοθήκης `ic` της ECL^{i}PS^e με την `branch_and_bound`. Η βιβλιοθήκη αυτή παρέχει ουσιαστικά το κατηγόρημα που περιγράφεται στη συνέχεια, το οποίο υλοποιεί τη διαδικασία «διακλάδωσε και φράξε»:

`bb_min/3`

Το κατηγόρημα καλείται ως `bb_min(`$Goal$`,`$Cost$`,`$Options$`)`, όπου το $Goal$ είναι ο στόχος αναζήτησης λύσεων (συνήθως με κατηγόρημα `search/6`), το $Cost$ είναι μια μεταβλητή πεδίου που κωδικοποιεί το κόστος μιας λύσης και το $Options$ είναι μια δομή της μορφής `bb_options{...}`. Στα ... υπάρχουν εκφράσεις της μορφής $Option$`:`$Value$, χωρισμένες με κόμματα, που καθορίζουν διάφορες επιλογές λειτουργίας για τη διαδικασία «διακλάδωσε και φράξε». Συνηθισμένοι συνδυασμοί επιλογών/τιμών είναι οι εξής:

- `strategy:continue` - Μετά την εύρεση λύσης, η αναζήτηση για επόμενη λύση με μικρότερο κόστος συνεχίζει από το σημείο αυτό.
- `strategy:restart` - Μετά την εύρεση λύσης, η αναζήτηση για επόμενη λύση με μικρότερο κόστος αρχίζει από την αρχή.
- `from:`$From$ - Αρχικό κάτω φράγμα του κόστους είναι το $From$.
- `to:`To - Αρχικό άνω φράγμα του κόστους είναι το To.
- `delta:`$Delta$ - Ελάχιστη απόλυτη βελτίωση του κόστους μεταξύ των επαναλήψεων της διαδικασίας «διακλάδωσε και φράξε».
- `factor:`$Factor$ - Ελάχιστος λόγος βελτίωσης του κόστους μεταξύ των επαναλήψεων της διαδικασίας «διακλάδωσε και φράξε».
- `timeout:`$Timeout$ - Το $Timeout$ είναι το άνω φράγμα CPU χρόνου σε δευτερόλεπτα, για την εύρεση της λύσης από τη διαδικασία «διακλάδωσε και φράξε». Αν παρέλθει ο χρόνος αυτός, επιστρέφεται η τελευταία λύση που βρέθηκε.

Μια τυπική χρήση του κατηγορήματος `bb_min/3` είναι η εξής:

```
bb_min(search(List, 0, input_order, indomain, complete, []),
       Cost, bb_options{strategy:restart, timeout:300}
```

Παράδειγμα 6.6
Στο γενικευμένο πρόβλημα των N βασιλισσών ορίζουμε ότι σε κάθε λύση αντιστοιχεί ένα κόστος που υπολογίζεται ως το $max\{|2 \times i - j|\}$, όπου (i, j) είναι οι συντεταγμένες των βασιλισσών στη λύση (βασίλισσα στην i στήλη και j γραμμή). Για παράδειγμα, η λύση που φαίνεται στο Σχήμα 6.1 έχει κόστος το:

$$\begin{aligned} max\{|2\times 1-5|,|2\times 2-2|,|2\times 3-6|,|2\times 4-1|,|2\times 5-7|,|2\times 6-4|,|2\times 7-8|,|2\times 8-3|\} &= \\ max\{|-3|,|2|,|0|,|7|,|3|,|8|,|6|,|13|\} &= \\ 13 & \end{aligned}$$

Ένα πρόγραμμα ECL^{i}PS^e που βασίζεται στις βιβλιοθήκες `ic` και `branch_and_bound`, και λύνει το πρόβλημα αυτό είναι το εξής:

```
:- lib(ic).                % Φορτώστε τη βιβλιοθήκη ic
:- lib(branch_and_bound).  % Φορτώστε τη βιβλιοθήκη branch_and_bound

queensopt(N, Solution, Cost) :-
   length(Solution, N),    % Δημιουργία προτύπου λύσης
   Solution #:: 1..N,      % Πεδίο μεταβλητών της λύσης
   constrain(Solution, CostList, 1),  % Περιορισμοί και λίστα κοστών
   Cost #= max(CostList),  % Δήλωση κόστους λύσης
   bb_min(search(Solution, 0, first_fail, indomain_middle, complete, []),
          Cost, bb_options{strategy:restart}).   % Διακλάδωσε και φράξε

constrain([], [], _).
constrain([X|Xs], [abs(2*K-X)|CostList], K) :-
   noattack(X, Xs, 1),     % Μη απειλή βασίλισσας με τις επόμενες
   K1 is K + 1,
   constrain(Xs, CostList, K1).

noattack(_, [], _).
noattack(X, [Y|Ys], M) :-
   X #\= Y,                % Βασίλισσες όχι στην ίδια γραμμή
   X #\= Y-M,              % Βασίλισσες όχι στην ίδια ανιούσα διαγώνιο
   X #\= Y+M,              % Βασίλισσες όχι στην ίδια κατιούσα διαγώνιο
   M1 is M+1,
   noattack(X, Ys, M1).
```

Και παραδείγματα εκτέλεσης:

```
?- queensopt(8, Solution, Cost).
Solution = [4,2,8,6,1,3,5,7]
Cost = 9
?- queensopt(11, Solution, Cost).
Solution = [2,4,6,8,10,1,3,5,7,9,11]
Cost = 11
?- queensopt(16, Solution, Cost).
Solution = [2,4,6,8,10,12,14,16,1,3,5,7,9,11,13,15]
Cost = 17
?- queensopt(20, Solution, Cost).
Solution = [13,4,7,9,12,14,20,11,2,6,1,3,5,8,10,16,18,15,17,19]
Cost = 21
?- queensopt(21, Solution, Cost).
Solution = [8,4,11,9,6,16,13,2,18,14,1,3,5,7,10,12,20,15,17,19,21]
Cost = 21
?- queensopt(22, Solution, Cost).
Solution = [11,8,6,13,9,2,14,16,22,15,4,1,3,5,7,10,12,18,20,17,19,21]
Cost = 23
?- queensopt(23, Solution, Cost).
Solution = [7,9,4,6,15,11,18,16,13,2,20,1,3,5,8,10,12,14,22,17,19,21,23]
Cost = 23
```

Άσκηση 6.4

Το πρόβλημα της κατασκευής δυαδικών ακολουθιών χαμηλής αυτοσυσχέτισης είναι πολύ σημαντικό για διάφορες περιοχές εφαρμογών από τις τηλεπικοινωνίες, τη φυσική, τη χημεία κτλ. Το ζητούμενο στο πρόβλημα αυτό είναι να κατασκευασθεί μια ακολουθία από -1 και

+1, δεδομένου μήκους n, τέτοια ώστε το άθροισμα των τετραγώνων των αυτοσυσχετίσεων k-τάξης ($0 < k < n$) της ακολουθίας να είναι το ελάχιστο δυνατό. Συγκεκριμένα:

Έστω ότι η ζητούμενη ακολουθία είναι η S_i, όπου $0 \leq i < n$ και $S_i \in \{-1, +1\}$. Η αυτοσυσχέτιση k-τάξης C_k της ακολουθίας ορίζεται ως:

$$C_k = \sum_{i=0}^{n-k-1} S_i \cdot S_{i+k}$$

όπου $0 < k < n$. Ζητούμε ακολουθία μήκους n, για την οποία η ολική αυτοσυσχέτισή της:

$$E = \sum_{k=1}^{n-1} C_k^2$$

να είναι ελάχιστη.

Ορίστε μέσω των βιβλιοθηκών `ic` και `branch_and_bound` της ECL^{i}PS^e ένα κατηγόρημα `labs/3`, το οποίο, όταν καλείται ως `labs(N, Labs, AC)`, να επιστρέφει στη λίστα `Labs` ακολουθία μήκους `N` ελάχιστης αυτοσυσχέτισης. Το κατηγόρημα να επιστρέφει και την τιμή `AC` της ελάχιστης αυτοσυσχέτισης.

Άσκηση 6.5

Το πρόβλημα του χρωματισμού γράφου συνίσταται στην εύρεση του ελάχιστου αριθμού χρωμάτων που πρέπει να χρησιμοποιήσουμε για να χρωματίσουμε τους κόμβους δεδομένου γράφου, έτσι ώστε να μην υπάρχει ζευγάρι γειτονικών κόμβων που να έχουν το ίδιο χρώμα. Ο ελάχιστος αριθμός απαιτούμενων χρωμάτων ονομάζεται χρωματικός αριθμός του γράφου.

Για την αντιμετώπιση της άσκησης αυτής, θα χρησιμοποιήσουμε γράφους που κατασκευάζονται μέσω του κατηγορήματος `create_graph(N, D, G)`, το οποίο ορίζεται σε ECL^{i}PS^e ως εξής:

```
create_graph(NNodes, Density, Graph) :-
   cr_gr(1, 2, NNodes, Density, [], Graph).

cr_gr(NNodes, _, NNodes, _, Graph, Graph).
cr_gr(N1, N2, NNodes, Density, SoFarGraph, Graph) :-
   N1 < NNodes,
   N2 > NNodes,
   NN1 is N1 + 1,
   NN2 is NN1 + 1,
   cr_gr(NN1, NN2, NNodes, Density, SoFarGraph, Graph).
cr_gr(N1, N2, NNodes, Density, SoFarGraph, Graph) :-
   N1 < NNodes,
   N2 =< NNodes,
   rand(1, 100, Rand),
   (Rand =< Density ->
      append(SoFarGraph, [N1 - N2], NewSoFarGraph) ;
      NewSoFarGraph = SoFarGraph),
   NN2 is N2 + 1,
   cr_gr(N1, NN2, NNodes, Density, NewSoFarGraph, Graph).

rand(N1, N2, R) :-
   random(R1),
   R is R1 mod (N2 - N1 + 1) + N1.
```

Κατά την κλήση του κατηγορήματος `create_graph/3`, δίνονται το πλήθος `N` των κόμβων του γράφου και η πυκνότητά του `D` (ως ποσοστό των ακμών που υπάρχουν στο γράφο σε σχέση με όλες τις δυνατές ακμές) και επιστρέφεται ο γράφος `G` ως μια λίστα από ακμές της μορφής `N1-N2`, όπου `N1` και `N2` είναι οι δύο κόμβοι της ακμής (οι κόμβοι του γράφου παριστάνονται ως ακέραιοι από το `1` έως το `N`). Ένα παράδειγμα εκτέλεσης του κατηγορήματος αυτού είναι το εξής (το κατηγόρημα `seed/1` αρχικοποιεί τη γεννήτρια τυχαίων αριθμών της ECL^{i}PS^e):

```
?- seed(1), create_graph(9, 30, G).
G = [1-3,1-4,1-7,2-4,2-6,3-4,3-7,3-9,4-5,5-9,6-7]
```

Ορίστε το `color_graph(N, D, Col, C)`, έτσι ώστε, αφού δημιουργήσει ένα γράφο `N` κόμβων και πυκνότητας `D`, καλώντας το κατηγόρημα `create_graph/3`, να βρίσκει έναν βέλτιστο χρωματισμό του γράφου, επιστρέφοντας στο `Col` τη λίστα με τα χρώματα (ακέραιοι αριθμοί) των κόμβων και στο `C` τον χρωματικό αριθμό του γράφου.

Απαντήσεις ασκήσεων

Απάντηση άσκησης 6.1
Ένα πρόγραμμα Prolog που επιλύει το πρόβλημα είναι το εξής:

```
gtsendmory(List) :-
   List = [S, E, N, D, M, O, R, Y], % Λίστα μεταβλητών
   generate(List, [0,1,2,3,4,5,6,7,8,9]), % Δημιουργία υποψήφιας λύσης
   1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E =:=
      10000*M+1000*O+100*N+10*E+Y,  % Περιορισμός πρόσθεσης
   M \= 0,                 % Να μην αρχίζουν από 0 τα MORE και MONEY
   S \= 0.                 % Να μην αρχίζει από 0 το SEND

generate([], _).
generate([Var|List], Values) :-
   del(Var, Values, RestValues),
   generate(List, RestValues).

del(X, [X|L], L).          % Συνήθως ενσωματωμένο ως delete/3
del(X, [Y|L1], [Y|L2]) :-
   del(X, L1, L2).
```

Και η εκτέλεση:

```
?- gtsendmory(List).
List = [9, 5, 6, 7, 1, 0, 8, 2]   ;
no
?-
```

Απάντηση άσκησης 6.2
Το παρακάτω πρόγραμμα ECL^{i}PS^e επιλύει το πρόβλημα. Θεωρούμε όλες τις τιμές πολλαπλασιασμένες με το 100, ώστε το πρόβλημα να είναι επάνω σε ακέραιους:

```
:- lib(ic).

seven_eleven([A,B,C,D]) :-
   [A,B,C,D] #:: [1..711], % Πολλαπλασιάζουμε όλες τις τιμές επί 100
```

```
   A * B * C * D #= 711000000,
   A + B + C + D #= 711,
   A #=< B,                % Οι ανισότητες χρειάζονται
   B #=< C,                % για να αποφύγουμε την
   C #=< D,                % εύρεση συμμετρικών λύσεων
   search([A,B,C,D], 0, largest, indomain_split, complete, []).
```

Και η εκτέλεση:

```
?- seven_eleven(List).
List = [120, 125, 150, 316]     ;
no
```

Δηλαδή, τα τέσσερα προϊόντα κόστιζαν 1,20, 1,25, 1,50 και 3,16 ευρώ.

Απάντηση άσκησης 6.3
Το παρακάτω πρόγραμμα ECL^{i}PS^e επιλύει το πρόβλημα:

```
:- lib(ic).

fractions(List) :-
   List = [A,B,C,D,E,F,G,H,I],
   List #:: 1..9,
   alldifferent(List),
   X #= 10*B+C,           % Ο παρονομαστής BC
   Y #= 10*E+F,           % Ο παρονομαστής EF
   Z #= 10*H+I,           % Ο παρονομαστής HI
   A*Y*Z + D*X*Z + G*X*Y #= X*Y*Z, % Έχοντας κάνει τα κλάσματα ομώνυμα
   A*Y #=< D*X,     % Το πρώτο κλάσμα όχι μεγαλύτερο από το δεύτερο
   D*Z #=< G*Y,     % Το δεύτερο κλάσμα όχι μεγαλύτερο από το τρίτο
   search(List, 0, input_order, indomain, complete, []).
```

Και η εκτέλεση:

```
?- fractions(List).
List = [7, 6, 8, 5, 3, 4, 9, 1, 2]     ;
no
```

Δηλαδή, η λύση είναι η:

$$\frac{7}{68} + \frac{5}{34} + \frac{9}{12} = 1$$

Απάντηση άσκησης 6.4
Το παρακάτω πρόγραμμα ECL^{i}PS^e επιλύει το πρόβλημα:

```
:- lib(ic).
:- lib(branch_and_bound).

labs(N, Labs, Cost) :-
   length(Labs, N),           % Πρότυπο ζητούμενης ακολουθίας
   Labs #:: [-1,1],           % Ορισμός πεδίου μεταβλητών
   compute_auto_corrs_sq(Labs, Labs, AutoCorrsSq),
   Cost #= sum(AutoCorrsSq),  % Ολική αυτοσυσχέτιση της ακολουθίας
   bb_min(search(Labs, 0, input_order, indomain, complete, []), Cost, _).
```

```
compute_auto_corrs_sq([], [], []).
compute_auto_corrs_sq(Labs1, [_|Labs3], [AutoCorr*AutoCorr|AutoCorrsSq]) :-
   append(Labs2, [_], Labs1),
   inner_prod(Labs2, Labs3, AutoCorr),  % Αυτοσυσχέτιση k-τάξης
   compute_auto_corrs_sq(Labs2, Labs3, AutoCorrsSq).

inner_prod([], [], 0).
inner_prod([X|L1], [Y|L2], X*Y+InnerProd) :-
   inner_prod(L1, L2, InnerProd).
```

Ενδεικτικές εκτελέσεις είναι οι εξής:

```
?- labs(3, Labs, AC).
Labs = [-1,-1,1]
AC = 1
?- labs(4, Labs, AC).
Labs = [-1,-1,-1,1]
AC = 2
?- labs(5, Labs, AC).
Labs = [-1,-1,-1,1,-1]
AC = 2
?- labs(7, Labs, AC).
Labs = [-1,-1,-1,1,1,-1,1]
AC = 3
?- labs(10, Labs, AC).
Labs = [-1,-1,-1,-1,-1,1,1,-1,1,-1]
AC = 13
?- labs(13, Labs, AC).
Labs = [-1,-1,-1,-1,-1,1,1,-1,-1,1,-1,1,-1]
AC = 6
?- labs(15, Labs, AC).
Labs = [-1,-1,-1,-1,-1,1,1,-1,-1,1,1,-1,1,-1,1]
AC = 15
?- labs(17, Labs, AC).
Labs = [-1,-1,-1,-1,1,1,-1,-1,1,-1,-1,1,-1,1,-1,1,1]
AC = 32
?- labs(18, Labs, AC).
Labs = [-1,-1,-1,-1,-1,1,-1,1,1,-1,1,-1,-1,-1,1,1,-1,-1]
AC = 25
?- labs(19, Labs, AC).
Labs = [-1,-1,-1,-1,1,-1,1,-1,1,1,1,1,-1,-1,1,1,-1,1,1]
AC = 29
?- labs(20, Labs, AC).
Labs = [-1,-1,-1,-1,-1,1,-1,1,1,1,-1,1,-1,-1,1,1,1,-1,-1,1]
AC = 26
```

Απάντηση άσκησης 6.5

Το παρακάτω πρόγραμμα ECL^{i}PS^e επιλύει το πρόβλημα:

```
:- lib(ic).
:- lib(branch_and_bound).

color_graph(NNodes, Density, Nodes, Chromatic) :-
   create_graph(NNodes, Density, Graph),   % Δημιουργήστε τυχαίο γράφο
   def_vars(NNodes, Nodes),     % Ορισμός των μεταβλητών και των πεδίων τους
   state_constrs(Nodes, Graph), % Δήλωση των περιορισμών
```

```
   Chromatic #= max(Nodes),     % Ορισμός του χρωματικού αριθμού
   bb_min(search(Nodes, 0, input_order, indomain, complete, []),
          Chromatic, _).

def_vars(NNodes, Nodes) :-
   length(Nodes, NNodes),
   Nodes #:: 1..NNodes.

state_constrs(_, []).
state_constrs(Nodes, [N1 - N2| Graph]) :-
   n_th(N1, Nodes, Node1),
   n_th(N2, Nodes, Node2),
   Node1 #\= Node2,      % Διαφορετικό χρώμα σε γειτονικούς κόμβους
   state_constrs(Nodes, Graph).

n_th(1, [Node| _], Node).
n_th(N, [_| Nodes], Node) :-
   N \= 1,
   N1 is N - 1,
   n_th(N1, Nodes, Node).
```

Ενδεικτικές εκτελέσεις (σε λειτουργικό σύστημα Windows 10 x64 - σε άλλο περιβάλλον, ενδέχεται οι τυχαίοι γράφοι να είναι διαφορετικοί) είναι οι εξής:

```
?- seed(1), color_graph(10, 80, Col, C).
Col = [1,2,3,2,4,1,5,5,4,3]
C = 5
?- seed(2000), color_graph(15, 60, Col, C).
Col = [1,2,2,1,2,2,3,3,4,4,4,3,5,6,6]
C = 6
?- seed(12345), color_graph(20, 70, Col, C).
Col = [1,1,2,2,3,4,4,5,6,6,7,7,5,8,2,1,3,5,4,3]
C = 8
?- seed(1000), color_graph(25, 50, Col, C).
Col = [1,1,2,1,3,4,2,1,3,2,4,3,5,6,4,7,7,5,7,7,6,1,5,5,2]
C = 7
?- seed(9876), color_graph(50, 20, Col, C).
Col = [1,1,1,1,2,2,1,2,3,3,1,3,2,4,4,1,5,1,4,2,4,5,4,2,5,
       3,1,4,2,3,5,4,3,3,3,5,5,4,4,2,1,2,3,4,5,5,2,3,1,2]
C = 5
?- seed(10), color_graph(40, 34, Col, C).
Col = [1,1,2,1,2,1,2,3,1,1,4,3,5,5,2,5,6,4,6,2,3,2,3,4,6,
       5,1,3,4,3,5,2,4,6,5,3,2,6,5,1]
C = 6
```

Προβλήματα

Πρόβλημα 6.1

Γράψτε ένα πρόγραμμα Prolog που να επιλύει το πρόβλημα του γρίφου του Παραδείγματος 6.1 με τη μέθοδο της οπισθοδρόμησης, δηλαδή χωρίς χρήση προγραμματισμού με περιορισμούς. Ενδεχομένως, θα χρειαστεί να αντικαταστήσετε τον περιορισμό που εκφράζει την ορθότητα της πρόσθεσης με ένα σύνολο στοιχειωδέστερων περιορισμών, έναν για το άθροισμα των μονάδων, έναν για τις δεκάδες κ.ο.κ.

Πρόβλημα 6.2

Επιλύστε μέσω λογικού προγραμματισμού με περιορισμούς το γρίφο:

$$\begin{array}{cccccccc} & D & O & N & A & L & D \\ + & G & E & R & A & L & D \\ \hline & R & O & B & E & R & T \end{array}$$

με λογική παρόμοια αυτής που ακολουθήσαμε στο Παράδειγμα 6.4, για το γρίφο $SEND + MORE = MONEY$.

Πρόβλημα 6.3
Το πρόβλημα της διαμέρισης αριθμών είναι από τα σημαντικότερα στη Θεωρητική Πληροφορική. Σε μια εκδοχή του προβλήματος, το ζητούμενο είναι να διαμερίσουμε το σύνολο των ακέραιων αριθμών $\{1, 2, 3, \ldots, N\}$, για δεδομένο N, σε δύο σύνολα A και B, τέτοια ώστε:

1. Τα σύνολα A και B να έχουν το ίδιο πλήθος στοιχείων.[1]
2. Το άθροισμα των στοιχείων του A να ισούται με το άθροισμα των στοιχείων του B.
3. Το άθροισμα των τετραγώνων των στοιχείων του A να ισούται με το άθροισμα των τετραγώνων των στοιχείων του B.

Χρησιμοποιώντας τη βιβλιοθήκη `ic`, ορίστε σε ECL^{i}PS^e ένα κατηγόρημα `numpart/3`, το οποίο, όταν καλείται ως `numpart(N, A, B)`, για δεδομένο `N`, να επιστρέφει στα `A` και `B` δύο λίστες που προκύπτουν από τη διαμέριση του συνόλου $\{1, 2, 3, \ldots, \texttt{N}\}$, σύμφωνα με τα παραπάνω. Κάποια παραδείγματα εκτέλεσης είναι τα εξής:

```
?- numpart(8, A, B).
A = [1, 4, 6, 7]
B = [2, 3, 5, 8]
?- numpart(20, A, B).
A = [1, 2, 4, 10, 12, 13, 14, 15, 16, 18]
B = [3, 5, 6, 7, 8, 9, 11, 17, 19, 20]      ;
A = [1, 2, 5, 9, 11, 13, 14, 15, 17, 18]
B = [3, 4, 6, 7, 8, 10, 12, 16, 19, 20]     ;
.....................................
```

Πρόβλημα 6.4
Στο συμμετρικό πρόβλημα του πλανόδιου πωλητή, είναι δεδομένο ένα σύνολο από πόλεις και, για κάθε πιθανό ζευγάρι πόλεων C_1 και C_2, είναι γνωστό το κόστος μετάβασης από τη μία προς την άλλη, το ίδιο για οποιαδήποτε από τις δύο πιθανές κατευθύνσεις. Αυτό το κόστος μπορεί να είναι, για παράδειγμα, η απόσταση των δύο πόλεων. Να βρεθεί με ποια σειρά πρέπει να επισκεφθεί ο πωλητής όλες τις πόλεις, αρχίζοντας από κάποια και καταλήγοντας σε αυτήν από την οποία ξεκίνησε, ώστε το συνολικό κόστος των μεταβάσεών του να είναι το ελάχιστο δυνατό.

Δεδομένα για το πρόβλημα, στη μορφή ενός γεγονότος Prolog, που αντιστοιχούν σε ένα δίκτυο 12 πόλεων, δίνονται στη συνέχεια:

```
costs([[31,12,7,2,17,19,25,21,30,10,32],
       [13,11,8,29,23,14,17,4,21,11],
       [9,14,28,25,11,32,14,10,8],
       [22,11,15,8,13,7,23,25],
```

[1] Προφανώς, το πρόβλημα αποκλείεται να έχει λύση αν το N είναι περιττός.

```
        [18,19,3,29,5,18,34],
        [14,30,9,22,17,11],
        [22,11,10,8,19],
        [8,24,42,33],
        [17,6,32],
        [15,28],
        [22]]).
```

Η λίστα που δίνεται ως όρισμα στο κατηγόρημα `costs/1` περιλαμβάνει στοιχεία λίστες, όπου η πρώτη περιέχει τα κόστη μετάβασης από την 1η πόλη προς όλες τις επόμενές της (2η, 3η κτλ.), η δεύτερη λίστα περιλαμβάνει τα κόστη μετάβασης από τη 2η πόλη προς όλες τις επόμενές της (3η, 4η κτλ.) κ.ο.κ. Το τελευταίο στοιχείο (22) είναι το κόστος μετάβασης από την 11η πόλη στη 12η (ή αντίστροφα).

Χρησιμοποιώντας τις βιβλιοθήκες `ic` και `branch_and_bound`, ορίστε σε ECL^{i}PS^e ένα κατηγόρημα `tsp/1`, το οποίο, όταν καλείται ως `tsp(N, R, C)`, να επιλύει το συμμετρικό πρόβλημα του πλανόδιου πωλητή, λαμβάνοντας ως είσοδο τις `N` τελευταίες πόλεις των δεδομένων που έχουν οριστεί μέσω του κατηγορήματος `costs/1` (δηλαδή τα `N-1` τελευταία στοιχεία του ορίσματός του). Το κατηγόρημα να επιστρέφει στο `R` τη βέλτιστη σειρά επίσκεψης των `N` πόλεων και στο `C` το βέλτιστο συνολικό κόστος. Παραδείγματα εκτέλεσης είναι τα εξής:

```
?- tsp(1, R, C).
R = [1]
C = 0
?- tsp(2, R, C).
R = [1, 2]
C = 44
?- tsp(3, R, C).
R = [1, 2, 3]
C = 65
?- tsp(4, R, C).
R = [1, 2, 4, 3]
C = 73
?- tsp(5, R, C).
R = [1, 2, 4, 5, 3]
C = 88
?- tsp(6, R, C).
R = [1, 4, 2, 3, 5, 6]
C = 89
?- tsp(7, R, C).
R = [1, 4, 3, 5, 2, 6, 7]
C = 92
?- tsp(8, R, C).
R = [1, 4, 5, 2, 8, 7, 3, 6]
C = 76
?- tsp(9, R, C).
R = [1, 3, 9, 4, 8, 6, 5, 2, 7]
C = 78
?- tsp(10, R, C).
R = [1, 2, 6, 3, 8, 5, 9, 7, 4, 10]
C = 77
?- tsp(11, R, C).
R = [1, 4, 7, 3, 2, 11, 5, 8, 10, 6, 9]
C = 84
?- tsp(12, R, C).
```

```
R = [1, 4, 2, 10, 7, 11, 9, 6, 12, 3, 8, 5]
C = 90
```

Πρόβλημα 6.5
Στη Θεωρητική Πληροφορική, το πρόβλημα της βέλτιστης κάλυψης κορυφών ενός γράφου συνίσταται στην εύρεση ενός συνόλου κορυφών, ελάχιστου πλήθους, τέτοιου ώστε, για κάθε ακμή του γράφου, τουλάχιστον μία από τις κορυφές-άκρα της να ανήκει στο σύνολο αυτό.

Για την αντιμετώπιση του προβλήματος αυτού, χρησιμοποιήστε γράφους που κατασκευάζονται με το κατηγόρημα `create_graph(N, D, G)`, από την Άσκηση 6.5. Χρησιμοποιώντας τις βιβλιοθήκες `ic` και `branch_and_bound` της ECL^{i}PS^e, ορίστε ένα κατηγόρημα `vertexcover/3`, το οποίο, όταν καλείται ως `vertexcover(N, D, C)`, να δημιουργεί, καλώντας το κατηγόρημα `create_graph/3`, ένα γράφο `N` κόμβων και πυκνότητας `D`, και να επιστρέφει στο `C` μια λίστα από κόμβους που αποτελούν μια βέλτιστη κάλυψη κορυφών του γράφου. Κάποια παραδείγματα εκτέλεσης (σε λειτουργικό σύστημα Windows 10 x64) είναι τα εξής:

```
?- seed(2000), vertexcover(9, 30, C).
C = [2,5,6,7]
?- seed(1000), vertexcover(17, 50, C).
C = [1,2,3,6,7,8,9,11,14,15,16,17]
?- seed(12345), vertexcover(33, 65, C).
C = [3,4,5,7,8,9,10,11,12,13,14,15,17,18,19,20,21,22,
     23,24,25,26,27,28,29,30,31,32]
?- seed(1), vertexcover(100, 70, C), length(C, L).
C = .....................
L = 94
?- seed(100), vertexcover(100, 35, C), length(C, L).
C = .....................
L = 88
```

Πρόβλημα 6.6
Αντιμετωπίστε το Πρόβλημα 5.29 μέσω λογικού προγραμματισμού με περιορισμούς.

Βιβλιογραφικές αναφορές

[1] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, The MIT Press, 1989.

[2] K. R. Apt and M. G. Wallace, *Constraint Logic Programming Using ECL^{i}PS^e*, Cambridge University Press, 2007.

[3] N. Pothitos, *Naxos Solver*, `http://di.uoa.gr/~pothitos/naxos`, 2015.

[4] *ECL^{i}PS^e Constraint Library Manual*, Release 6.1 2015.

[5] *ECL^{i}PS^e User Manual*, Release 6.1, 2015.

[6] *ECL^{i}PS^e 6.1 Reference Manual*, 2015.

Κεφάλαιο 7
Λογική πρώτης τάξης

Σύνοψη

Στο κεφάλαιο αυτό περιγράφονται το συντακτικό της λογικής πρώτης τάξης, δηλαδή πώς πρέπει να είναι διατυπωμένοι οι τύποι στη λογική αυτή, και, ειδικότερα, οι προτάσεις, μια μορφή τύπων, ειδική περίπτωση των οποίων είναι οι προτάσεις Horn. Επίσης, παρουσιάζεται ο μηχανισμός της ενοποίησης όρων ή τύπων, μέσα από μια αλγοριθμική περιγραφή του. Τέλος, γίνεται αναφορά στην αρχή της ανάλυσης, καθώς και σε μια ειδική περίπτωσή της, το modus ponens, που είναι κανόνες συμπερασμού για την παραγωγή νέων προτάσεων από ήδη υπάρχουσες.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης δεν απαιτείται να έχει ειδικές γνώσεις. Ωστόσο, στοιχειώδης γνώση της Prolog, όπως αυτή παρουσιάζεται στο Κεφάλαιο 3, θα βοηθούσε.

7.1 Προτάσεις Horn

Στο Κεφάλαιο 2 πήρατε μια πρώτη ιδέα του τρόπου με τον οποίο μπορεί να χρησιμοποιηθεί η **λογική** για την επίλυση προβλημάτων. Το εργαλείο που βοηθά στην επίτευξη αυτού του στόχου είναι ο **λογικός προγραμματισμός** [1, 2, 3, 4], μέσω της εφαρμογής του στην πράξη από τη γλώσσα προγραμματισμού Prolog [5]. Σε αυτό το κεφάλαιο, αλλά και στο επόμενο, θα δούμε τα βασικά στοιχεία της θεμελίωσης του λογικού προγραμματισμού στη λογική.

Το θεωρητικό υπόβαθρο του λογικού προγραμματισμού είναι η **λογική πρώτης τάξης**, που πολλές φορές αναφέρεται και ως **κατηγορηματική λογική πρώτης τάξης** ή, απλώς, **κατηγορηματική λογική**. Η λογική πρώτης τάξης είναι γενίκευση της **προτασιακής λογικής**. Στοιχεία από την προτασιακή λογική και τη λογική πρώτης τάξης είδαμε στο Κεφάλαιο 2.

Σε αυτήν την ενότητα, καταρχάς ενδιαφερόμαστε και θα παρουσιάσουμε την αξιωματική θεμελίωση των **προτάσεων Horn** στη λογική πρώτης τάξης, οι οποίες δεν είναι τίποτε άλλο από τη θεωρητική εκδοχή των προτάσεων που περιλαμβάνουμε στα προγράμματα Prolog.

Στη λογική πρώτης τάξης, ορίζουμε τα εξής σύνολα:

P: σύνολο **κατηγορημάτων**
F: σύνολο **συναρτησιακών συμβόλων**
V: σύνολο **μεταβλητών**

Σε κάθε κατηγόρημα ή συναρτησιακό σύμβολο k αντιστοιχεί ένας ακέραιος αριθμός $n \geq 0$, που ονομάζεται **τάξη** (ή **βαθμός**) του k. Το k συμβολίζεται και ως k/n. Τα συναρτησιακά σύμβολα βαθμού 0 ονομάζονται και **σταθερές**. Στη λογική πρώτης τάξης, χρησιμοποιούμε τα **σημεία στίξης** «(», «)» και «,».

Ορισμός 7.1
Ένας **όρος** ορίζεται ως εξής:

1. Αν $x \in V$, το x είναι όρος.
2. Αν $c/0 \in F$, το c είναι όρος.
3. Αν $f/n \in F$ και t_1, t_2, ..., t_n είναι όροι, το $f(t_1, t_2, \ldots, t_n)$ είναι όρος.

Παράδειγμα 7.1
Αν $V = \{x\}$ και $F = \{john/0,\ mary/0,\ father\_of/1,\ children\_of/2\}$, τότε τα:

x
$john$
$father\_of(john)$
$children\_of(father\_of(x), mary)$

είναι όροι.

Ορισμός 7.2
Ένας **ατομικός τύπος** (ή **άτομο**) ορίζεται ως εξής:

1. Αν $p/0 \in P$, τότε το p είναι άτομο.
2. Αν $p/n \in P$ και t_1, t_2, ..., t_n είναι όροι, τότε το $p(t_1, t_2, \ldots, t_n)$ είναι άτομο.

Παράδειγμα 7.2
Αν $V = \{x\},\ F = \{john/0,\ father\_of/1\}$ και $P = \{it\_rains/0,\ young/1,\ father/2\}$, τότε τα:

it_rains
$young(john)$
$father(father\_of(x), x)$

είναι άτομα.

Ορισμός 7.3
Ένας όρος που δεν περιέχει μεταβλητές ονομάζεται **βασικός όρος**. Ένα άτομο που δεν περιέχει μεταβλητές ονομάζεται **βασικό άτομο**. □

Για τη δόμηση πιο σύνθετων τύπων, στη λογική πρώτης τάξης χρησιμοποιούνται τα **συνδετικά**, που είναι τα εξής:

$\neg$:	άρνηση		
$\wedge$:	σύζευξη	$\vee$:	διάζευξη
$\rightarrow$:	συνεπαγωγή	$\leftrightarrow$:	ισοδυναμία
$\forall$:	καθολικός ποσοδείκτης	$\exists$:	υπαρξιακός ποσοδείκτης

Ορισμός 7.4
Ένας **καλοσχηματισμένος τύπος** (ή **τύπος**) ορίζεται ως εξής:

1. Αν το A είναι άτομο, τότε το A είναι τύπος.
2. Αν τα F_1 και F_2 είναι τύποι, τότε και τα $(\neg F_1)$, $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, $(F_1 \rightarrow F_2)$,[1] $(F_1 \leftrightarrow F_2)$ είναι τύποι.
3. Αν το F είναι τύπος και $x \in V$, τότε τα $(\forall x)(F)$ και $(\exists x)(F)$ είναι τύποι.

Παράδειγμα 7.3
Αν $V = \{x,\ y\},\ F = \{john/0,\ mary/0\}$ και $P = \{man/1,\ woman/1,\ house/2,\ loves/2,\ hates/2\}$, τότε τα:

$$loves(mary, x)$$
$$(man(john) \wedge ((\neg hates(john, john)) \vee (\neg loves(john, john))))$$
$$(\neg(\exists x)(loves(x, y)))$$
$$(\forall x)((\exists y)(house(x, y)))$$
$$(\exists y)((\forall x)(house(x, y)))$$
$$(\forall x)((man(x) \rightarrow (\exists y)((woman(y) \wedge loves(x, y)))))$$

είναι τύποι. □

Υιοθετώντας μια ιεραρχία των συνδετικών, τέτοια ώστε η άρνηση $\neg$ και οι ποσοδείκτες (καθολικός $\forall$ και υπαρξιακός $\exists$) να είναι ισχυρότερα από τη διάζευξη $\vee$, αυτή ισχυρότερη από τη σύζευξη $\wedge$ και αυτή από τη συνεπαγωγή $\rightarrow$ και την ισοδυναμία $\leftrightarrow$, μπορούμε να γράφουμε τους τύπους απλούστερα, αποφεύγοντας αρκετές παρενθέσεις, χωρίς ασάφεια.

Παράδειγμα 7.4
Με βάση την προηγούμενη απλούστευση, οι τύποι του Παραδείγματος 7.3 θα μπορούσαν να γραφούν ως εξής:

$$loves(mary, x)$$
$$man(john) \wedge \neg hates(john, john) \vee \neg loves(john, john)$$
$$\neg(\exists x)(loves(x, y))$$
$$(\forall x)(\exists y)(house(x, y))$$
$$(\exists y)(\forall x)(house(x, y))$$
$$(\forall x)(man(x) \rightarrow (\exists y)(woman(y) \wedge loves(x, y)))$$

[1] Μερικές φορές, είναι βολικό το $F_1 \rightarrow F_2$ να γράφεται $F_2 \leftarrow F_1$.

Ορισμός 7.5
Δεδομένων ενός συνόλου κατηγορημάτων P, ενός συνόλου συναρτησιακών συμβόλων F και ενός συνόλου μεταβλητών V, το σύνολο όλων των συντακτικά αποδεκτών τύπων, βάσει του ορισμού 7.4, αποτελεί μια **γλώσσα πρώτης τάξης**.

Ορισμός 7.6
Η **εμβέλεια** του $\forall$ στον τύπο $(\forall x)(F)$ ή του $\exists$ στον τύπο $(\exists x)(F)$ είναι ο τύπος F.

Ορισμός 7.7
Δεσμευμένη εμφάνιση μεταβλητής σε έναν τύπο είναι μια εμφάνισή της αμέσως μετά από έναν ποσοδείκτη ή μέσα στην εμβέλεια ενός ποσοδείκτη που εφαρμόζεται στη μεταβλητή. Οποιαδήποτε άλλη εμφάνιση της μεταβλητής ονομάζεται **ελεύθερη εμφάνιση**.

Παράδειγμα 7.5
Στους τύπους του Παραδείγματος 7.4, ελεύθερες εμφανίσεις μεταβλητών είναι:

- η εμφάνιση της x στον τύπο $loves(mary, x)$
- η εμφάνιση της y στον τύπο $\neg(\exists x)(loves(x, y))$

Όλες οι άλλες εμφανίσεις μεταβλητών είναι δεσμευμένες.

Ορισμός 7.8
Ένας τύπος ονομάζεται **κλειστός** όταν δεν περιέχει ελεύθερες εμφανίσεις μεταβλητών.

Παράδειγμα 7.6
Από τους τύπους του Παραδείγματος 7.4, οι:

$$man(john) \wedge \neg hates(john, john) \vee \neg loves(john, john)$$
$$(\forall x)(\exists y)(house(x, y))$$
$$(\exists y)(\forall x)(house(x, y))$$
$$(\forall x)(man(x) \rightarrow (\exists y)(woman(y) \wedge loves(x, y)))$$

είναι κλειστοί, αλλά όχι οι:

$$loves(mary, x)$$
$$\neg(\exists x)(loves(x, y))$$

Ορισμός 7.9
Στοιχειώδης τύπος είναι ένα άτομο ή η άρνηση ενός ατόμου.

Ορισμός 7.10
Οι τύποι της μορφής:

$$(\forall x_1)(\forall x_2) \ \ldots \ (\forall x_s)(L_1 \vee L_2 \vee \ldots \vee L_m)$$

καλούνται **προτάσεις**, όπου τα L_i $(1 \leq i \leq m)$ είναι στοιχειώδεις τύποι και τα x_j $(1 \leq j \leq s)$ είναι όλες οι μεταβλητές που εμφανίζονται στο $L_1 \vee L_2 \vee \ldots \vee L_m$.

Παράδειγμα 7.7
Αν $V = \{x,\ y\ z\},\ F = \{john/0,\ a\_building/0\}$ και $P = \{man/1,\ mortal/1,$ $parent/2,\ grandparent/2,\ rich/1,\ has\_big\_house/1,\ has\_expensive\_car/1\}$, τότε τα:

$man(john)$
$\neg man(a\_building)$
$(\forall x)(mortal(x) \vee \neg man(x))$
$(\forall x)(\forall y)(\forall z)(grandparent(x,z) \vee \neg parent(x,y) \vee \neg parent(y,z))$
$(\forall x)(has\_big\_house(x) \vee has\_expensive\_car(x) \vee \neg rich(x))$

είναι προτάσεις. □

Υπάρχει η δυνατότητα και εναλλακτικής γραφής για τις προτάσεις. Συγκεκριμένα, η πρόταση:

$$(\forall x_1)(\forall x_2)\ \ldots\ (\forall x_s)(A_1 \vee A_2 \vee \ldots \vee A_k \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n)$$

όπου τα A_i $(1 \leq i \leq k)$ και B_j $(1 \leq j \leq n)$ είναι άτομα, γράφεται και ως εξής:

$$A_1,\ A_2,\ \ldots\ ,\ A_k \leftarrow B_1,\ B_2,\ \ldots\ ,\ B_n$$

Το αριστερό μέλος του $\leftarrow$ ονομάζεται **κεφαλή** της πρότασης, ενώ το δεξιό μέλος ονομάζεται **σώμα** της πρότασης. Δεν αποκλείεται κάποια πρόταση να έχει κενή κεφαλή $(k = 0)$ ή να έχει κενό σώμα $(n = 0)$ ή και τα δύο.

Παράδειγμα 7.8
Με την προηγούμενη εναλλακτική γραφή, οι προτάσεις του Παραδείγματος 7.7 θα μπορούσαν να γραφούν ως εξής:

$man(john) \leftarrow$
$\leftarrow man(a\_building)$
$mortal(x) \leftarrow man(x)$
$grandparent(x,z) \leftarrow parent(x,y),\ parent(y,z)$
$has\_big\_house(x),\ has\_expensive\_car(x) \leftarrow rich(x)$

Ορισμός 7.11
Οι προτάσεις με $k = 1$ $(A \leftarrow B_1,\ B_2,\ \ldots\ ,\ B_n)$, σύμφωνα με την εναλλακτική γραφή που προτάθηκε, ονομάζονται **οριστικές προτάσεις**. Οι οριστικές προτάσεις με $n = 0$ $(A \leftarrow)$ ονομάζονται **μοναδιαίες προτάσεις**. Ένα σύνολο οριστικών προτάσεων είναι ένα **οριστικό πρόγραμμα**, που πολλές φορές αναφέρεται και ως **λογικό πρόγραμμα**. □

Ένα πρόγραμμα Prolog που δεν περιλαμβάνει ούτε αποκοπές, ούτε αρνήσεις, ούτε οποιοδήποτε άλλο ενσωματωμένο κατηγόρημα είναι ένα οριστικό (ή λογικό) πρόγραμμα.

Ορισμός 7.12
Οι προτάσεις με $k = 0$ $(\leftarrow B_1,\ B_2,\ \ldots\ ,\ B_n)$, σύμφωνα με την εναλλακτική γραφή που προτάθηκε, ονομάζονται **οριστικοί στόχοι**.

Ορισμός 7.13
Για $k = 0$ και $n = 0$, στην εναλλακτική γραφή των προτάσεων, έχουμε την **κενή πρόταση** ($\leftarrow$ ή $\Box$), που αναφέρεται και ως **αντίφαση**.

Παράδειγμα 7.9
Από τις προτάσεις του Παραδείγματος 7.8, η:

$$man(john) \leftarrow$$

είναι οριστική μοναδιαία πρόταση, η:

$$\leftarrow man(a\_building)$$

είναι οριστικός στόχος, οι:

$$mortal(x) \leftarrow man(x)$$
$$grandparent(x, z) \leftarrow parent(x, y),\ parent(y, z)$$

είναι οριστικές προτάσεις, όχι όμως και η:

$$has\_big\_house(x),\ has\_expensive\_car(x) \leftarrow rich(x)$$

Ορισμός 7.14
Μια **πρόταση Horn** είναι είτε οριστική πρόταση είτε οριστικός στόχος.

Άσκηση 7.1
Δίνονται οι εξής τύποι λογικής πρώτης τάξης:

1. $borders(x, greece)$
2. $(\exists x)(man(x) \wedge (\forall y)(woman(y) \rightarrow loves(y, x)))$
3. $(\forall x)(\forall y)(may\_steal(x, y) \vee \neg likes(x, y) \vee \neg thief(x))$
4. $connected(x, y) \wedge connected(y, z) \rightarrow connected(x, z)$
5. $on\_diet(x),\ athlete(x) \leftarrow fit(x)$
6. $\neg loves(john, mary) \vee \neg loves(mary, john)$
7. $father(x, y) \leftarrow male(x),\ parent(x, y)$
8. $append(.(a, []), .(b, []), .(a, .(b, [])))$

Αν το σύνολο των μεταβλητών που χρησιμοποιήθηκαν στους τύπους αυτούς είναι το $V = \{x,\ y,\ z\}$, τότε ποια πρέπει να είναι τα σύνολα των συναρτησιακών συμβόλων F και κατηγορημάτων P, για να είναι οι τύποι αυτοί συντακτικά σωστοί; Ποιοι από αυτούς τους τύπους είναι κλειστοί και ποιοι όχι; Αυτοί που δεν είναι κλειστοί, γιατί δεν είναι; Ποιοι τύποι είναι προτάσεις διατυπωμένες με τον κλασικό τρόπο της λογικής πρώτης τάξης; Διατυπώστε τες και με τον εναλλακτικό απλουστευμένο τρόπο. Υπάρχουν προτάσεις στους προηγούμενους τύπους, ήδη διατυπωμένες με τον εναλλακτικό τρόπο; Ποιες από τις προτάσεις (με όποιο τρόπο και αν είναι διατυπωμένες) είναι προτάσεις Horn και ποιες όχι; Αυτές που

δεν είναι, γιατί δεν είναι; Ποιες από τις προτάσεις Horn είναι οριστικές προτάσεις και ποιες οριστικοί στόχοι; Από τις οριστικές προτάσεις, είναι κάποιες μοναδιαίες;

Άσκηση 7.2
Έστω ότι, για μια γλώσσα πρώτης τάξης, έχουμε σύνολο μεταβλητών το $V = \{x,\ y\}$, συναρτησιακών συμβόλων το $F = \{liz/0,\ mother\_of/1\}$ και κατηγορημάτων το $P = \{alive/1\}$. Ποιο είναι το σύνολο των βασικών όρων που μπορούμε να έχουμε στη γλώσσα αυτή; Ποιο είναι το σύνολο των βασικών ατόμων; (Θυμηθείτε τον Ορισμό 7.3.) Γράψτε πέντε τύπους της γλώσσας που ορίζεται από τα προηγούμενα σύνολα.

7.2 Ενοποίηση

Από τα βασικά «εργαλεία» που είναι απαραίτητα για να περιγράψουμε τον τρόπο με τον οποίο μπορεί στη λογική πρώτης τάξης να παραχθεί νέα γνώση από ήδη υπάρχουσα, δηλαδή να κατασκευαστεί ένας νέος τύπος από άλλους τύπους λογικής πρώτης τάξης, είναι η διαδικασία της **ενοποίησης**. Για να περιγράψουμε αυτόν το μηχανισμό, πρέπει να δώσουμε πρώτα κάποιους απαιτούμενους ορισμούς.

Ορισμός 7.15
Μια **αντικατάσταση** θ είναι ένα πεπερασμένο σύνολο $\{v_1/t_1,\ v_2/t_2,\ \ldots\ ,\ v_n/t_n\}$, όπου τα v_i είναι μεταβλητές και τα t_i είναι όροι ($1 \leq i \leq n$), κάθε t_i είναι διαφορετικό από το αντίστοιχο v_i και όλα τα v_i είναι διαφορετικά μεταξύ τους. Ουσιαστικά, μια αντικατάσταση ορίζει ότι τα v_i είναι δεσμευμένα (έχουν πάρει τιμές) με τα αντίστοιχα t_i. Η **κενή αντικατάσταση** ορίζεται από το κενό σύνολο.

Ορισμός 7.16
Αν E είναι όρος ή πρόταση, τότε το $E\theta$ ονομάζεται **στιγμιότυπο** του E, μέσω της αντικατάστασης $\theta = \{v_1/t_1,\ v_2/t_2,\ \ldots\ ,\ v_n/t_n\}$, αν στο E αντικατασταθούν ταυτόχρονα οι εμφανίσεις των μεταβλητών v_i με τα αντίστοιχα t_i.

Ορισμός 7.17
Αν E και F είναι όροι ή προτάσεις, τότε το E ονομάζεται **παραλλαγή** του F (ή το F παραλλαγή του E), αν υπάρχουν αντικαταστάσεις θ και σ, τέτοιες ώστε $E = F\theta$ και $F = E\sigma$. □

Ουσιαστικά, κάποια E και F είναι παραλλαγές το ένα του άλλου, αν είναι ακριβώς τα ίδια, με εξαίρεση τα ονόματα των μεταβλητών τους.

Ορισμός 7.18
Αν $\theta = \{u_1/s_1,\ u_2/s_2,\ \ldots\ ,\ u_m/s_m\}$ και $\sigma = \{v_1/t_1,\ v_2/t_2,\ \ldots\ ,\ v_n/t_n\}$ είναι δύο αντικαταστάσεις, τότε η **σύνθεσή** τους $\theta\sigma$ είναι η αντικατάσταση:

$$\{u_1/s_1\sigma,\ u_2/s_2\sigma,\ \ldots\ ,\ u_m/s_m\sigma,\ v_1/t_1,\ v_2/t_2,\ \ldots\ ,\ v_n/t_n\}$$

διαγράφοντας από αυτήν κάθε $u_i/s_i\sigma$ ($1 \leq i \leq m$) για το οποίο $u_i = s_i\sigma$ και κάθε v_j/t_j ($1 \leq j \leq n$) για το οποίο $v_j \in \{u_1,\ u_2,\ \ldots\ ,\ u_m\}$.

Ορισμός 7.19
Αν E και F είναι όροι ή άτομα, τότε μια αντικατάσταση θ ονομάζεται **ενοποιητής**, για τα E και F, αν $E\theta = F\theta$. Η θ είναι ο **γενικότερος ενοποιητής**, για τα E και F (συμβολικά $mgu(E, F)$), αν για κάθε ενοποιητή σ των E και F, υπάρχει αντικατάσταση γ, τέτοια ώστε $\sigma = \theta\gamma$. Η διαδικασία υπολογισμού ενός $mgu(E, F)$ ονομάζεται *ενοποίηση*. □

Αλγόριθμος ενοποίησης όρων ή ατόμων **E** *και* **F**

Βήμα 1: *Αρχικοποιήστε τη στοίβα ζευγαριών όρων ή ατόμων προς ενοποίηση* **S** *με το ζευγάρι* **(E,F)**.
Βήμα 2: *Αρχικοποιήστε τον γενικότερο ενοποιητή* **MGU** *με την κενή αντικατάσταση.*
Βήμα 3: *Αν η στοίβα* **S** *είναι κενή, τερματίστε με επιτυχία δίνοντας ως γενικότερο ενοποιητή των* **E** *και* **F** *το* **MGU**.
Βήμα 4: *Βγάλτε από την* **S** *το πρώτο ζευγάρι* **(A,B)**.
Βήμα 5: *Αν το* **A** *είναι μεταβλητή που δεν περιέχεται στο* **B**, *αντικαταστήστε μέσα στη στοίβα* **S** *και στον γενικότερο ενοποιητή* **MGU** *όλες τις εμφανίσεις του* **A** *με το* **B**, *προσθέστε στο* **MGU** *τη δέσμευση* **A/B** *και πηγαίνετε στο βήμα 3.*
Βήμα 6: *Αν το* **B** *είναι μεταβλητή που δεν περιέχεται στο* **A**, *τότε αντικαταστήστε μέσα στη στοίβα* **S** *και στον γενικότερο ενοποιητή* **MGU** *όλες τις εμφανίσεις του* **B** *με το* **A**, *προσθέστε στο* **MGU** *τη δέσμευση* **B/A** *και πηγαίνετε στο βήμα 3.*
Βήμα 7: *Αν τα* **A** *και* **B** *είναι ίδιες μεταβλητές ή ίδιες σταθερές, τότε πηγαίνετε στο βήμα 3.*
Βήμα 8: *Αν το* **A** *είναι ο όρος ή το άτομο* **f(**$\mathbf{X}_1$**,** $\mathbf{X}_2$**,** $\ldots$ **,** $\mathbf{X}_n$**)** *και το* **B** *είναι ο όρος ή το άτομο* **f(**$\mathbf{Y}_1$**,** $\mathbf{Y}_2$**,** $\ldots$ **,** $\mathbf{Y}_n$**)**, *όπου το* **f** *είναι συναρτησιακό σύμβολο ή κατηγόρημα βαθμού* n, *τότε προσθέστε στη στοίβα* **S** *όλα τα ζευγάρια* **(**$\mathbf{X}_i$**,**$\mathbf{Y}_i$**)**, *για κάθε* i *με* $1 \leq i \leq n$, *και πηγαίνετε στο βήμα 3.*
Βήμα 9: *Αλλιώς, τερματίστε με αποτυχία, επισημαίνοντας ότι δεν υπάρχει ενοποιητής των* **E** *και* **F**.

Παράδειγμα 7.10
Δείτε μερικές περιπτώσεις ζευγαριών όρων ή ατόμων και τους γενικότερους ενοποιητές τους, όταν υπάρχουν. Τα σύμβολα x, y, z στο παράδειγμα αυτό είναι μεταβλητές. Οτιδήποτε άλλο είναι είτε συναρτησιακό σύμβολο είτε κατηγόρημα, αλλά δεν μας ενδιαφέρει τι ακριβώς, για το σκοπό του παραδείγματος.

	E	F	$mgu(E, F)$
1	x	$john$	$\{x/john\}$
2	$house\_of(mary)$	x	$\{x/house\_of(mary)\}$
3	$lawyer(y)$	$lawyer(jack)$	$\{y/jack\}$
4	x	y	$\{x/y\}$
5	$likes(john, x)$	$likes(x, mary)$	$\nexists\, mgu$
6	$hates(y, z)$	$hates(z, ann)$	$\{y/ann,\ z/ann\}$
7	$likes(x, house\_of(x))$	$likes(y, y)$	$\nexists\, mgu$
8	$.(a, .(b, .(c, .(d, []))))$	$.(x, .(y, z))$	$\{x/a,\ y/b,\ z/.(c, .(d, []))\}$

Όπου στην τρίτη στήλη υπάρχει το $\nexists\, mgu$, αυτό σημαίνει ότι τα E και F δεν ενοποιούνται. Ας πούμε όμως δύο λόγια για την περίπτωση 7, επειδή είναι λίγο ιδιόρρυθμη. Εδώ, τα E

και F δεν ενοποιούνται, γιατί, αφού γίνει η ενοποίηση των πρώτων ορισμάτων των $likes/2$ και προστεθεί στο γενικότερο ενοποιητή η δέσμευση x/y, θα πρέπει στη συνέχεια να ενοποιηθούν το $house\_of(y)$ με το y. Αυτό όμως δεν μπορεί να γίνει, γιατί το μόνο υποψήφιο βήμα στον αλγόριθμο ενοποίησης που δώσαμε είναι το βήμα 6, αλλά και αυτό δεν επιτρέπει την ενοποίηση, επειδή η μεταβλητή y περιέχεται στο $house\_of(y)$. Έτσι, αναγκαστικά ο αλγόριθμος καταλήγει στο βήμα 9, με συνέπεια να μην μπορεί να βρεθεί, σωστά, γενικότερος ενοποιητής. Αυτή η παρεμπόδιση της ενοποίησης μιας μεταβλητής με έναν όρο που την περιέχει ονομάζεται **έλεγχος εμφάνισης**, και είναι πολύ κρίσιμο να εξασφαλίζεται από συστήματα που χρησιμοποιούν διαδικασίες ενοποίησης. Επειδή όμως έχει μεγάλο υπολογιστικό κόστος, συνήθως παραλείπεται, όπως συμβαίνει στους αλγορίθμους ενοποίησης των περισσότερων συστημάτων Prolog.

Άσκηση 7.3
Αν $V = \{u,\ v,\ w,\ x,\ y,\ z,\ x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ x_7,\ x_8\}$ και $F = \{f/2,\ g/2,\ h/3,\ j/2,\ k/4,\ m/2,\ n/1,\ a/0,\ b/0\}$, βρείτε τους γενικότερους ενοποιητές των όρων E και F του επόμενου πίνακα:

	E	F
1	$f(n(v), m(u, v))$	$f(n(w), m(w, j(x, y)))$
2	$f(n(v), m(u, v))$	$f(n(w), m(w, j(x, u)))$
3	$f(x, f(u, x))$	$f(f(y, a), f(z, f(b, z)))$
4	$k(h(x_1, x_2, x_3), h(x_6, x_7, x_8), x_3, x_6)$	$k(h(g(x_4, x_5), x_1, x_2), h(x_7, x_8, x_6), g(x_5, a), x_5)$
5	$k(x_1, g(x_2, x_3), x_2, b)$	$k(g(m(a, x_5), x_2), x_1, m(a, x_4), x_4)$
6	$j(f(x, g(x, y)), m(z, y))$	$j(z, m(f(u, v), f(a, b)))$

7.3 Αρχή της ανάλυσης

Για να έχει πρακτική χρησιμότητα η λογική πρώτης τάξης, εκτός από αυτήν της αναπαράστασης γνώσης, πιο συγκεκριμένα, για να μπορεί να παράγεται νέα γνώση από ήδη υπάρχουσα, πρέπει να εφοδιαστεί με **κανόνες εξαγωγής συμπερασμάτων** ή, αλλιώς, **κανόνες συμπερασμού**. Ένας κανόνας συμπερασμού παρέχει το μέσο με το οποίο μπορούμε να κατασκευάσουμε έναν νέο τύπο από δύο ή περισσότερους άλλους τύπους. Ο βασικός κανόνας συμπερασμού που υπάρχει στη λογική πρώτης τάξης είναι η **αρχή της ανάλυσης**, την οποία θα περιγράψουμε σε αυτήν την ενότητα.

Η αρχή της ανάλυσης είναι ένας κανόνας συμπερασμού, με τη βοήθεια του οποίου, από δύο προτάσεις C_1 και C_2, παράγεται μια τρίτη πρόταση C. Συμβολικά, γράφουμε:

$$\{C_1, C_2\} \vdash C$$

Συγκεκριμένα, αν η πρόταση C_1 είναι η:

$$a_1,\ a_2,\ \ldots\ ,\ a_k \leftarrow b_1,\ b_2,\ \ldots\ ,\ b_n$$

και η πρόταση C_2 είναι η:

$$d_1,\ d_2,\ \ldots\ ,\ d_l \leftarrow e_1,\ e_2,\ \ldots\ ,\ e_m$$

και οι στοιχειώδεις τύποι a_i (για κάποιο i με $1 \leq i \leq k$) στην κεφαλή της C_1 και e_j (για κάποιο j με $1 \leq j \leq m$) στο σώμα της C_2 έχουν γενικότερο ενοποιητή το σ, δηλαδή $\sigma = mgu(a_i, e_j)$, τότε η παραγόμενη πρόταση C είναι η:

$$(a_1,\ a_2,\ \ldots\ ,\ a_{i-1},\ a_{i+1},\ \ldots\ ,\ a_k,\ d_1,\ d_2,\ \ldots\ ,\ d_l \leftarrow$$
$$b_1,\ b_2,\ \ldots\ ,\ b_n,\ e_1,\ e_2,\ \ldots\ ,\ e_{j-1},\ e_{j+1},\ \ldots\ ,\ e_m)\sigma$$

ή, με έναν καθιερωμένο συμβολισμό, η:

Αρχή της ανάλυσης

Αν $\sigma = mgu(a_i, e_j)$

$$\frac{\begin{array}{rcl} a_1,\ a_2,\ \ldots\ ,\ a_k & \leftarrow & b_1,\ b_2,\ \ldots\ ,\ b_n \\ d_1,\ d_2,\ \ldots\ ,\ d_l & \leftarrow & e_1,\ e_2,\ \ldots\ ,\ e_m \end{array}}{\begin{array}{rcl} (a_1,\ a_2,\ \ldots\ ,\ a_{i-1},\ a_{i+1},\ \ldots\ ,\ a_k,\ d_1,\ d_2,\ \ldots\ ,\ d_l & \leftarrow & \\ b_1,\ b_2,\ \ldots\ ,\ b_n,\ e_1,\ e_2,\ \ldots\ ,\ e_{j-1},\ e_{j+1},\ \ldots\ ,\ e_m)\sigma & & \end{array}}$$

Παράδειγμα 7.11
Εδώ έχουμε μια περίπτωση εφαρμογής της αρχής της ανάλυσης:

$$\frac{\begin{array}{rcl} healthy(x),\ wealthy(x) & \leftarrow & happy(x) \\ & \leftarrow & wealthy(jim) \end{array}}{\begin{array}{rcl} healthy(jim) & \leftarrow & happy(jim) \end{array}}$$

Απλώς, απαλείψαμε το άτομο $wealthy(x)$ από την κεφαλή της πρώτης πρότασης και το άτομο $wealthy(jim)$ από το σώμα της δεύτερης πρότασης, εφαρμόζοντας και τον γενικότερο ενοποιητή τους $\sigma = \{x/jim\}$ στην πρόταση που προέκυψε με τα άτομα τα οποία απέμειναν. □

Όταν εφαρμόζεται η αρχή της ανάλυσης σε προτάσεις Horn, έχουμε μια ειδική μορφή της, τον κανόνα συμπερασμού **modus ponens**. Έτσι, η διατύπωση της αρχής της ανάλυσης, για $k = 1$ και $l = 1$, γίνεται:

Modus ponens

Αν $\sigma = mgu(a, e_j)$

$$\frac{\begin{array}{rcl} a & \leftarrow & b_1,\ b_2,\ \ldots\ ,\ b_n \\ d & \leftarrow & e_1,\ e_2,\ \ldots\ ,\ e_m \end{array}}{\begin{array}{rcl} (d & \leftarrow & b_1,\ b_2,\ \ldots\ ,\ b_n,\ e_1,\ e_2,\ \ldots\ ,\ e_{j-1},\ e_{j+1},\ \ldots\ ,\ e_m)\sigma \end{array}}$$

Παράδειγμα 7.12
Εδώ έχουμε μια περίπτωση εφαρμογής του modus ponens:

$$\frac{\begin{array}{rcl} parent(john, nick) & \leftarrow & \\ grandparent(x, z) & \leftarrow & parent(x, y),\ parent(y, z) \end{array}}{\begin{array}{rcl} grandparent(john, z) & \leftarrow & parent(nick, z) \end{array}}$$

Απαλείψαμε το μοναδικό άτομο της κεφαλής της πρώτης πρότασης και το πρώτο άτομο $parent(x, y)$ του σώματος της δεύτερης πρότασης, και στην πρόταση που προέκυψε εφαρμόσαμε τον γενικότερο ενοποιητή $\sigma = \{x/john,\ y/nick\}$.

Εναλλακτικά, θα μπορούσαμε να συνδυάσουμε το $parent(john, nick)$, αλλά τώρα με το δεύτερο άτομο του σώματος της δεύτερης πρότασης $parent(y, z)$. Οπότε, ο γενικότερος

ενοποιητής τους θα ήταν ο $\sigma = \{y/john,\ z/nick\}$, και θα είχαμε:

$$\begin{array}{rcl} parent(john, nick) & \leftarrow & \\ grandparent(x, z) & \leftarrow & parent(x, y),\ parent(y, z) \\ \hline grandparent(x, nick) & \leftarrow & parent(x, john) \end{array}$$

Εν γένει, όταν έχουμε ένα σύνολο από προτάσεις στη λογική πρώτης τάξης, οι πιθανές εφαρμογές ενός κανόνα συμπερασμού μπορούν να είναι πάρα πολλές. □

Όπως θα δούμε στην Ενότητα 8.3, στον λογικό προγραμματισμό, οι απαντήσεις ερωτήσεων που υποβάλλονται σε οριστικά προγράμματα υπολογίζονται με τη βοήθεια μιας εξειδίκευσης του κανόνα modus ponens, που είναι ο κανόνας συμπερασμού της **SLD-ανάλυσης**, ο οποίος εφαρμόζεται σε μια οριστική πρόταση και σε έναν οριστικό στόχο, δηλαδή σε προτάσεις Horn. Ειδικότερα, στη γλώσσα προγραμματισμού Prolog, η SLD-ανάλυση εφαρμόζεται με τέτοιον τρόπο ώστε το άτομο του οριστικού στόχου e_j που απαλείφεται με την κεφαλή a της οριστικής πρότασης να είναι πάντοτε το αριστερότερο άτομο του στόχου, δηλαδή να έχουμε $j = 1$.

Άσκηση 7.4
Δίνονται οι εξής προτάσεις λογικής πρώτης τάξης:

$$\begin{array}{rcll} pink(sam) & \leftarrow & & (7.1) \\ & \leftarrow & grey(x_1),\ pink(y_1),\ likes(x_1, y_1) & (7.2) \\ likes(oscar, sam) & \leftarrow & & (7.3) \\ pink(oscar),\ grey(oscar) & \leftarrow & & (7.4) \\ & \leftarrow & grey(x_2),\ pink(y_2),\ likes(x_2, y_2) & (7.5) \\ likes(clyde, oscar) & \leftarrow & & (7.6) \\ grey(clyde) & \leftarrow & & (7.7) \end{array}$$

Μπορείτε να προχωρήσετε σε μια σειρά από εφαρμογές της αρχής της ανάλυσης μεταξύ αυτών των προτάσεων, αλλά και όσων θα παραγάγετε από τις εφαρμογές αυτές, έτσι ώστε να προκύψει τελικά η κενή πρόταση;

Απαντήσεις ασκήσεων

Απάντηση άσκησης 7.1
Το σύνολο των συναρτησιακών συμβόλων για τους τύπους της εκφώνησης είναι το $F = \{greece/0,\ john/0,\ mary/0,\ ./2,\ []/0,\ a/0,\ b/0\}$ και το σύνολο των κατηγορημάτων είναι το $P = \{borders/2,\ man/1,\ woman/1,\ loves/2,\ may\_steal/2,\ likes/2,\ thief/1,\ connected/2,\ on\_diet/1,\ athlete/1,\ fit/1,\ father/2,\ male/1,\ parent/2,\ append/3\}$.

Κλειστοί τύποι είναι οι 2, 3, 5, 6, 7 και 8. Δεν είναι κλειστοί οι 1 και 4, ο 1 γιατί περιέχει την ελεύθερη εμφάνιση μεταβλητής x και ο 4 γιατί έχει όλες τις μεταβλητές του (x, y και z) σε ελεύθερη εμφάνιση. Προσέξτε, οι τύποι 5 και 7 είναι κλειστοί, παρότι, με πρώτη ματιά, ίσως να φαίνεται ότι έχουν ελεύθερες εμφανίσεις μεταβλητών. Δεν είναι όμως έτσι. Απλώς, είναι προτάσεις γραμμένες στη συντομευμένη μορφή τους, όπου δεν αναφέρονται οι καθολικοί ποσοδείκτες για τις μεταβλητές, όμως ουσιαστικά υπάρχουν, και όλες οι εμφανίσεις μεταβλητών είναι δεσμευμένες. Συνεπώς, οι τύποι είναι κλειστοί.

Οι τύποι 3, 6 και 8 είναι προτάσεις διατυπωμένες με τον κλασικό τρόπο της λογικής πρώτης τάξης (διάζευξη στοιχειωδών τύπων με καθολική ποσοτικοποίηση στις μεταβλητές τους, εφόσον υπάρχουν). Με τον εναλλακτικό απλουστευμένο τρόπο, θα μπορούσαν να διατυπωθούν, αντίστοιχα, ως εξής:

$$may\_steal(x, y) \leftarrow likes(x, y),\ thief(x)$$
$$\leftarrow loves(john, mary),\ loves(mary, john)$$
$$append(.(a, []), .(b, []), .(a, .(b, []))) \leftarrow$$

Οι τύποι 5 και 7 είναι προτάσεις ήδη διατυπωμένες με τον εναλλακτικό τρόπο.

Από τους τύπους 3, 5, 6, 7 και 8, που είναι προτάσεις, διατυπωμένες είτε με τον κλασικό τρόπο είτε με τον εναλλακτικό, μόνο οι 3, 6, 7 και 8 είναι προτάσεις Horn. Η 5 δεν είναι πρόταση Horn, γιατί στην κεφαλή της έχει δύο άτομα.

Από τους τύπους 3, 6, 7 και 8, που είναι προτάσεις Horn, οι 3, 7 και 8 είναι οριστικές προτάσεις, ενώ η 6 είναι οριστικός στόχος. Η 8 είναι μοναδιαία οριστική πρόταση.

Απάντηση άσκησης 7.2

Αν η γλώσσα πρώτης τάξης για την οποία συζητάμε είναι το σύνολο L, τότε το σύνολο των βασικών όρων είναι το:

$$U_L = \{liz,\ mother\_of(liz),\ mother\_of(mother\_of(liz)),\ \ldots\}$$

ενώ το σύνολο των βασικών ατόμων είναι το:

$$B_L = \{alive(liz),\ alive(mother\_of(liz)),\ alive(mother\_of(mother\_of(liz))),\ \ldots\}$$

Στη γλώσσα L περιλαμβάνονται, μεταξύ άλλων, οι εξής τύποι:

$$alive(mother\_of(liz))$$
$$(\exists x)(alive(mother\_of(mother\_of(x))))$$
$$(\forall y)(alive(mother\_of(y)) \rightarrow alive(y))$$
$$(\exists x)(\exists y)(\neg alive(x) \vee \neg alive(mother\_of(x)))$$
$$(\forall x)(alive(x)) \rightarrow alive(liz)$$

Απάντηση άσκησης 7.3

Οι γενικότεροι ενοποιητές των όρων E και F, που δόθηκαν στην εκφώνηση της άσκησης, φαίνονται στη συνέχεια:

	$mgu(E, F)$
1	$\{u/j(x, y),\ v/j(x, y),\ w/j(x, y)\}$
2	$\nexists\, mgu$
3	$\{u/a,\ x/f(b, a),\ y/b,\ z/a\}$
4	$\{x_1/g(a, a),\ x_2/g(a, a),\ x_3/g(a, a),\ x_4/a,\ x_5/a,\ x_6/a,\ x_7/a,\ x_8/a\}$
5	$\{x_1/g(m(a, b), m(a, b)),\ x_2/m(a, b),\ x_3/m(a, b),\ x_4/b,\ x_5/b\}$
6	$\{v/g(u, f(a, b)),\ x/u,\ y/f(a, b),\ z/f(u, g(u, f(a, b)))\}$

Ο λόγος για τον οποίο δεν υπάρχει γενικότερος ενοποιητής στην περίπτωση 2 είναι ο έλεγχος εμφάνισης. Συγκεκριμένα, η ενοποίηση αποτυγχάνει, επειδή γίνεται απόπειρα να ενοποιηθούν το w με το $j(x, w)$.

Απάντηση άσκησης 7.4
Ο απλούστερος τρόπος να προκύψει τελικά η κενή πρόταση, έπειτα από διαδοχικές εφαρμογές της αρχής της ανάλυσης στις προτάσεις που δόθηκαν, είναι να συνδυάσουμε τις (7.1) και (7.2), το αποτέλεσμα με την (7.3), το αποτέλεσμα με την (7.4) κ.ο.κ. Έτσι, θα έχουμε:

$$
\begin{array}{lllll}
(7.1) & \overset{(7.2)}{\Rightarrow} & \leftarrow & grey(x_1),\ likes(x_1, sam) & \sigma_1 = \{y_1/sam\} \\
& \overset{(7.3)}{\Rightarrow} & \leftarrow & grey(oscar) & \sigma_2 = \{x_1/oscar\} \\
& \overset{(7.4)}{\Rightarrow} \quad pink(oscar) & \leftarrow & & \sigma_3 = \{\ \} \\
& \overset{(7.5)}{\Rightarrow} & \leftarrow & grey(x_2),\ likes(x_2, oscar) & \sigma_4 = \{y_2/oscar\} \\
& \overset{(7.6)}{\Rightarrow} & \leftarrow & grey(clyde) & \sigma_5 = \{x_2/clyde\} \\
& \overset{(7.7)}{\Rightarrow} & \leftarrow & & \sigma_6 = \{\ \}
\end{array}
$$

Προβλήματα

Πρόβλημα 7.1
Διατυπώστε σε λογική πρώτης τάξης πέντε από τους κανόνες Prolog που αναφέρονται στην Ενότητα 3.1, είτε στα παραδείγματα είτε στις ασκήσεις.

Πρόβλημα 7.2
Γιατί πιστεύετε ότι στον αλγόριθμο ενοποίησης δεν επιτρέπεται να γίνει δέσμευση μεταβλητής σε μια τιμή που περιέχει τη μεταβλητή αυτή;

Πρόβλημα 7.3
Δίνονται οι εξής προτάσεις λογικής πρώτης τάξης:

$$
\begin{array}{rll}
append([], L, L) & \leftarrow & \\
append(.(X, L1), L2, .(X, L3)) & \leftarrow & append(L1, L2, L3) \\
& \leftarrow & append(.(a, .(b, .(c, []))), .(d, []), .(a, .(b, .(c, .(d, [])))))
\end{array}
$$

Μπορείτε να προχωρήσετε σε μια σειρά από εφαρμογές της αρχής της ανάλυσης μεταξύ αυτών των προτάσεων, αλλά και όσων θα παραγάγετε από τις εφαρμογές αυτές, έτσι ώστε να προκύψει τελικά η κενή πρόταση;

Βιβλιογραφικές αναφορές

[1] Γ. Μητακίδης, *Απο τη Λογική στο Λογικό Προγραμματισμό και την Prolog*, Καρδαμίτσας, 1992.

[2] K. Doets, *From Logic to Logic Programming*, The MIT Press, 1994.

[3] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1993.

[4] J. A. Robinson, Logic and Logic Programming, *CACM*, pp. 35(3), 40-65, 1992.

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

Κεφάλαιο 8
Σημασιολογία λογικών προγραμμάτων

Σύνοψη

Στο κεφάλαιο αυτό παρουσιάζεται η μοντελοθεωρητική σημασιολογία του λογικού προγραμματισμού, δηλαδή αυτή που βασίζεται σε ερμηνείες και μοντέλα, με τελικό στόχο τη μελέτη της σημασίας των οριστικών προγραμμάτων. Επίσης, γίνεται αναφορά στη σημασιολογία σταθερού σημείου και στον τελεστή άμεσου επακόλουθου, αλλά και στη λειτουργική σημασιολογία, μέσω της εφαρμογής ενός κανόνα συμπερασμού, της SLD-ανάλυσης.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει μελετήσει το Κεφάλαιο 7.

8.1 Μοντελοθεωρητική σημασιολογία

Εκτός από τη συντακτική πλευρά των οριστικών (ή λογικών) προγραμμάτων στη λογική πρώτης τάξης, που ουσιαστικά είδαμε στην Ενότητα 7.1, μας ενδιαφέρει να μπορούμε με κάποιον τρόπο να περιγράφουμε τι σημαίνουν τα προγράμματα αυτά, δηλαδή ποιες καταστάσεις του περιβάλλοντος κόσμου περιγράφουν [1, 2]. Για το σκοπό αυτό, υπάρχουν διάφορες μεθοδολογίες μελέτης της σημασίας οριστικών προγραμμάτων. Σε αυτήν την ενότητα θα περιγράψουμε τη **μοντελοθεωρητική σημασιολογία** [3, 4, 5, 6]. Πριν προχωρήσουμε όμως στην ουσία, είναι απαραίτητο να δώσουμε κάποιους ορισμούς.

Ορισμός 8.1
Δεδομένης κάποιας γλώσσας πρώτης τάξης L, το σύνολο των βασικών όρων της γλώσσας, δηλαδή το σύνολο των όρων χωρίς μεταβλητές που μπορούν να κατασκευαστούν από τα συναρτησιακά σύμβολα της γλώσσας (συμπεριλαμβανομένων των σταθερών), ονομάζεται **σύμπαν Herbrand** και συμβολίζεται με U_L. Αν το σύνολο των συναρτησιακών συμβόλων δεν περιλαμβάνει καμία σταθερά, εισάγουμε αυθαίρετα κάποια, έτσι ώστε το U_L να μην είναι το κενό σύνολο.

Ορισμός 8.2
Δεδομένης κάποιας γλώσσας πρώτης τάξης L, το σύνολο των βασικών ατόμων της γλώσσας, δηλαδή το σύνολο των ατόμων με κατηγορήματα από αυτά της γλώσσας και ορίσματα από το U_L, ονομάζεται **βάση Herbrand** και συμβολίζεται με B_L. □

Στην Άσκηση 7.2 είδαμε ένα παράδειγμα μιας γλώσσας πρώτης τάξης και τα αντίστοιχα σύνολα U_L και B_L. Ας δούμε άλλο ένα:

Παράδειγμα 8.1
Αν $F = \{a/0,\ f/1,\ g/1\}$ και $P = \{p/0,\ q/1,\ r/2\}$, τότε:

$$U_L = \{a,\ f(a),\ g(a),\ f(f(a)),\ f(g(a)),\ g(f(a)),\ g(g(a)),\ f(f(f(a))),\ \ldots\}$$

και:

$$B_L = \{p,\ q(a),\ q(f(a)),\ q(g(a)),\ q(f(f(a))),\ \ldots,\ r(a,a),\ r(a,f(a)),\ \ldots\}$$

Αν σε μια γλώσσα πρώτης τάξης L, το F περιέχει έστω και ένα συναρτησιακό σύμβολο με βαθμό διάφορο του 0, τότε τόσο το U_L, όσο και το B_L είναι απειροσύνολα. Αν το F περιέχει μόνο σταθερές, τότε τα U_L και B_L είναι πεπερασμένα σύνολα.

Ορισμός 8.3
Ένα υποσύνολο I της βάσης Herbrand B_L ($I \subseteq B_L$) ονομάζεται **ερμηνεία Herbrand** ή, στο εξής, απλώς **ερμηνεία**. □

Ουσιαστικά, μια ερμηνεία αντιστοιχεί σε κάθε κατηγόρημα p/n της L μια σχέση επάνω στο ${U_L}^n$. Άτυπα, μια ερμηνεία περιγράφει μια υποψήφια κατάσταση του περιβάλλοντος κόσμου, με την έννοια ότι περιέχει εκείνα τα βασικά άτομα που αντιστοιχούν σε ό,τι ισχύει στην κατάσταση αυτή του κόσμου.

Παράδειγμα 8.2
Αν $U_L = \{a,\ b,\ c\}$ και $B_L = \{p(a),\ p(b),\ p(c),\ q(a),\ q(b),\ q(c),\ r(a),\ r(b),\ r(c)\}$, τότε μια ερμηνεία είναι η $I = \{p(a),\ p(c),\ q(c),\ r(b)\}$. Σε έναν κόσμο στον οποίο υπάρχουν τρεις οντότητες, αυτές που παριστάνονται από τις σταθερές a, b και c, και τρεις υποψήφιες ιδιότητες για τις οντότητες, αυτές που περιγράφονται με τα κατηγορήματα p, q και r, η ερμηνεία I οριοθετεί μια κατάσταση του κόσμου στην οποία οι οντότητες a και c έχουν την ιδιότητα p, η c έχει την ιδιότητα q και η b έχει την ιδιότητα r. □

Η βασική ιδέα για τη μοντελοθεωρητική μελέτη της σημασίας των οριστικών προγραμμάτων είναι αυτή της αλήθειας ενός κλειστού τύπου σε σχέση με κάποια ερμηνεία. Ο σχετικός ορισμός ακολουθεί.

Ορισμός 8.4
Ένας κλειστός τύπος είναι **αληθής** σε μια ερμηνεία I αν:

- είναι ένα βασικό άτομο A και ισχύει $A \in I$,
- είναι της μορφής $\neg F$ και ο τύπος F δεν είναι αληθής στην I,
- είναι της μορφής $F \wedge G$ και οι τύποι F και G είναι αληθείς στην I,
- είναι της μορφής $F \vee G$ και κάποιος από τους τύπους F ή G είναι αληθής στην I,
- είναι της μορφής $F \rightarrow G$ (ή $G \leftarrow F$) και είτε ο τύπος F δεν είναι αληθής στην I είτε ο G είναι αληθής στην I,
- είναι της μορφής $F \leftrightarrow G$ και είτε και οι δύο τύποι (F και G) είναι αληθείς στην I είτε και οι δύο δεν είναι αληθείς στην I,
- είναι της μορφής $(\forall x)(F)$ και για όλα τα $d \in U_L$, αν αντικατασταθεί η μεταβλητή x στον τύπο F με το d, οι τύποι που προκύπτουν να είναι όλοι αληθείς στην I,

- είναι της μορφής $(\exists x)(F)$ και υπάρχει κάποιο $d \in U_L$ τέτοιο ώστε, αν αντικατασταθεί η μεταβλητή x στον τύπο F με το d, ο τύπος που προκύπτει να είναι αληθής στην I.

Ορισμός 8.5
Ένας κλειστός τύπος είναι **ψευδής** σε μια ερμηνεία I, αν δεν είναι αληθής στην I. □

Οι ανοικτοί τύποι δεν μπορούν να χαρακτηριστούν αληθείς ή ψευδείς σε σχέση με μια ερμηνεία, εκτός αν έχουμε και μια αντικατάσταση των ελεύθερων εμφανίσεων μεταβλητών τους από στοιχεία του σύμπαντος Herbrand. Οπότε, γίνονται πλέον κλειστοί.

Ορισμός 8.6
Δύο τύποι που είναι αληθείς ακριβώς στις ίδιες ερμηνείες ονομάζονται **ισοδύναμοι**.

Παράδειγμα 8.3
Οι τύποι $\neg F \vee G$ και $F \rightarrow G$ είναι ισοδύναμοι. Πράγματι, αυτό προκύπτει αμέσως από τον ορισμό 8.4, και συγκεκριμένα από τις περιπτώσεις για την άρνηση, τη διάζευξη και τη συνεπαγωγή. Επίσης, από τον ίδιο ορισμό μπορούμε πολύ εύκολα να βρούμε και άλλα ενδιαφέροντα ζευγάρια ισοδύναμων τύπων, όπως τα:

- $\neg(F \wedge G)$ και $\neg F \vee \neg G$
- $\neg(F \vee G)$ και $\neg F \wedge \neg G$
- $\neg(\exists x)(F)$ και $(\forall x)(\neg F)$
- $\neg(\forall x)(F)$ και $(\exists x)(\neg F)$

Ορισμός 8.7
Μια ερμηνεία στην οποία ένας τύπος είναι αληθής ονομάζεται **μοντέλο Herbrand** του τύπου ή, στο εξής, απλώς **μοντέλο** του τύπου. Μια ερμηνεία είναι μοντέλο ενός συνόλου από κλειστούς τύπους, αν είναι μοντέλο για κάθε τύπο του συνόλου.

Ορισμός 8.8
Αν δεν υπάρχει κάποιο μοντέλο για ένα σύνολο από κλειστούς τύπους, το σύνολο αυτό ονομάζεται **ασυνεπές**. Αν έχει έστω και ένα μοντέλο, ονομάζεται **συνεπές**.

Ορισμός 8.9
Αν S είναι ένα σύνολο από κλειστούς τύπους και F είναι ένας κλειστός τύπος, τότε το F είναι **λογικό επακόλουθο** του S (συμβολικά $S \models F$), αν κάθε μοντέλο του S είναι και μοντέλο του F. □

Ας εξετάσουμε όμως για λίγο τη φυσική σημασία του ορισμού 8.9. Αν όλα τα μοντέλα του S είναι και μοντέλα του F, τότε, σε οποιαδήποτε κατάσταση του κόσμου στην οποία το σύνολο τύπων S είναι αληθές, πρέπει οπωσδήποτε και ο τύπος F να είναι αληθής. Με άλλα λόγια, η ισχύς του S επιβάλλει πάντα και την ισχύ του F. Για αυτόν το λόγο, το F χαρακτηρίζεται λογικό επακόλουθο του S. Προσέξτε όμως ότι μπορεί να υπάρχουν καταστάσεις του κόσμου, δηλαδή ερμηνείες, που να είναι μοντέλα του F, αλλά όχι του S. Δηλαδή, η αλήθεια

του F δεν επιβάλλει την αλήθεια του S. Ουσιαστικά, τα μοντέλα του F είναι υπερσύνολο των μοντέλων του S, όταν το F είναι λογικό επακόλουθο του S. Οπότε, αν ο κόσμος μας αντιπροσωπεύεται από μια ερμηνεία μέσα από τη διαφορά των συνόλων των μοντέλων, τότε το F ισχύει, αλλά όχι και το S. Επομένως, η σχέση του λογικού επακόλουθου δεν είναι αντιμεταθετική.

Σχετικά με την έννοια του λογικού επακόλουθου, ισχύει η εξής ενδιαφέρουσα πρόταση:

Πρόταση 8.1
Αν S είναι ένα σύνολο από κλειστούς τύπους και F ένας κλειστός τύπος, τότε ισχύει ότι το F είναι λογικό επακόλουθο του S ($S \models F$), αν και μόνο αν το σύνολο τύπων $S \cup \{\neg F\}$ είναι ασυνεπές.
Απόδειξη:
Έστω $S \models F$. Έστω επίσης ότι το $S \cup \{\neg F\}$ είναι συνεπές, δηλαδή υπάρχει μια ερμηνεία I που είναι μοντέλο του. Συνεπώς, η I είναι μοντέλο του S, αλλά και του $\neg F$. Άρα, η I δεν είναι μοντέλο του F. Έχουμε τώρα μια ερμηνεία (την I) που είναι μοντέλο του S, αλλά όχι του F. Αυτό όμως είναι άτοπο, αφού $S \models F$. Άρα, το $S \cup \{\neg F\}$ είναι ασυνεπές.
Αντίστροφα τώρα: Έστω ότι το $S \cup \{\neg F\}$ είναι ασυνεπές. Έστω επίσης ότι δεν ισχύει το $S \models F$, δηλαδή ότι υπάρχει μια ερμηνεία I που είναι μοντέλο του S, αλλά όχι του F. Τότε η I είναι μοντέλο του $\neg F$, άρα και του $S \cup \{\neg F\}$. Αυτό όμως είναι άτοπο, γιατί, σύμφωνα με την υπόθεσή μας, το $S \cup \{\neg F\}$ είναι ασυνεπές. Επομένως, $S \models F$. □

Η πρόταση 8.1 παρουσιάζει ενδιαφέρον, γιατί συσχετίζει την έννοια του λογικού επακόλουθου ή, κατά μια άλλη ορολογία, του «συμπεραίνειν», με την έννοια της ασυνέπειας.

Ας θυμηθούμε στο σημείο αυτό, από την Ενότητα 7.1, τις προτάσεις, που είναι ειδικές περιπτώσεις κλειστών τύπων της μορφής:

$$(\forall x_1)(\forall x_2)\ \ldots\ (\forall x_s)(A_1 \vee A_2 \vee \ldots \vee A_k \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n)$$

όπου τα A_i ($1 \leq i \leq k$) και B_j ($1 \leq j \leq n$) είναι άτομα. Επίσης, γνωρίζουμε ότι αυτή η πρόταση έχει και μια απλούστερη εναλλακτική γραφή, την:

$$A_1,\ A_2,\ \ldots\ ,\ A_k \leftarrow B_1,\ B_2,\ \ldots\ ,\ B_n$$

όπου τα άτομα που είναι αριστερά του $\leftarrow$ αποτελούν την κεφαλή της πρότασης, ενώ αυτά που είναι δεξιά το σώμα της.

Εφαρμόζοντας τους ορισμούς 8.4 και 8.7 στην περίπτωση των προτάσεων, καταλαβαίνουμε ότι μια ερμηνεία I είναι μοντέλο για μια πρόταση, αν για κάθε δυνατή αντικατάσταση των μεταβλητών της πρότασης με στοιχεία από το U_L, η πρόταση που προκύπτει είναι αληθής στην I, δηλαδή είτε κάποιο από τα βασικά (πλέον) άτομα της κεφαλής ανήκει (είναι αληθές) στην I είτε κάποιο από τα βασικά άτομα του σώματος δεν ανήκει (είναι ψευδές) στην I. Δηλαδή, αν έχουμε μια ερμηνεία I τέτοια ώστε όλα τα άτομα του σώματος της πρότασης να είναι αληθή στην I, για να είναι αυτή η ερμηνεία μοντέλο της πρότασης, θα πρέπει κάποιο από τα άτομα της κεφαλής να είναι αληθές στην I. Για σκεφτείτε το όμως λίγο. Μήπως αυτό έχει τη μορφή μιας συνεπαγωγής, με υπόθεση μια σύζευξη και συμπέρασμα μια διάζευξη;

Αν θεωρήσουμε τη συνεπαγωγή την οποία αναφέραμε προηγουμένως και πάρουμε την περίπτωση που στο συμπέρασμα η διάζευξη έχει ένα ακριβώς άτομο, έχουμε την περίπτωση των οριστικών προτάσεων. Για τα οριστικά προγράμματα, δηλαδή τα σύνολα από οριστικές προτάσεις, ισχύει η εξής πρόταση:

Πρόταση 8.2
Αν P είναι ένα οριστικό πρόγραμμα, τότε η τομή δύο μοντέλων του είναι επίσης μοντέλο του P.
Απόδειξη:
Ας πάρουμε δύο ερμηνείες, I_1 και I_2, που είναι μοντέλα του P. Έστω ότι η ερμηνεία $I_1 \cap I_2$ δεν είναι μοντέλο του P. Άρα, δεν είναι μοντέλο για κάποια τουλάχιστον πρόταση του P, έστω την C. Προκειμένου να συμβαίνει αυτό, για κάποια αποτίμηση των μεταβλητών της C από το σύμπαν Herbrand, πρέπει η πρόταση που προκύπτει να είναι τέτοια ώστε όλα τα άτομα του σώματός της να ανήκουν στην $I_1 \cap I_2$, αλλά να μην ανήκει εκεί το άτομο της κεφαλής της. Άρα, τα άτομα του σώματος ανήκουν και στην I_1 και στην I_2, και επειδή οι ερμηνείες αυτές είναι μοντέλα του P, άρα και της C, θα πρέπει και το άτομο της κεφαλής να ανήκει και στην I_1 και στην I_2, άρα και στην $I_1 \cap I_2$. Αυτό όμως, όπως είδαμε, δεν μπορεί να συμβαίνει. Άρα, το ότι η $I_1 \cap I_2$ δεν είναι μοντέλο του P είναι άτοπο και, συνεπώς, ισχύει το αποδεικτέο.

Ορισμός 8.10
Η τομή M_P όλων των μοντέλων ενός οριστικού προγράμματος ονομάζεται **ελάχιστο μοντέλο Herbrand** ή, απλώς, **ελάχιστο μοντέλο**, του P.

Θεώρημα 8.1
Αν P είναι ένα οριστικό πρόγραμμα σε μιας γλώσσα πρώτης τάξης L, τότε το ελάχιστο μοντέλο του P αποτελείται από όλα τα στοιχεία του B_L που είναι λογικά επακόλουθα του P. Δηλαδή:

$$M_P = \{A \in B_L \mid P \models A\}$$

Απόδειξη:
Έστω $A \in M_P$, δηλαδή το A ανήκει σε όλα τα μοντέλα του P. Άρα, όλα τα μοντέλα του P είναι και μοντέλα του A. Συνεπώς, $P \models A$, δηλαδή $A \in \{A \in B_L \mid P \models A\}$. Οπότε, ισχύει ότι $M_P \subseteq \{A \in B_L \mid P \models A\}$.
Αντιστρόφως: Έστω $A \in \{A \in B_L \mid P \models A\}$, δηλαδή $P \models A$. Άρα, κάθε μοντέλο του P, συνεπώς και το M_P, είναι μοντέλο του A. Δηλαδή, $A \in M_P$. Οπότε, έχουμε επίσης ότι $\{A \in B_L \mid P \models A\} \subseteq M_P$.
Συνεπώς, ισχύει $M_P = \{A \in B_L \mid P \models A\}$. □

Το θεώρημα 8.1 είναι το συμπέρασμα της μοντελοθεωρητικής σημασιολογίας για τα οριστικά προγράμματα. Όταν διατυπώνουμε ένα οριστικό πρόγραμμα, έχουμε την πρόθεση να μοντελοποιήσουμε τον ελάχιστο κόσμο στον οποίο οι προτάσεις του προγράμματός μας είναι αληθείς. Ο κόσμος αυτός είναι το ελάχιστο μοντέλο του προγράμματος, το οποίο περιλαμβάνει ακριβώς όλα τα βασικά άτομα που είναι λογικά επακόλουθα του προγράμματος.

Παράδειγμα 8.4
Έστω το οριστικό πρόγραμμα P:

$$\begin{array}{rcl} p(x) & \leftarrow & q(x) \\ q(x) & \leftarrow & r(x) \\ p(a) & \leftarrow & \\ q(b) & \leftarrow & \\ r(c) & \leftarrow & \end{array}$$

Τότε, έχουμε:

$$U_L = \{a,\ b,\ c\}$$

και:

$$B_L = \{p(a),\ p(b),\ p(c),\ q(a),\ q(b),\ q(c),\ r(a),\ r(b),\ r(c)\}$$

Το ελάχιστο μοντέλο M_P του P είναι:

$$M_P = \{p(a),\ p(b),\ p(c),\ q(b),\ q(c),\ r(c)\}$$

Μπορείτε να το επιβεβαιώσετε αυτό, αν απαριθμήσετε όλα τα μοντέλα του P (εκείνες από τις $2^{|B_L|} = 512$ δυνατές ερμηνείες στις οποίες όλες οι προτάσεις του P είναι αληθείς) και να πάρετε το ελάχιστο από αυτά. Είναι δυνατόν όμως να βρείτε έτσι το ελάχιστο μοντέλο ενός οριστικού προγράμματος; Προφανώς όχι. Έναν κατασκευαστικό τρόπο θα δούμε στη μελέτη της σημασίας των οριστικών προγραμμάτων μέσω σταθερού σημείου, στην Ενότητα 8.2.

Άσκηση 8.1
Έστω το λογικό πρόγραμμα P:

$$myplus(x, 0, x) \leftarrow \tag{8.1}$$
$$myplus(x, s(y), s(z)) \leftarrow myplus(x, y, z) \tag{8.2}$$

Αν τα σύμβολα x, y και z είναι μεταβλητές, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Είναι δυνατόν ένα μοντέλο του P να περιέχει το βασικό άτομο $myplus(0, 0, s(0))$;

Άσκηση 8.2
Έστω το λογικό πρόγραμμα P:

$$p(x) \leftarrow q(x),\ r(x) \tag{8.3}$$
$$r(x) \leftarrow s(x) \tag{8.4}$$
$$p(x) \leftarrow s(x) \tag{8.5}$$
$$q(a) \leftarrow \tag{8.6}$$
$$q(b) \leftarrow \tag{8.7}$$
$$r(b) \leftarrow \tag{8.8}$$
$$r(c) \leftarrow \tag{8.9}$$
$$s(d) \leftarrow \tag{8.10}$$

Αν το σύμβολο x είναι μεταβλητή, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Γιατί οι ερμηνείες:

$$\begin{aligned}
I_1 &= \{p(d),\ q(a),\ q(b),\ r(b),\ r(c),\ s(d)\} \\
I_2 &= \{p(b),\ p(d),\ q(a),\ q(b),\ r(b),\ r(d),\ s(d)\} \\
I_3 &= \{q(a),\ q(b),\ r(c),\ r(d),\ s(d)\} \\
I_4 &= \{p(b),\ p(d),\ q(a),\ q(b),\ q(c),\ r(b),\ r(c),\ r(d),\ s(d)\} \\
I_5 &= \{p(b),\ q(a),\ q(b),\ r(b),\ r(c),\ r(d),\ s(a),\ s(d)\}
\end{aligned}$$

δεν είναι μοντέλα για το P; Βλέπετε κάποιο λόγο για τον οποίο η ερμηνεία:

$$I_6 = \{p(b),\ p(d),\ q(a),\ q(b),\ r(b),\ r(c),\ r(d),\ s(d)\}$$

δεν θα μπορούσε να ήταν το ελάχιστο μοντέλο του P;

Άσκηση 8.3
Γνωρίζετε ότι η τομή δύο μοντέλων ενός οριστικού προγράμματος είναι επίσης μοντέλο του (Πρόταση 8.2). Δείξτε, μέσω ενός αντιπαραδείγματος, ότι η πρόταση αυτή δεν ισχύει για προγράμματα που δεν είναι οριστικά.

8.2 Σημασιολογία σταθερού σημείου

Στην ενότητα αυτή θα μελετήσουμε τη δεύτερη μεθοδολογία απόδοσης σημασίας σε οριστικά προγράμματα. Αναφερόμαστε στη **σημασιολογία σταθερού σημείου** [3, 4, 5, 6], που, όπως προαναγγείλαμε, θα μας εφοδιάσει με έναν τρόπο κατασκευής του ελάχιστου μοντέλου ενός οριστικού προγράμματος. Ως συνήθως, θα αρχίσουμε με κάποιους ορισμούς.[1]

Ορισμός 8.11
Ένα σύνολο L εφοδιασμένο με μια σχέση μερικής διάταξης λέγεται ότι είναι **πλήρες πλέγμα**, αν κάθε υποσύνολό του X έχει ελάχιστο άνω φράγμα $lub(X)$ και μέγιστο κάτω φράγμα $glb(X)$. Το $lub(L)$ ονομάζεται **ανώτερο στοιχείο** του L (συμβολικά $\top$) και το $glb(L)$ ονομάζεται **κατώτερο στοιχείο** του L (συμβολικά $\bot$).

Παράδειγμα 8.5
Κλασική περίπτωση πλήρους πλέγματος είναι το δυναμοσύνολο 2^S ενός συνόλου S, με σχέση μερικής διάταξης αυτήν του υποσυνόλου $\subseteq$. Το ανώτερο στοιχείο $\top$ είναι το S και το κατώτερο στοιχείο $\bot$ είναι το κενό σύνολο $\emptyset$.

Ορισμός 8.12
Αν T είναι μια απεικόνιση σε ένα σύνολο L, δηλαδή $T : L \longrightarrow L$, τότε ένα στοιχείο $a \in L$ ονομάζεται **σταθερό σημείο** της T αν ισχύει $T(a) = a$. Επιπλέον, αν το L είναι πλήρες πλέγμα, οπότε υπάρχει ορισμένη και μια σχέση μερικής διάταξης $\leq$ σε αυτό, τότε ένα σταθερό σημείο a της T ονομάζεται **ελάχιστο σταθερό σημείο** (συμβολικά $lfp(T)$), αν, για οποιοδήποτε σταθερό σημείο b της T, ισχύει $a \leq b$.

Ορισμός 8.13
Αν L είναι ένα πλήρες πλέγμα και X είναι ένα υποσύνολό του ($X \subseteq L$), τότε το X είναι **κατευθυνόμενο**, αν κάθε πεπερασμένο υποσύνολο του X έχει άνω φράγμα στο X.

Ορισμός 8.14
Αν L είναι ένα πλήρες πλέγμα και T είναι μια απεικόνιση στο L, δηλαδή $T : L \longrightarrow L$, τότε η T είναι **συνεχής**, αν ισχύει $T(lub(X)) = lub(T(X))$, για κάθε κατευθυνόμενο υποσύνολο X του L. □

Κρίσιμη για τη σημασιολογία σταθερού σημείου των οριστικών προγραμμάτων είναι η

[1] Η παρουσίαση δεν θα είναι εξαντλητική από πλευράς θεωρητικής τεκμηρίωσης και μαθηματικής αυστηρότητας. Θα παραλείψουμε αρκετούς ορισμούς και αποδείξεις, και σε μερικές περιπτώσεις θα παρουσιάσουμε κάποιες έννοιες λίγο άτυπα. Οι αναγνώστες που ενδιαφέρονται για την πλήρη και λεπτομερή παρουσίαση της σημασιολογίας σταθερού σημείου για οριστικά προγράμματα παραπέμπονται στη βιβλιογραφία.

πρόταση που ακολουθεί, την οποία παραθέτουμε χωρίς απόδειξη.

Πρόταση 8.3
Αν μια συνεχής απεικόνιση T είναι ορισμένη σε ένα πλήρες πλέγμα L, τότε η T έχει ένα ελάχιστο σταθερό σημείο $lfp(T)$, το οποίο ισούται με:

$$lfp(T) = lub(\{\bot,\ T(\bot),\ T(T(\bot)),\ T(T(T(\bot))),\ \ldots\}) \qquad \square$$

Ας δούμε τώρα πώς μπορούμε να εφαρμόσουμε τα προηγούμενα για την περίπτωση των οριστικών προγραμμάτων. Δεδομένης μιας γλώσσας πρώτης τάξης L και ενός οριστικού προγράμματος P σε αυτήν, ορίζουμε μια απεικόνιση T_P στο σύνολο των ερμηνειών, δηλαδή $T_P : 2^{B_L} \longrightarrow 2^{B_L}$, που ονομάζεται **τελεστής άμεσου επακόλουθου**, ως εξής:

$$T_P(I) = \{A \in B_L \ \mid \ A \leftarrow A_1,\ A_2,\ \ldots\ ,\ A_n \text{ είναι πρόταση του } P \text{ με τις μεταβλητές της δεσμευμένες με στοιχεία από το } U_L \text{ και } \{A_1,\ A_2,\ \ldots\ ,\ A_n\} \subseteq I\}$$

Αποδεικνύεται ότι η T_P είναι συνεχής. Συνεπώς, αφού το 2^{B_L} είναι πλήρες πλέγμα, η T_P έχει ελάχιστο σταθερό σημείο, το:

$$lfp(T_P) = lub(\{\emptyset,\ T_P(\emptyset),\ T_P(T_P(\emptyset)),\ T_P(T_P(T_P(\emptyset))),\ \ldots\})$$

Το αποτέλεσμα της σημασιολογίας σταθερού σημείου για τα οριστικά προγράμματα είναι το θεώρημα που ακολουθεί, το οποίο παραθέτουμε χωρίς απόδειξη.

Θεώρημα 8.2
Το ελάχιστο μοντέλο M_P ενός οριστικού προγράμματος P είναι το ελάχιστο σταθερό σημείο του τελεστή άμεσου επακόλουθου T_P. Δηλαδή:

$$M_P = lfp(T_P) \qquad \square$$

Έτσι, με το θεώρημα 8.2, σε συνδυασμό και με την πρόταση 8.3, έχουμε ένα «εργαλείο» κατασκευής του ελάχιστου μοντέλου ενός οριστικού προγράμματος. Απλώς, πρέπει να ξεκινήσουμε από το κενό σύνολο, να εφαρμόσουμε επάνω του τον τελεστή άμεσου επακόλουθου T_P, στο αποτέλεσμα να εφαρμόσουμε πάλι τον T_P κ.ο.κ. Αυτή η διαδικασία πρέπει να συνεχιστεί έως ότου το αποτέλεσμα να μην αλλάζει, όσο και να εφαρμόζουμε εμείς τον T_P. Έτσι, το ελάχιστο άνω φράγμα όλων των ενδιάμεσων αποτελεσμάτων που βρήκαμε είναι το ελάχιστο σταθερό σημείο του T_P, το οποίο είναι τελικά το ελάχιστο μοντέλο M_P του P.

Παράδειγμα 8.6
Ας εξετάσουμε πάλι το οριστικό πρόγραμμα που δώσαμε στο Παράδειγμα 8.4 και ας εφαρμόσουμε όσα μάθαμε σε αυτήν την ενότητα, για να υπολογίσουμε το ελάχιστο μοντέλο του προγράμματος. Έχουμε, λοιπόν:

- $\emptyset$: Το ξεκίνημά μας.
- $T_P(\emptyset) =$
  $= \{p(a),\ q(b),\ r(c)\}$: Αυτό είναι το πρώτο βήμα εφαρμογής του T_P. Οπότε, προκύπτει μόνο ό,τι μας δίνουν οι μοναδιαίες προτάσεις του προγράμματος.[2]

[2] Αυτή η περίπτωση καλύπτεται από τον ορισμό του T_P, αλλά για $n = 0$, που κανείς δεν μας απαγόρευσε.

- $T_P(T_P(\emptyset)) =$
  $= T_P(\{p(a),\ q(b),\ r(c)\}) = \{p(a),\ p(b),\ q(b),\ q(c),\ r(c)\}$: Δεδομένου του $q(b)$ και λόγω της πρότασης $p(x) \leftarrow q(x)$, προστίθεται στο αποτέλεσμα και το $p(b)$. Επίσης, λόγω της πρότασης $q(x) \leftarrow r(x)$ και του $r(c)$, προστίθεται και το $q(c)$.
- $T_P(T_P(T_P(\emptyset))) =$
  $= T_P(\{p(a),\ p(b),\ q(b),\ q(c),\ r(c)\}) = \{p(a),\ p(b),\ p(c),\ q(b),\ q(c),\ r(c)\}$: Λόγω του $q(c)$ και της πρότασης $p(x) \leftarrow q(x)$, προστίθεται και το $p(c)$.
- $T_P(T_P(T_P(T_P(\emptyset)))) =$
  $= T_P(\{p(a),\ p(b),\ p(c),\ q(b),\ q(c),\ r(c)\}) = \{p(a),\ p(b),\ p(c),\ q(b),\ q(c),\ r(c)\}$: Δεν άλλαξε το αποτέλεσμα ...
- ... και ούτε θα αλλάξει, όσο και αν εφαρμόσουμε τον T_P.

Συνεπώς, το ελάχιστο μοντέλο του προγράμματος είναι το:

$$M_P = \{p(a),\ p(b),\ p(c),\ q(b),\ q(c),\ r(c)\}$$

Άσκηση 8.4
Θεωρώντας πάλι το πρόγραμμα P, για το κατηγόρημα $myplus/3$ της Άσκησης 8.1, και τον τελεστή άμεσου επακόλουθου T_P, για το πρόγραμμα αυτό, ποια είναι η εικόνα $T_P(\emptyset)$ της κενής ερμηνείας $\emptyset$;

Άσκηση 8.5
Έστω το λογικό πρόγραμμα P:

$$\begin{array}{rcl} p(x,y) & \leftarrow & q(x,y) \\ p(x,x) & \leftarrow & p(x,y) \\ p(y,y) & \leftarrow & p(x,y) \\ q(a,b) & \leftarrow & \end{array}$$

Αν τα σύμβολα x και y είναι μεταβλητές, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Κατασκευάστε το ελάχιστο μοντέλο M_P του προγράμματος.

8.3 Λειτουργική σημασιολογία

Στην Ενότητα 8.1 είδαμε έναν δηλωτικό τρόπο χαρακτηρισμού του ελάχιστου μοντέλου ενός οριστικού προγράμματος, μέσω των λογικών επακόλουθων του προγράμματος. Στην Ενότητα 8.2 εξετάσαμε μια μέθοδο κατασκευής αυτού του ελάχιστου μοντέλου, μέσω του τελεστή άμεσου επακόλουθου και του σταθερού του σημείου. Ουσιαστικά, αυτή η μέθοδος κατασκευής εισάγει έναν τρόπο **εμπρόσθιας συλλογιστικής** για τις προτάσεις του προγράμματος, αφού τις διαχειρίζεται με κατεύθυνση από την υπόθεση (σώμα) προς το συμπέρασμα (κεφαλή). Δεν είναι όμως απαραίτητο να μας ενδιαφέρει πάντοτε το πλήρες ελάχιστο μοντέλο ενός οριστικού προγράμματος. Συνήθως, είναι χρήσιμο να μπορούμε να μάθουμε αν κάποιο συγκεκριμένο βασικό άτομο ανήκει στο ελάχιστο μοντέλο ή, με άλλα λόγια, αν είναι λογικό επακόλουθο του προγράμματος.

Η τρίτη μεθοδολογία μελέτης της σημασίας οριστικών προγραμμάτων, η **λειτουργική σημασιολογία**, που θα δούμε σε αυτήν την ενότητα, προτείνει έναν αλγόριθμο με τον οποίο έχουμε τη δυνατότητα να αποδείξουμε αν κάτι προκύπτει από ένα οριστικό πρόγραμμα,

δηλαδή αν είναι λογικό επακόλουθό του [3, 4, 5, 6]. Όπως θα δούμε, η μεθοδολογία αυτή αντικατοπτρίζει έναν τρόπο **οπίσθιας συλλογιστικής**, γιατί επεξεργάζεται τις προτάσεις του προγράμματος με κατεύθυνση από το συμπέρασμα (κεφαλή) προς την υπόθεση (σώμα).

Στην ενότητα αυτή δεν χρειάζεται να παραθέσουμε πληθώρα ορισμών, γιατί ήδη στις Ενότητες 7.2 και 7.3 έχουμε παρουσιάσει τα μέσα που θα χρησιμοποιήσουμε για τη λειτουργική σημασιολογία των οριστικών προγραμμάτων, συγκεκριμένα το μηχανισμό της ενοποίησης και την αρχή της ανάλυσης. Η αρχή της ανάλυσης είναι ένας κανόνας συμπερασμού, μια ειδική περίπτωση του οποίου, την **SLD-ανάλυση**, θα εφαρμόσουμε για τις ανάγκες της λειτουργικής σημασιολογίας.

Πριν προχωρήσουμε, πρέπει να διευκρινίσουμε ότι ένας κανόνας συμπερασμού στη λογική πρώτης τάξης είναι, καταρχάς, αυθαίρετος. Δηλαδή, μπορούμε να προτείνουμε έναν κανόνα συμπερασμού περιγράφοντας αναλυτικά πώς από ένα σύνολο τύπων S κατασκευάζουμε ένα νέο τύπο F. Έχοντας κάνει κάτι τέτοιο, που το συμβολίζουμε με $S \vdash F$, δεν είναι καθόλου βέβαιο ότι αυτός ο κανόνας συμπερασμού θα δημιουργήσει τύπους οι οποίοι θα είναι λογικά επακόλουθα των δεδομένων μας. Συγκεκριμένα, τα βασικά ερωτήματα είναι τα εξής δύο:

1. *Αν με κάποιον κανόνα συμπερασμού από το S παράγεται το F, δηλαδή αν $S \vdash F$, είναι το F λογικό επακόλουθο του S, με τη μοντελοθεωρητική έννοια, ισχύει δηλαδή το $S \models F$;*
2. *Αν κάποιο F είναι λογικό επακόλουθο, με τη μοντελοθεωρητική έννοια, ενός S, δηλαδή $S \models F$, είναι σε θέση ο κανόνας συμπερασμού που έχουμε να κατασκευάσει το F από το S, ισχύει δηλαδή $S \vdash F$;*

Οι κανόνες συμπερασμού για τους οποίους η απάντηση στο πρώτο ερώτημα είναι καταφατική θεωρούνται **ορθοί**, ενώ αυτοί για τους οποίους η απάντηση στο δεύτερο ερώτημα είναι καταφατική θεωρούνται **πλήρεις**. Προφανώς, οι «καλοί» κανόνες συμπερασμού είναι και ορθοί και πλήρεις, δηλαδή προτείνουν έναν αλγόριθμο υλοποίησης της μοντελοθεωρητικής σχέσης του «συμπεραίνειν», τίποτα περισσότερο ή τίποτα λιγότερο από αυτήν.

Ας δούμε όμως τον κανόνα συμπερασμού της SLD-ανάλυσης. Ο κανόνας αυτός, που είναι μια ειδίκευση του modus ponens, συνδυάζει έναν οριστικό στόχο με μια οριστική πρόταση και παράγει έναν νέο οριστικό στόχο. Συγκεκριμένα, αν G είναι ένας οριστικός στόχος $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ και C είναι μια οριστική πρόταση $A \leftarrow B_1, \ldots, B_q$, τότε ο οριστικός στόχος $G' = (\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots A_k)\theta$ παράγεται από τα G και C, όπου θ είναι ο γενικότερος ενοποιητής των ατόμων A_m και A. Συνοπτικά:

SLD-ανάλυση

$$\frac{\begin{array}{ll} \text{Αν } \theta = mgu(A_m, A) & \\ \quad \leftarrow & A_1, \ldots, A_m, \ldots, A_k \\ A \leftarrow & B_1, \ldots, B_q \end{array}}{(\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots A_k)\theta}$$

Το πολύ ενδιαφέρον όμως χαρακτηριστικό της SLD-ανάλυσης είναι ότι αποδεικνύεται το εξής:

Ο κανόνας συμπερασμού της SLD-ανάλυσης είναι ορθός και πλήρης.

Εκτός από την ανάγκη απόδειξης ότι ένα βασικό άτομο είναι λογικό επακόλουθο ενός οριστικού προγράμματος, δηλαδή ανήκει στο ελάχιστο μοντέλο του, τις περισσότερες φορές θέλουμε κάτι πιο πολύπλοκο. Συγκεκριμένα, έχουμε ένα σύνολο από συζευγμένα άτομα,

που περιέχουν, κατά κανόνα, και μεταβλητές, δηλαδή δεν είναι βασικά, και ζητάμε να βρούμε αν υπάρχει αποτίμηση των μεταβλητών αυτών, τέτοια ώστε τα βασικά άτομα που θα προκύψουν μετά την αντικατάσταση των μεταβλητών από τις τιμές τους να είναι λογικά επακόλουθα του προγράμματος. Με όρους λογικής πρώτης τάξης, θέλουμε να ελέγξουμε αν ο τύπος:

$$(\exists x_1)(\exists x_2)\ \ldots\ (\exists x_n)(p_1(\ \ldots\ ) \wedge p_2(\ \ldots\ ) \wedge\ \ldots\ \wedge p_m(\ \ldots\ ))$$

όπου τα άτομα p_i ($1 \leq i \leq m$) περιέχουν τις μεταβλητές x_j ($1 \leq j \leq n$), είναι λογικό επακόλουθο ενός οριστικού προγράμματος. Ας πάρουμε την άρνηση αυτού του τύπου:

$$\neg(\exists x_1)(\exists x_2)\ \ldots\ (\exists x_n)(p_1(\ \ldots\ ) \wedge p_2(\ \ldots\ ) \wedge\ \ldots\ \wedge p_m(\ \ldots\ ))$$

Τότε μπορούμε εύκολα να δούμε (θυμηθείτε το Παράδειγμα 8.3) ότι αυτή είναι ισοδύναμη με τον τύπο:

$$(\forall x_1)(\forall x_2)\ \ldots\ (\forall x_n)(\neg p_1(\ \ldots\ ) \vee \neg p_2(\ \ldots\ ) \vee\ \ldots\ \vee \neg p_m(\ \ldots\ ))$$

Αυτός όμως ο τύπος δεν είναι τίποτε άλλο από τον οριστικό στόχο:

$$\leftarrow p_1(\ \ldots\ ),\ p_2(\ \ldots\ ),\ \ldots\ ,\ p_m(\ \ldots\ )$$

Οπότε, αυτό που μπορούμε να κάνουμε πλέον είναι να προσπαθήσουμε από το συνδυασμό του οριστικού προγράμματος που έχουμε και του προηγούμενου οριστικού στόχου να παραγάγουμε την αντίφαση, δηλαδή την κενή πρόταση. Αν το καταφέρουμε αυτό, σημαίνει ότι η άρνηση της ερώτησής μας και το πρόγραμμα είναι ασυνεπή. Άρα, η ερώτηση είναι λογικό επακόλουθο του προγράμματος (δείτε πάλι την Πρόταση 8.1). Ο ορισμός που ακολουθεί παρουσιάζει πιο τυπικά το θέμα και δίνει κάποιες επιπλέον λεπτομέρειες.

Ορισμός 8.15
Δεδομένου ενός οριστικού προγράμματος P επάνω σε μια γλώσσα πρώτης τάξης L και ενός οριστικού στόχου G, μια **SLD-απόρριψη** του $P \cup \{G\}$ είναι μια διαδοχική εφαρμογή του κανόνα συμπερασμού της SLD-ανάλυσης πεπερασμένες φορές, όπου σε κάθε βήμα χρησιμοποιείται ο στόχος που παρήχθη στο προηγούμενο βήμα (στο πρώτο βήμα χρησιμοποιείται ο G) και μια παραλλαγή κάποιας πρότασης του P, τέτοια ώστε οι μεταβλητές της να είναι διαφορετικές από αυτές που έχουν χρησιμοποιηθεί μέχρι εκείνη τη στιγμή, και, επιπλέον, ο τελευταίος στόχος που παράγεται είναι η κενή πρόταση $\leftarrow$.

Ορισμός 8.16
Δεδομένου ενός οριστικού προγράμματος P επάνω σε μια γλώσσα πρώτης τάξης L, το σύνολο όλων των $A \in B_L$ για τα οποία το $P \cup \{\leftarrow A\}$ έχει κάποια SLD-απόρριψη ονομάζεται **σύνολο επιτυχίας** του P, συμβολικά $SS(P)$. □

Το αποτέλεσμα της λειτουργικής σημασιολογίας για τα οριστικά προγράμματα, που συσχετίζει τις αλγοριθμικές δυνατότητες της SLD-ανάλυσης με τη μοντελοθεωρητική σημασιολογία (αλλά και τη σημασιολογία σταθερού σημείου) των οριστικών προγραμμάτων, συνοψίζεται στο θεώρημα που ακολουθεί, το οποίο παραθέτουμε χωρίς απόδειξη.

Θεώρημα 8.3
Το ελάχιστο μοντέλο M_P ενός οριστικού προγράμματος P είναι το σύνολο επιτυχίας του. Δηλαδή:

$$M_P = SS(P) \qquad \square$$

Έτσι, έχουμε στη διάθεσή μας έναν διαδικαστικό τρόπο, την SLD-απόρριψη, που χρησιμοποιεί τον κανόνα συμπερασμού της SLD-ανάλυσης, για να ελέγχουμε αν ένα βασικό άτομο ανήκει στο ελάχιστο μοντέλο ενός οριστικού προγράμματος.

Παράδειγμα 8.7
Θεωρώντας πάλι το οριστικό πρόγραμμα που δώσαμε στο Παράδειγμα 8.4, ας δούμε πώς μπορεί να αποδειχθεί ότι το $p(c)$ ανήκει στο ελάχιστο μοντέλο του προγράμματος.

Αρχίζουμε με τον οριστικό στόχο $\leftarrow p(c)$, τον οποίο μπορούμε να συνδυάσουμε με την πρόταση $p(x) \leftarrow q(x)$, και, εφαρμόζοντας SLD-ανάλυση, να πάρουμε το στόχο $\leftarrow q(c)$. Ο στόχος αυτός μπορεί με τη σειρά του να συνδυαστεί με τη δεύτερη πρόταση του προγράμματος, συγκεκριμένα με μια παραλλαγή της, έστω την $q(x') \leftarrow r(x')$, επειδή η μεταβλητή x έχει ήδη χρησιμοποιηθεί, και να πάρουμε τον οριστικό στόχο $\leftarrow r(c)$. Με μια τελευταία εφαρμογή της SLD-ανάλυσης μεταξύ του στόχου αυτού και της πρότασης του προγράμματος $r(c) \leftarrow$, παίρνουμε την κενή πρόταση $\leftarrow$.

Συνεπώς, έχοντας κατασκευάσει μια SLD-απόρριψη για το δεδομένο πρόγραμμα και το στόχο $\leftarrow p(c)$, συμπεραίνουμε ότι το $p(c)$ ανήκει στο σύνολο επιτυχίας του προγράμματος, άρα και στο ελάχιστο μοντέλο του, δηλαδή είναι λογικό επακόλουθο του προγράμματος. $\square$

Η διαδικασία εφαρμογής της SLD-ανάλυσης που περιγράψαμε σε αυτήν την ενότητα είναι η βάση του μηχανισμού τον οποίο χρησιμοποιεί η γλώσσα προγραμματισμού Prolog για να απαντά σε ερωτήσεις που της υποβάλλονται. Υπάρχουν κάποια σημεία στα οποία διαφοροποιείται η Prolog. Για παράδειγμα, σε κάθε βήμα εφαρμογής του κανόνα συμπερασμού επιλέγει πάντα το πρώτο άτομο από τον τρέχοντα στόχο, για να απαλείψει με την κεφαλή μιας πρότασης. Επίσης, για το χειρισμό της άρνησης, χρησιμοποιεί μια παραλλαγή της SLD-ανάλυσης, την **SLDNF-ανάλυση**. Η βασική ιδέα όμως είναι αυτή που είδαμε στην παρούσα ενότητα.

Άσκηση 8.6
Θεωρώντας το πρόγραμμα P της Άσκησης 8.5, ποια από τα $q(b,b)$, $p(a,b)$, $q(b,a)$, $p(b,b)$ και $q(a,b)$ ανήκουν στο σύνολο επιτυχίας $SS(P)$ του P;

Άσκηση 8.7
Έχοντας το πρόγραμμα P της Άσκησης 8.1, μπορείτε να κατασκευάσετε μια SLD-απόρριψη για το:

$$P \cup \{\leftarrow myplus(s(0), s(s(0)), w),\ myplus(s(s(s(0))), u, w)\}$$

όπου τα w και u είναι μεταβλητές; Τι δηλώνει αυτή η SLD-απόρριψη για τις τιμές των μεταβλητών w και u;

Απαντήσεις ασκήσεων

Απάντηση άσκησης 8.1
Το σύμπαν Herbrand για το P είναι το:

$$U_L = \{0,\ s(0),\ s(s(0)),\ s(s(s(0))),\ \ldots\}$$

και η βάση Herbrand είναι η:

$$B_L = \{myplus(0,0,0),\ myplus(0,0,s(0)),\ myplus(0,s(0),0),\ myplus(s(0),0,0),$$

$myplus(0, s(0), s(0)),\ \ldots\ , myplus(s(0), s(0), s(0)),\ \ldots\ , myplus(0, 0, s(s(0))),$
$\ldots\ , myplus(s(0), s(0), s(s(0))),\ \ldots\ , myplus(s(s(s(0))), s(0), 0),\ \ldots\}$

Είναι δυνατόν το βασικό άτομο $myplus(0, 0, s(0))$ να περιέχεται σε ένα μοντέλο M του P, αλλά τότε, για να είναι το M μοντέλο της πρότασης (8.2) του P, θα πρέπει το M να περιέχει επίσης τα άτομα $myplus(0, s(0), s(s(0)))$ και $myplus(0, s(s(0)), s(s(s(0))))$, όπως και το $myplus(0, s(s(s(0))), s(s(s(s(0)))))$, και όλα τα υπόλοιπα, με την ίδια λογική.

Έχοντας στο μυαλό μας την πρόθεση του συγγραφέα του προγράμματος (δείτε πάλι την Άσκηση 4.3), που ήταν η υλοποίηση της πρόσθεσης, καταλαβαίνουμε ότι το άτομο αυτό περιγράφει ουσιαστικά τη σχέση $0 + 0 = 1$. Βέβαια, αυτό αρνούμαστε να το δεχτούμε. Η άποψή μας ότι η σχέση αυτή είναι λάθος δεν μπορεί να απαγορεύσει στο αντίστοιχο άτομο να βρίσκεται μέσα σε ένα μοντέλο του προγράμματος, αλλά, αν βρίσκεται, τότε θα πρέπει να βρίσκονται και τα άλλα άτομα που αναφέραμε, δηλαδή θα πρέπει να ισχύουν και οι σχέσεις $0+1 = 2$, $0+2 = 3$, $0+3 = 4$ κ.ο.κ. Λογικό δεν είναι; Ωστόσο, αποκλείεται μόνο το άτομο $myplus(0, 0, s(0))$ να βρίσκεται στο ελάχιστο μοντέλο του προγράμματος, γιατί σε καμία περίπτωση δεν είναι λογικό επακόλουθό του.

Απάντηση άσκησης 8.2
Το σύμπαν Herbrand για το P είναι το:

$$U_L = \{a,\ b,\ c,\ d\}$$

και η βάση Herbrand η:

$$B_L = \{p(a),\ p(b),\ p(c),\ p(d),\ q(a),\ q(b),\ q(c),\ q(d),$$
$$r(a),\ r(b),\ r(c),\ r(d),\ s(a),\ s(b),\ s(c),\ s(d)\}$$

Σχετικά με τις ερμηνείες I_1, I_2, I_3, I_4, και I_5:

- Η I_1 δεν είναι μοντέλο του P, γιατί, αφού περιέχει το $s(d)$, θα έπρεπε να περιέχει και το $r(d)$, λόγω της πρότασης (8.4). Άλλος εναλλακτικός λόγος είναι ότι, αφού περιέχει τα $q(b)$ και $r(b)$, θα έπρεπε να περιέχει και το $p(b)$, λόγω της πρότασης (8.3).
- Η I_2 δεν είναι μοντέλο του P, γιατί δεν περιέχει το $r(c)$, οπότε δεν είναι μοντέλο της πρότασης (8.9).
- Η I_3 δεν είναι μοντέλο του P, γιατί δεν περιέχει το $r(b)$, οπότε δεν είναι μοντέλο της πρότασης (8.8). Άλλος εναλλακτικός λόγος είναι ότι, αφού περιέχει το $s(d)$, θα έπρεπε να περιέχει και το $p(d)$, λόγω της πρότασης (8.5).
- Η I_4 δεν είναι μοντέλο του P, γιατί, αφού περιέχει το $q(c)$ και το $r(c)$, θα έπρεπε να περιέχει και το $p(c)$, λόγω της πρότασης (8.3).
- Η I_5 δεν είναι μοντέλο του P, γιατί, αφού περιέχει το $s(d)$, θα έπρεπε να περιέχει και το $p(d)$, λόγω της πρότασης (8.5). Άλλοι εναλλακτικοί λόγοι είναι ότι, αφού περιέχει το $s(a)$, θα έπρεπε να περιέχει και τα $r(a)$ και $p(a)$, λόγω των προτάσεων (8.4) και (8.5), αντίστοιχα.

Όσον αφορά την ερμηνεία I_6, δεν υπάρχει προφανής λόγος για τον οποίο δεν θα μπορούσε να ήταν το ελάχιστο μοντέλο του P. Μπορούμε σχετικά εύκολα να επαληθεύσουμε ότι είναι όντως μοντέλο του P, ελέγχοντας αν είναι μοντέλο για καθεμία από τις προτάσεις του P. Αλλά, η επαλήθευση αν είναι ελάχιστο μοντέλο είναι σχετικά δύσκολη, γιατί θα

πρέπει, με όσα γνωρίζουμε μέχρι στιγμής, να βεβαιωθούμε ότι δεν υπάρχει υποσύνολο της I_6 που να είναι μοντέλο του P. Κάτι τέτοιο δεν είναι αδύνατο βέβαια, αλλά απαιτεί αρκετή δουλειά. Καλύτερα να περιμένουμε μέχρι την Ενότητα 8.2, στην οποία θα εξετάσουμε μια μέθοδο κατασκευής του ελάχιστου μοντέλου ενός οριστικού προγράμματος, οπότε, εφαρμόζοντάς την, θα καταλάβουμε ότι όντως η ερμηνεία I_6 είναι το ελάχιστο μοντέλο του P.

Απάντηση άσκησης 8.3
Ένα πολύ απλό πρόγραμμα που δεν είναι οριστικό αποτελείται από την πρόταση:

$$p(a),\ q(b) \leftarrow$$

Οι ερμηνείες $I_1 = \{p(a)\}$ και $I_2 = \{q(b)\}$ είναι μοντέλα του προγράμματος, αλλά η τομή τους $I_1 \cap I_2 = \emptyset$ δεν είναι.

Απάντηση άσκησης 8.4
Η κενή ερμηνεία $\emptyset$ έχει, μέσω του τελεστή άμεσου επακόλουθου, σαν εικόνα πάντοτε τα στοιχεία εκείνα της βάσης Herbrand B_L που είναι στιγμιότυπα των κεφαλών των μοναδιαίων προτάσεων του προγράμματος. Για την περίπτωσή μας, μόνο η πρόταση (8.1) είναι μοναδιαία. Οπότε, έχουμε:

$$T_P(\emptyset) = \{myplus(0,0,0),\ myplus(s(0),0,s(0)),\ myplus(s(s(0)),0,s(s(0))),\ \ldots\}$$

Απάντηση άσκησης 8.5
Το σύμπαν Herbrand για το P είναι το:

$$U_L = \{a,\ b\}$$

και η βάση Herbrand η:

$$B_L = \{p(a,a),\ p(a,b),\ p(b,a),\ p(b,b),\ q(a,a),\ q(a,b),\ q(b,a),\ q(b,b)\}$$

Για το ελάχιστο μοντέλο M_P, γνωρίζουμε ότι:

$$\begin{aligned}
M_P &= lub(\{\emptyset,\ T_P(\emptyset),\ T_P(T_P(\emptyset)),\ T_P(T_P(T_P(\emptyset))),\ T_P(T_P(T_P(T_P(\emptyset)))),\ \ldots\}) \\
&= lub(\{\emptyset,\ \{q(a,b)\},\ T_P(\{q(a,b)\}),\ T_P(T_P(\{q(a,b)\})), \\
&\qquad T_P(T_P(T_P(\{q(a,b)\}))),\ \ldots\}) \\
&= lub(\{\emptyset,\ \{q(a,b)\},\ \{q(a,b),\ p(a,b)\},\ T_P(\{q(a,b),\ p(a,b)\}), \\
&\qquad T_P(T_P(\{q(a,b),\ p(a,b)\})),\ \ldots\}) \\
&= lub(\{\emptyset,\ \{q(a,b)\},\ \{q(a,b),\ p(a,b)\},\ \{q(a,b),\ p(a,b),\ p(a,a),\ p(b,b)\}, \\
&\qquad T_P(\{q(a,b),\ p(a,b), p(a,a),\ p(b,b)\}),\ \ldots\}) \\
&= lub(\{\emptyset,\ \{q(a,b)\},\ \{q(a,b),\ p(a,b)\},\ \{q(a,b),\ p(a,b),\ p(a,a),\ p(b,b)\}, \\
&\qquad \{q(a,b),\ p(a,b),\ p(a,a),\ p(b,b)\},\ \ldots\}) \\
&= \{q(a,b),\ p(a,b),\ p(a,a),\ p(b,b)\}
\end{aligned}$$

Απάντηση άσκησης 8.6
Στην απάντηση της Άσκησης 8.5 κατασκευάσαμε το ελάχιστο μοντέλο M_P του προγράμματος P, μέσω του τελεστή άμεσου επακόλουθου. Βρήκαμε ότι:

$$M_P = \{q(a,b),\ p(a,b),\ p(a,a),\ p(b,b)\}$$

Αλλά, όπως γνωρίζουμε από το θεώρημα 8.3, το ελάχιστο μοντέλο ταυτίζεται με το σύνολο επιτυχίας του προγράμματος. Συνεπώς, από τα βασικά άτομα που δίνονται στην εκφώνηση της άσκησης, μόνο τα $p(a,b)$, $p(b,b)$ και $q(a,b)$ ανήκουν στο σύνολο επιτυχίας του P.

Απάντηση άσκησης 8.7
Ορίστε μια SLD-απόρριψη που θα μπορούσαμε να κατασκευάσουμε:

	$\theta_1 = \{x_1/s(0),\ y_1/s(0),\ w/s(z_1)\}$		
		$\leftarrow$	$myplus(s(0), s(s(0)), w),\ myplus(s(s(s(0))), u, w)$
(8.2)	$myplus(x_1, s(y_1), s(z_1))$	$\leftarrow$	$myplus(x_1, y_1, z_1)$
		$($ $\leftarrow$	$myplus(x_1, y_1, z_1),\ myplus(s(s(s(0))), u, w)\ )\theta_1$
	$\theta_2 = \{x_2/s(0),\ y_2/0,\ z_1/s(z_2)\}$		
		$\leftarrow$	$myplus(s(0), s(0), z_1),\ myplus(s(s(s(0))), u, s(z_1))$
(8.2)	$myplus(x_2, s(y_2), s(z_2))$	$\leftarrow$	$myplus(x_2, y_2, z_2)$
		$($ $\leftarrow$	$myplus(x_2, y_2, z_2),\ myplus(s(s(s(0))), u, s(z_1))\ )\theta_2$
	$\theta_3 = \{x_3/s(0),\ z_2/s(0)\}$		
		$\leftarrow$	$myplus(s(0), 0, z_2),\ myplus(s(s(s(0))), u, s(s(z_2)))$
(8.1)	$myplus(x_3, 0, x_3)$	$\leftarrow$	
		$($ $\leftarrow$	$myplus(s(s(s(0))), u, s(s(z_2)))\ )\theta_3$
	$\theta_4 = \{x_4/s(s(s(0))),\ u/0\}$		
		$\leftarrow$	$myplus(s(s(s(0))), u, s(s(s(0))))$
(8.1)	$myplus(x_4, 0, x_4)$	$\leftarrow$	
		$\leftarrow$	

Όσον αφορά την τιμή της μεταβλητής u, βλέπουμε ότι, από τον γενικότερο ενοποιητή θ_4, αυτή είναι 0. Για την τιμή της μεταβλητής w, πρέπει να συνθέσουμε τους γενικότερους ενοποιητές θ_1, θ_2 και θ_3, οπότε θα πάρουμε την $s(s(s(0)))$. Τελικά δηλαδή, μια SLD-απόρριψη είναι σε θέση να μας δώσει και τιμές για τις μεταβλητές του οριστικού στόχου που την ενεργοποίησε.

Προβλήματα

Πρόβλημα 8.1
Πότε το ελάχιστο μοντέλο ενός οριστικού προγράμματος είναι το κενό σύνολο; Τι επίπτωση έχει αυτό το γεγονός;

Πρόβλημα 8.2
Δώστε δύο πολύ απλά παραδείγματα οριστικών προγραμμάτων με βάσεις Herbrand, που να είναι απειροσύνολα και στο μεν ένα το ελάχιστο μοντέλο να είναι πεπερασμένο, στο δε άλλο απειροσύνολο.

Πρόβλημα 8.3
Είναι η βάση Herbrand μοντέλο για ένα οριστικό πρόγραμμα; Για ένα μη οριστικό πρόγραμμα; Αιτιολογήστε τις απαντήσεις σας.

Πρόβλημα 8.4
Έστω μια γλώσσα πρώτης τάξης στην οποία $p/1, q/1 \in P$, $a/0, b/0 \in F$ και $x \in V$. Δώστε μια οριστική πρόταση αυτής της γλώσσας και δύο μοντέλα της πρότασης που να είναι ξένα

μεταξύ τους. Αν ένα οριστικό πρόγραμμα αποτελείται μόνο από αυτή την πρόταση, ποιο είναι το ελάχιστο μοντέλο του; Έχετε να κάνετε κάποιο ενδιαφέρον σχόλιο;

Πρόβλημα 8.5
Δώστε ένα απλό παράδειγμα οριστικού προγράμματος, τη βάση Herbrand που αντιστοιχεί σε αυτό και μια ερμηνεία I, η οποία όμως να μην είναι μοντέλο για το πρόγραμμα, αλλά τέτοια ώστε, είτε διαγράφοντας το στοιχείο A_1 από αυτήν είτε προσθέτοντας το στοιχείο A_2 σε αυτήν, οι ερμηνείες που προκύπτουν, δηλαδή οι $I_1 = I - \{A_1\}$ και $I_2 = I \cup \{A_2\}$, αντίστοιχα, να είναι μοντέλα του προγράμματος. Ποια είναι τα A_1 και A_2, για το παράδειγμά σας; Υπάρχει περίπτωση κάποια, ή και οι δύο, από τις ερμηνείες I_1 και I_2 να είναι το ελάχιστο μοντέλο του προγράμματος;

Πρόβλημα 8.6
Έστω το οριστικό πρόγραμμα P:

$$\begin{array}{rcl} p(a) & \leftarrow & \\ p(f(b)) & \leftarrow & \\ p(f(f(x))) & \leftarrow & p(x) \end{array}$$

Αν το σύμβολο x είναι μεταβλητή, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Κατασκευάστε το ελάχιστο μοντέλο M_P του προγράμματος. Μπορεί το $p(f(a))$ να ανήκει σε κάποιο μοντέλο του προγράμματος; Αν όχι, γιατί; Αν ναι, δώστε την τομή όλων των μοντέλων που περιλαμβάνουν το $p(f(a))$. Ομοίως, για το $p(f(f(f(f(b)))))$.

Πρόβλημα 8.7
Έστω το οριστικό πρόγραμμα P:

$$\begin{array}{rcl} p(a, f(b)) & \leftarrow & \\ p(x, f(f(y))) & \leftarrow & p(y, f(x)) \end{array}$$

Αν τα σύμβολα x και y είναι μεταβλητές, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Ποια από τα $p(a, b), p(b, f(a))$ και $p(f(a), f(f(b)))$ ανήκουν στο σύνολο επιτυχίας $SS(P)$ του P και γιατί;

Πρόβλημα 8.8
Έστω το οριστικό πρόγραμμα P:

$$\begin{array}{rcl} p(a, c) & \leftarrow & \\ p(b, c) & \leftarrow & \\ p(x, x) & \leftarrow & \\ p(x, y) & \leftarrow & p(y, x) \\ p(x, z) & \leftarrow & p(x, y), p(y, z) \end{array}$$

Αν τα σύμβολα x, y και z είναι μεταβλητές, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Κατασκευάστε, σύμφωνα με το αποτέλεσμα της σημασιολογίας σταθερού σημείου, το ελάχιστο μοντέλο M_P αυτού του προγράμματος. Δώστε μια SLD-απόρριψη του $P \cup \{\leftarrow p(a, b)\}$. Θα γράφατε ποτέ αυτό το οριστικό πρόγραμμα σε Prolog, φυσικά μετά τις κατάλληλες συντακτικές τροποποιήσεις; Αν όχι, γιατί;

Πρόβλημα 8.9
Έστω το οριστικό πρόγραμμα P:

$$\begin{aligned} p(f(h(a))) &\leftarrow \\ p(h(x)) &\leftarrow p(x) \\ p(x) &\leftarrow q(h(x)) \\ q(x) &\leftarrow p(f(x)) \end{aligned}$$

Αν το σύμβολο x είναι μεταβλητή, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Βρείτε το ελάχιστο μοντέλο M_P του προγράμματος, εφαρμόζοντας την κατάλληλη μέθοδο για το σκοπό αυτό. Μπορεί το $q(h(f(a)))$ να ανήκει σε κάποιο μοντέλο του προγράμματος; Αν όχι, γιατί; Αν ναι, δώστε την τομή όλων των μοντέλων που περιλαμβάνουν και το $q(h(f(a)))$.

Πρόβλημα 8.10
Έστω το οριστικό πρόγραμμα P:

$$\begin{aligned} p(x, f(y)) &\leftarrow p(f(x), y) \\ p(f(y), x) &\leftarrow p(x, f(y)) \\ p(f(a), b) &\leftarrow \end{aligned}$$

Αν τα σύμβολα x και y είναι μεταβλητές, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Βρείτε το ελάχιστο μοντέλο M_P του προγράμματος, εφαρμόζοντας την κατάλληλη μέθοδο για το σκοπό αυτό. Μπορεί το $p(a, a)$ να ανήκει σε κάποιο μοντέλο του προγράμματος; Αν όχι, γιατί; Αν ναι, δώστε την τομή όλων των μοντέλων που περιλαμβάνουν και το $p(a, a)$. Ομοίως, για το $p(b, f(b))$.

Πρόβλημα 8.11
Έστω το οριστικό πρόγραμμα P:

$$\begin{aligned} p(g(x)) &\leftarrow p(f(f(x))) \\ p(f(x)) &\leftarrow p(g(g(x))) \\ p(f(f(g(f(g(a)))))) &\leftarrow \end{aligned}$$

Αν το σύμβολο x είναι μεταβλητή, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Βρείτε το ελάχιστο μοντέλο M_P του προγράμματος, εφαρμόζοντας την κατάλληλη μέθοδο για το σκοπό αυτό. Μπορεί το $p(g(g(f(a))))$ να ανήκει σε κάποιο μοντέλο του προγράμματος; Αν όχι, γιατί; Αν ναι, δώστε την τομή όλων των μοντέλων που περιλαμβάνουν και το $p(g(g(f(a))))$. Ομοίως, για το $p(f(g(a)))$.

Πρόβλημα 8.12
Έστω το οριστικό πρόγραμμα P:

$$\begin{aligned} p(f(a), a) &\leftarrow \\ p(x, f(y)) &\leftarrow p(x, y) \\ p(f(x), x) &\leftarrow p(x, x) \end{aligned}$$

Αν τα σύμβολα x και y είναι μεταβλητές, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Βρείτε το ελάχιστο μοντέλο M_P του προγράμματος, εφαρμόζοντας την κατάλληλη μέθοδο για το σκοπό αυτό. Μπορεί το $p(a, a)$ να ανήκει σε κάποιο μοντέλο του

προγράμματος; Αν όχι, γιατί; Αν ναι, δώστε την τομή όλων των μοντέλων που περιλαμβάνουν και το $p(a, a)$. Ομοίως, για το $p(f(f(a)), a)$.

Πρόβλημα 8.13
Έστω το οριστικό πρόγραμμα P:

$$\begin{array}{rcl} p(a) & \leftarrow & \\ p(f(b)) & \leftarrow & \\ p(g(f(x))) & \leftarrow & p(f(x)) \\ p(f(g(x))) & \leftarrow & p(g(x)) \end{array}$$

Αν το σύμβολο x είναι μεταβλητή, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Βρείτε το ελάχιστο μοντέλο M_P του προγράμματος, εφαρμόζοντας την κατάλληλη μέθοδο για το σκοπό αυτό. Μπορεί το $p(b)$ να ανήκει σε κάποιο μοντέλο του προγράμματος; Αν όχι, γιατί; Αν ναι, δώστε την τομή όλων των μοντέλων που περιλαμβάνουν και το $p(b)$. Ομοίως, για το $p(f(a))$.

Πρόβλημα 8.14
Έστω το οριστικό πρόγραμμα P:

$$\begin{array}{rcl} p(f(f(f(a)))) & \leftarrow & \\ p(x) & \leftarrow & p(f(x)) \\ p(f(b)) & \leftarrow & p(a) \\ p(f(f(f(b)))) & \leftarrow & p(c) \end{array}$$

Αν το σύμβολο x είναι μεταβλητή, ποιο είναι το σύμπαν Herbrand και ποια η βάση Herbrand για το P; Βρείτε το ελάχιστο μοντέλο M_P του προγράμματος, εφαρμόζοντας την κατάλληλη μέθοδο για το σκοπό αυτό. Μπορεί το $p(c)$ να ανήκει σε κάποιο μοντέλο του προγράμματος; Αν όχι, γιατί; Αν ναι, δώστε την τομή όλων των μοντέλων που περιλαμβάνουν και το $p(c)$.

Πρόβλημα 8.15
Δίνονται οι εξής οριστικές προτάσεις (το x είναι μεταβλητή):

C_1: $p(f(g(a))) \leftarrow$
C_2: $p(x) \leftarrow p(f(x))$
C_3: $p(f(x)) \leftarrow p(g(x))$
C_4: $p(g(f(x))) \leftarrow p(f(g(x)))$

Έστω $C = \{C_1, C_2, C_3, C_4\}$.

1. Επιλέξτε ένα $P \subseteq C$, τέτοιο ώστε το ελάχιστο μοντέλο M_P του P να περιλαμβάνει $N = 2$ στοιχεία. Ποιο είναι το M_P;
2. Το ίδιο με το προηγούμενο, αλλά για $N = 3$.
3. Το ίδιο με το προηγούμενο, αλλά για $N = 4$.
4. Το ίδιο με το προηγούμενο, αλλά για $N = 5$.
5. Το ίδιο με το προηγούμενο, αλλά για $N = 7$.
6. Το ίδιο με το προηγούμενο, αλλά για $N = 1$.
7. Το ίδιο με το προηγούμενο, αλλά για $N = 0$.

8. Το ίδιο με το προηγούμενο, αλλά για $N = \infty$.

Σε κάποια από τα προηγούμενα ερωτήματα, ενδέχεται να υπάρχουν περισσότερες της μιας σωστές απαντήσεις. Αρκεί να δώσετε κάποια από αυτές. Επίσης, είναι ενδεχόμενο σε κάποια ερωτήματα να μην υπάρχει απάντηση. Αρκεί να το επισημάνετε.

Πρόβλημα 8.16
Έστω το οριστικό πρόγραμμα P, όπου το σύμβολο x είναι μεταβλητή:

$$\begin{aligned} p(f(f(f(a)))) &\leftarrow \\ p(x) &\leftarrow p(f(x)), p(f(f(x))) \end{aligned}$$

1. Ποιο είναι το ελάχιστο μοντέλο του προγράμματος P;
2. Προτείνετε μια οριστική πρόταση C_1, έτσι ώστε το ελάχιστο μοντέλο του προγράμματος $P \cup \{C_1\}$ να περιέχει ακριβώς τέσσερα στοιχεία. Ποιο είναι αυτό το ελάχιστο μοντέλο;
3. Προτείνετε μια οριστική πρόταση C_2, έτσι ώστε το ελάχιστο μοντέλο του προγράμματος $P \cup \{C_2\}$ να περιέχει ακριβώς πέντε στοιχεία. Ποιο είναι αυτό το ελάχιστο μοντέλο;
4. Προτείνετε μια οριστική πρόταση C_3, έτσι ώστε το ελάχιστο μοντέλο του προγράμματος $P \cup \{C_3\}$ να είναι απειροσύνολο, αλλά να μην ταυτίζεται με τη βάση Herbrand. Ποιο είναι αυτό το ελάχιστο μοντέλο;
5. Επιλέξτε μία μόνο από τις προηγούμενες περιπτώσεις και προτείνετε έναν ατομικό τύπο A που να μην ανήκει στο ελάχιστο μοντέλο. Ποια είναι η τομή όλων των μοντέλων στα οποία ανήκει ο A;

Πρόβλημα 8.17
Δίνεται το εξής οριστικό πρόγραμμα P, όπου τα σύμβολα x, y, z και w είναι μεταβλητές:

$$\begin{aligned} p(n, w, w) &\leftarrow \\ p(f(n, x), y, f(n, z)) &\leftarrow p(x, y, z) \end{aligned}$$

Αν M_P είναι το ελάχιστο μοντέλο του προγράμματος, είναι αληθές το παρακάτω;

$$p(f(n, f(n, n)), f(n, n), f(n, f(n, f(n, n)))) \in M_P$$

Αιτιολογήστε την απάντησή σας. Αν M_I είναι η τομή όλων των μοντέλων στα οποία ανήκει το $p(n, n, f(n, n))$, με τι ισούται η διαφορά $M_I - M_P$;

Βιβλιογραφικές αναφορές

[1] R. Kowalski, Predicate Logic as Programming Language, *Proceedings of IFIP '74*, 569-574, 1974.

[2] R. Kowalski, Algorithm = Logic + Control, *CACM*, 22(7), 424-436, 1979.

[3] K. R. Apt and M. H. van Emden, Contributions to the Theory of Logic Programming, *JACM*, 29(3), 841-862, 1982.

[4] K. Doets, *From Logic to Logic Programming*, The MIT Press, 1994.

[5] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1993.

[6] M. H. van Emden and R. A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *JACM*, 23(4), 73-742, 1976.

Κεφάλαιο 9
Συναρτησιακός προγραμματισμός – Υπολογισμός με συναρτήσεις

Σύνοψη

Σκοπός του κεφαλαίου αυτού είναι η εισαγωγή του αναγνώστη στη φιλοσοφία του συναρτησιακού προγραμματισμού. Ο συναρτησιακός προγραμματισμός συνίσταται στη συγγραφή συναρτησιακών προγραμμάτων, τα οποία αποτελούνται από ορισμούς μαθηματικών συναρτήσεων. Η λύση ενός προβλήματος στο περιβάλλον του συναρτησιακού προγραμματισμού προκύπτει από τον υπολογισμό εκφράσεων που περιγράφουν την εφαρμογή συναρτήσεων σε κατάλληλα δεδομένα εισόδου. Ο συναρτησιακός προγραμματισμός έχει ισχυρή θεωρητική βάση, τον λάμδα λογισμό.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει στοιχειώδεις γνώσεις μαθηματικών συναρτήσεων.

9.1 Ιστορική αναδρομή και γενικά

Ο **συναρτησιακός προγραμματισμός** είναι η δεύτερη δηλωτική προγραμματιστική φιλοσοφία που θα μελετήσουμε, μετά τον λογικό προγραμματισμό [1, 2, 3, 4]. Η αντιμετώπιση ενός προβλήματος με συναρτησιακό προγραμματισμό γίνεται μέσω της δήλωσης αξιωμάτων που περιγράφουν τι ισχύει στον κόσμο του προβλήματος και όχι πώς λύνεται το πρόβλημα, όπως άλλωστε ισχύει σε κάθε δηλωτικό τρόπο προγραμματισμού. Σε αντίθεση με τον λογικό προγραμματισμό, στον συναρτησιακό προγραμματισμό τα αξιώματα δηλώνονται ως **συναρτήσεις** και η επίλυση ενός προβλήματος ανάγεται στον **υπολογισμό εκφράσεων**, που δεν είναι τίποτε άλλο από την εφαρμογή συναρτήσεων σε κατάλληλες τιμές εισόδου. Αντίθετα, όπως είδαμε μέχρι τώρα, στον λογικό προγραμματισμό τα αξιώματα είναι συνεπαγωγές και τα προβλήματα επιλύονται με την υποβολή ερωτήσεων και τον υπολογισμό των απαντήσεων σε αυτές με την εφαρμογή ενός μηχανισμού ενοποίησης και της αρχής της ανάλυσης.

Ο συναρτησιακός προγραμματισμός έχει ισχυρή θεωρητική θεμελίωση, τον **λάμδα λογισμό**, που εισήχθη από τον Church τη δεκαετία του 1930. Ο λάμδα λογισμός είναι ένα αξιωματικά ορισμένο μαθηματικό σύστημα, πρόθεση του οποίου είναι να παράσχει τα αναγκαία μέσα, για να περιγραφεί η υπολογιστική συμπεριφορά των μαθηματικών συναρτήσεων. Τα μέσα αυτά είναι οι συντακτικοί κανόνες για τη διατύπωση **λάμδα εκφράσεων** και ένα σύνολο από **κανόνες μετασχηματισμού** μεταξύ λάμδα εκφράσεων. Στην Ενότητα 12.1, θα παρουσιαστούν αναλυτικότερα κάποια στοιχεία του λάμδα λογισμού.

Εκτός από τον Church, και άλλοι σημαντικοί επιστήμονες του 20ού αιώνα, όπως οι Rosser, Kleene, Turing, Schönfinkel και Curry, μελέτησαν τον λάμδα λογισμό, καθώς και θέματα συναφή με αυτόν, όπως η σχέση του με τις έννοιες της αναδρομικότητας και της υπολογισιμότητας, αλλά και η **λογική των συνδυαστών**, την οποία θα συναντήσουμε στην Ενότητα 12.2. Έτσι, δημιουργήθηκε σιγά σιγά η βάση στην οποία έμελλε να στηριχθεί η ιδέα του συναρτησιακού προγραμματισμού. Η περιγραφή όσων ισχύουν στον κόσμο ενός προβλήματος που μας ενδιαφέρει γίνεται, στον συναρτησιακό προγραμματισμό, μέσω της συγγραφής ενός **συναρτησιακού προγράμματος**. Το πρόγραμμα αυτό αποτελείται μόνο από τους ορισμούς κατάλληλων μαθηματικών συναρτήσεων. Μπορούμε, λοιπόν, να δώσουμε τον εξής ορισμό:

Συναρτησιακό πρόγραμμα είναι ένα σύνολο από αξιώματα που ισχύουν σε κάποιον κόσμο διατυπωμένα ως ορισμοί μαθηματικών συναρτήσεων.

Κατά μια έννοια, ο λάμδα λογισμός μπορεί να θεωρηθεί η πρώτη γλώσσα συναρτησιακού προγραμματισμού, αν και την εποχή κατά την οποία προτάθηκε δεν υπήρχαν υπολογιστές για να «εκτελούν» συναρτησιακά προγράμματα εκφρασμένα με βάση τους κανόνες του λάμδα λογισμού. Από την άλλη πλευρά, πολλοί διατυπώνουν την άποψη ότι πρόγονος των συναρτησιακών γλωσσών προγραμματισμού είναι η Lisp, που κατασκευάστηκε από τον McCarthy τη δεκαετία του 1950. Αυτή η άποψη είναι εν μέρει σωστή και εν μέρει λανθασμένη. Η ορθότητά της έγκειται στο γεγονός ότι στη Lisp ένα πρόγραμμα είναι ένα σύνολο από ορισμούς συναρτήσεων. Η διατύπωση όμως αυτών των ορισμών στη Lisp δεν ακολουθεί αυστηρά τις απαιτήσεις του λάμδα λογισμού, με συνέπεια να εμφανίζονται διάφορες θεωρητικές, αλλά και πρακτικές, ανεπιθύμητες παρενέργειες. Εν πάση περιπτώσει, η πρόθεση του McCarthy δεν ήταν, με την εισαγωγή της Lisp, η εφαρμογή του λάμδα λογισμού στην πράξη, αλλά η κατασκευή μιας γλώσσας προγραμματισμού για επεξεργασία λιστών, με σκοπό την εφαρμογή της σε προβλήματα από την περιοχή της τεχνητής νοημοσύνης. Με βάση αυτό το σκεπτικό, η Lisp είναι μια επιτυχημένη γλώσσα προγραμματισμού, αλλά δεν μπορεί κατ' ουδένα τρόπο να θεωρηθεί σήμερα εκπρόσωπος της φιλοσοφίας του συναρτησιακού προγραμματισμού.

Όμως, παρά την ύπαρξη της Lisp από πολύ παλιά, έπρεπε, ουσιαστικά, να περάσουν τουλάχιστον δύο δεκαετίες από την εμφάνισή της, για να αρχίσουν να κατασκευάζονται «καθαρές» γλώσσες συναρτησιακού προγραμματισμού, που να βασίζονται, άλλες λιγότερο άλλες περισσότερο, στον λάμδα λογισμό. Τέτοιες γλώσσες, για να αναφέρουμε ενδεικτικά μερικές μόνο από αυτές, ήταν η Standard ML, η Hope, η SASL, η Miranda και η Haskell.

Στα κεφάλαια που ακολουθούν, θα παρουσιάσουμε τις βασικές δυνατότητες που παρέχονται από τη φιλοσοφία του συναρτησιακού προγραμματισμού. Αν και οι δυνατότητες αυτές μπορούν να εκφραστούν μέσω μιας πληθώρας από σύγχρονες συναρτησιακές γλώσσες προγραμματισμού, εμείς θα χρησιμοποιήσουμε τη γλώσσα Haskell, για τις ανάγκες της παρουσίασης, αφού είναι μια ευρύτερα αποδεκτή και διαδεδομένη γλώσσα σήμερα στην κοινότητα του συναρτησιακού προγραμματισμού.

Πριν αρχίσουμε, όμως, από το επόμενο κεφάλαιο, να βλέπουμε πώς μπορούμε να χρησιμοποιήσουμε τη Haskell για να λύσουμε διάφορα προβλήματα, ας μελετήσουμε λίγο καλύτερα την ιδέα του υπολογισμού που βασίζεται σε συναρτήσεις, από τη μαθηματική σκοπιά του.

Στα μαθηματικά, μια συνάρτηση f είναι ένα μέσο με το οποίο μπορούμε να απεικονίσουμε τα στοιχεία ενός συνόλου D, του **πεδίου ορισμού** της συνάρτησης, στα στοιχεία

ενός συνόλου R, του **πεδίου τιμών** της συνάρτησης. Συμβολικά:

$$f : D \longmapsto R$$

Παράδειγμα 9.1
Θα μπορούσαμε να ορίσουμε τη συνάρτηση $square$, η οποία απεικονίζει πραγματικούς αριθμούς στα τετράγωνά τους ($square : \mathcal{R} \longmapsto \mathcal{R}$), ως εξής:

$$square(x) = x^2$$

Επίσης, θα μπορούσαμε να διατυπώσουμε και μια συνάρτηση $sign$, με την οποία να ορίζουμε το πρόσημο ακέραιων αριθμών ($sign : \mathcal{Z} \longmapsto \{plus, zero, minus\}$). Δηλαδή:

$$sign(x) = \begin{cases} plus & \text{αν } x > 0 \\ zero & \text{αν } x = 0 \\ minus & \text{αν } x < 0 \end{cases}$$

Έχοντας, τώρα, δώσει τους κατάλληλους ορισμούς των συναρτήσεων που μας ενδιαφέρουν, μπορούμε να εφαρμόσουμε αυτές τις συναρτήσεις σε στοιχεία των πεδίων ορισμού τους, για να πάρουμε στοιχεία από τα πεδία τιμών τους. Έτσι, για τις συναρτήσεις $square$ και $sign$, είναι δυνατόν να γράψουμε $square(-3) = 9$, $square(1.7) = 2.89$, $sign(6) = plus$, $sign(0) = zero$ και $sign(-2) = minus$, όπως προκύπτει με απλές εφαρμογές των ορισμών. □

Βέβαια, πολλές φορές είναι χρήσιμο να ορίζουμε συναρτήσεις των οποίων τα πεδία ορισμού να αποτελούνται από ζευγάρια στοιχείων ή τριάδες κτλ., δηλαδή να είναι καρτεσιανά γινόμενα συνόλων που περιέχουν απλά στοιχεία. Επίσης, όταν δίνουμε τον ορισμό μιας συνάρτησης, μπορούμε να χρησιμοποιούμε άλλες συναρτήσεις (που έχουμε ήδη ορίσει ή που πρόκειται να ορίσουμε).

Παράδειγμα 9.2
Ο μέγιστος δύο πραγματικών αριθμών μπορεί να οριστεί με τη συνάρτηση $max2$ (έχοντας $max2 : \mathcal{R}^2 \longmapsto \mathcal{R}$), ως εξής:

$$max2(x, y) = \begin{cases} x & \text{αν } x > y \\ y & \text{σε άλλη περίπτωση} \end{cases}$$

Με τη βοήθεια της συνάρτησης $max2$, δεν είναι δύσκολο να ορίσουμε και μια συνάρτηση $max3$ (με $max3 : \mathcal{R}^3 \longmapsto \mathcal{R}$), για τον μέγιστο τριών πραγματικών αριθμών. Δηλαδή:

$$max3(x, y, z) = max2(x, max2(y, z))$$

Οπότε, $max2(-2.7, 1.4) = 1.4$ και $max3(5, 7, -3) = 7$. Το πρώτο προκύπτει με απευθείας εφαρμογή του ορισμού, αλλά δεν είναι δύσκολο να δούμε ότι το $max3(5, 7, -3)$ υπολογίζεται ως εξής:

$$max3(5, 7, -3) = max2(5, max2(7, -3)) = max2(5, 7) = 7$$

Ο απαιτούμενος υπολογισμός έγινε με εφαρμογή του ορισμού της συνάρτησης $max3$, ο οποίος απαιτεί εφαρμογές του ορισμού της συνάρτησης $max2$. □

Με βάση μόνο τα παραδείγματα που παρουσιάσαμε, ίσως είναι δύσκολο να δεχθούμε ότι ορισμοί συναρτήσεων μπορεί να αποτελέσουν την πρώτη ύλη για ένα μοντέλο υπολογισμού στο οποίο να βασίζεται μια γλώσσα προγραμματισμού που να μας διευκολύνει να λύνουμε προβλήματα από τον πραγματικό κόσμο. Ωστόσο, η πραγματική δύναμη των συναρτήσεων εντοπίζεται στο γεγονός ότι αυτές μπορεί να έχουν αναδρομικούς ορισμούς, δηλαδή είναι δυνατόν να ορίσουμε μια συνάρτηση μέσω του εαυτού της. Αυτό μας λύνει τα χέρια, δίνοντάς μας τη δυνατότητα να περιγράψουμε μέσω συναρτήσεων οποιαδήποτε υπολογιστική διαδικασία. Δείτε το επόμενο παράδειγμα:

Παράδειγμα 9.3
Η ακολουθία Fibonacci είναι μια ακολουθία που έχει πρώτο και δεύτερο όρο το 1, ενώ κάθε επόμενος όρος ισούται με το άθροισμα των δύο προηγούμενων όρων. Είναι αρκετά εύκολο να γράψουμε ένα πρόγραμμα σε κάποια διαδικαστική γλώσσα προγραμματισμού, όπως και σε κάποια γλώσσα λογικού προγραμματισμού, που να υπολογίζει τον n-οστό όρο της ακολουθίας Fibonacci, για δεδομένο n. Θα μπορούσαμε όμως, εξίσου εύκολα, να ορίσουμε μια συνάρτηση fib στους θετικούς ακέραιους αριθμούς ($fib : \mathcal{Z}_+ \longmapsto \mathcal{Z}_+$), η οποία να απεικονίζει την τάξη ενός όρου στην τιμή του ως εξής:

$$fib(n) = \begin{cases} 1 & \text{αν } n = 1 \\ 1 & \text{αν } n = 2 \\ fib(n-1) + fib(n-2) & \text{αν } n > 2 \end{cases}$$

Έτσι, μπορούμε να έχουμε, για παράδειγμα, $fib(5) = fib(4)+fib(3) = (fib(3)+fib(2))+(fib(2)+fib(1)) = ((fib(2)+fib(1))+1)+(1+1) = ((1+1)+1)+2 = (2+1)+2 = 3+2 = 5$.

Άσκηση 9.1
Για τις συναρτήσεις:

$succ : \mathcal{Z} \longmapsto \mathcal{Z}$	ο επόμενος ενός ακέραιου αριθμού
$hypot : \mathcal{R}_+^2 \longmapsto \mathcal{R}_+$	το μήκος της υποτείνουσας ενός ορθογώνιου τριγώνου με δεδομένες κάθετες πλευρές
$signmax4 : \mathcal{R}^4 \longmapsto \mathcal{R}$	το πρόσημο του μεγίστου τεσσάρων πραγματικών αριθμών
$gcd : \mathcal{Z}_+^2 \longmapsto \mathcal{Z}_+$	ο μέγιστος κοινός διαιρέτης δύο θετικών ακέραιων αριθμών

συμπληρώστε κατάλληλα τους ορισμούς που ακολουθούν:

$$\begin{aligned} succ(x) &= x \boxed{\textbf{A}} 1 \\ hypot(x,y) &= \sqrt{\boxed{\textbf{B}} + y^2} \\ signmax4(x,y,z,w) &= sign(\boxed{\textbf{C}}(max2(x,y), max2(\boxed{\textbf{D}}, w))) \\ gcd(x,y) &= \begin{cases} x & \text{αν } \boxed{\textbf{E}} \\ gcd(x, \boxed{\textbf{F}}) & \text{αν } x < y \\ \boxed{\textbf{G}}(x-y, \boxed{\textbf{H}}) & \text{αν } x > y \end{cases} \end{aligned}$$

Ποιες είναι οι τιμές $succ(-4)$, $hypot(5,12)$, $signmax4(-3,0,-1,2)$ και $gcd(24,9)$;

Άσκηση 9.2
Δώστε τον ορισμό μιας συνάρτησης για το παραγοντικό ενός μη αρνητικού ακέραιου αριθμού. Πώς μπορεί να υπολογιστεί το παραγοντικό του 6, με τη βοήθεια αυτού του ορισμού;

Απαντήσεις ασκήσεων

Απάντηση άσκησης 9.1
Οι σωστές απαντήσεις είναι:

A:	$+$	**E**:	$x = y$
B:	x^2	**F**:	$y - x$
C:	$max2$	**G**:	gcd
D:	z	**H**:	y

Όσον αφορά την εφαρμογή των συναρτήσεων στις δεδομένες τιμές, έχουμε:

$$\begin{array}{ll} succ(-4) & = (-4) + 1 = -3 \\ hypot(5, 12) & = \sqrt{5^2 + 12^2} = \sqrt{25 + 144} = \sqrt{169} = 13 \\ signmax4(-3, 0, -1, 2) & = sign(max2(max2(-3, 0), max2(-1, 2))) \\ & = sign(max2(0, 2)) = sign(2) = plus \\ gcd(24, 9) & = gcd(24 - 9, 9) = gcd(15, 9) = gcd(15 - 9, 9) = gcd(6, 9) \\ & = gcd(6, 9 - 6) = gcd(6, 3) = gcd(6 - 3, 3) = gcd(3, 3) = 3 \end{array}$$

Απάντηση άσκησης 9.2
Θα μπορούσαμε να ορίσουμε το παραγοντικό ενός μη αρνητικού ακέραιου αριθμού μέσω μιας συνάρτησης $fact$ (με $fact : \mathcal{N} \longmapsto \mathcal{N}$), ως εξής:

$$fact(n) = \begin{cases} 1 & \text{αν } n = 0 \\ n \cdot fact(n-1) & \text{αν } n > 0 \end{cases}$$

Για το $fact(6)$, έχουμε:

$$\begin{array}{ll} fact(6) & = 6 \cdot fact(6-1) = 6 \cdot fact(5) = 6 \cdot (5 \cdot fact(5-1)) = (6 \cdot 5) \cdot fact(4) \\ & = 30 \cdot (4 \cdot fact(4-1)) = (30 \cdot 4) \cdot fact(3) = 120 \cdot (3 \cdot fact(3-1)) \\ & = (120 \cdot 3) \cdot fact(2) = 360 \cdot (2 \cdot fact(2-1)) = (360 \cdot 2) \cdot fact(1) \\ & = 720 \cdot (1 \cdot fact(1-1)) = (720 \cdot 1) \cdot fact(0) = 720 \cdot 1 = 720 \end{array}$$

Προβλήματα

Πρόβλημα 9.1
Δώστε δύο εναλλακτικούς ορισμούς συνάρτησης για τον υπολογισμό του αθροίσματος όλων των ακέραιων αριθμών από το 1 έως το n. Ο ένας ορισμός να είναι αναδρομικός, ενώ ο άλλος όχι.

Πρόβλημα 9.2
Ορίστε συνάρτηση που να υπολογίζει το ελάχιστο κοινό πολλαπλάσιο δύο θετικών ακέραιων αριθμών.

Πρόβλημα 9.3
Ορίστε συνάρτηση που να υπολογίζει το άθροισμα των ψηφίων δεδομένου θετικού ακέραιου αριθμού.

Πρόβλημα 9.4
Ορίστε συναρτήσεις επάνω στο σύνολο $B = \{T, F\}$ (αληθές/ψευδές), για τις λογικές πράξεις της άρνησης, της σύζευξης, της διάζευξης, της συνεπαγωγής και της ισοδυναμίας.

Βιβλιογραφικές αναφορές

[1] J. Bakus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *CACM*, 21(8), 613-641, 1978.

[2] A. J. Field and P. G. Harrison, *Functional Programming*, Addison Wesley, 1988.

[3] B. Goldberg, Functional Programming Languages, *ACM Comput. Surv.*, 28(1), 249-251, 1996.

[4] M. Hanus and H. Kuchen, Integration of Functional and Logic Programming, *ACM Comput. Surv.*, 28(2), 306-308, 1996.

Κεφάλαιο 10
Η γλώσσα συναρτησιακού προγραμματισμού Haskell – Τα βασικά

Σύνοψη

Στο κεφάλαιο αυτό παρουσιάζεται ο τρόπος με τον οποίο μπορούμε να γράφουμε προγράμματα Haskell, διατυπώνοντας ορισμούς συναρτήσεων, συμπεριλαμβανομένων αναδρομικών ορισμών, και να υπολογίζουμε εκφράσεις που εμπλέκουν τους ορισμούς αυτούς. Επίσης, γίνεται αναφορά στις πλειάδες και στις λίστες, δύο πολύ χρήσιμες δομές δεδομένων στη Haskell. Επιπλέον, εξετάζονται η έννοια του πολυμορφισμού και ο τρόπος με τον οποίο μπορούμε να ορίζουμε στη Haskell συναρτήσεις με παραμετρικούς τύπους. Ακόμα, παρουσιάζεται μια σειρά από συναρτήσεις ορισμένες στο πρελούδιο, που είναι μια βιβλιοθήκη της Haskell. Τέλος, αναφέρονται οι κυριότερες περιοχές εφαρμογών, στις οποίες καλό είναι να χρησιμοποιείται η Haskell.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης δεν απαιτείται να έχει συγκεκριμένες γνώσεις.

10.1 Ορισμός συναρτήσεων και υπολογισμός εκφράσεων

Στο Κεφάλαιο 9 είδαμε κάποια εισαγωγικά στοιχεία για τον συναρτησιακό προγραμματισμό, δηλαδή εκείνη τη φιλοσοφία του δηλωτικού προγραμματισμού κατά την οποία διατυπώνουμε αξιώματα για όσα ισχύουν στον κόσμο μας υπό τη μορφή **ορισμών συναρτήσεων**. Όπως ήδη αναφέραμε, υπάρχουν διάφορες γλώσσες συναρτησιακού προγραμματισμού. Εμείς θα προσπαθήσουμε να δώσουμε, σε αυτό και στο επόμενο κεφάλαιο, τις βασικές ιδέες του συναρτησιακού προγραμματισμού μέσω μιας σύγχρονης και εξαιρετικά ισχυρής γλώσσας, της Haskell [1, 2].

Η Haskell πήρε το όνομά της προς τιμήν του Haskell B. Curry, ενός από τους σημαντικότερους πρωτοπόρους του λάμδα λογισμού και, κατ' επέκταση, του συναρτησιακού προγραμματισμού. Ο μεταγλωττιστής της Haskell ονομάζεται GHC και είναι ελεύθερα διαθέσιμος από την επίσημη ιστοσελίδα της γλώσσας `http://www.haskell.org`. Στην ιστοσελίδα αυτή, μπορούμε να βρούμε, εκτός του μεταγλωττιστή, διάφορα στοιχεία για τη Haskell. Για να μπορέσετε να αποκτήσετε εμπειρία στον προγραμματισμό με τη Haskell, πρέπει απαραιτήτως, ταυτόχρονα με τη μελέτη σας, να χρησιμοποιείτε τη γλώσσα και στον υπολογιστή. Γι' αυτόν το λόγο, είναι πολύ καλή ιδέα να «κατεβάσετε» από το διαδίκτυο το μεταγλωττιστή GHC και να τον εγκαταστήσετε σε έναν υπολογιστή στον οποίο θα έχετε πρόσβαση, για να μπορείτε να δείτε πώς θα δουλέψουν στην πράξη τα παραδείγματα που

θα παρουσιάσουμε, αλλά και να λύσετε ευκολότερα τις ασκήσεις και τα προβλήματα που θα σας ζητηθούν.

Ας δούμε, όμως, πώς μπορούμε να ορίσουμε στη Haskell απλές συναρτήσεις μέσω μερικών παραδειγμάτων.

Παράδειγμα 10.1
Έστω ότι θέλουμε να ορίσουμε μια συνάρτηση που να υπολογίζει το τετράγωνο ενός ακέραιου αριθμού. Θα γράφαμε τότε στη Haskell το εξής πρόγραμμα:

```
--     Υπολογισμός του τετραγώνου ενός ακεραίου
square :: Int -> Int
square x = x*x
```

Στο πρόγραμμα αυτό, η πρώτη γραμμή είναι ένα σχόλιο. Οτιδήποτε υπάρχει ύστερα από ένα «`--`» και μέχρι το τέλος της γραμμής είναι, για τη Haskell, σχόλιο. Αν θέλουμε να περιλαμβάνουμε στα προγράμματά μας σχόλια που να εκτείνονται σε περισσότερες της μιας γραμμές, μπορούμε, αντί να αρχίζουμε κάθε γραμμή με ένα «`--`», να περικλείσουμε όλο το τμήμα σχολίων μεταξύ ενός «`{-`» και ενός «`-}`». Η δεύτερη γραμμή στο πρόγραμμά μας προαναγγέλλει τον ορισμό μιας συνάρτησης, της `square`, της οποίας τόσο το πεδίο ορισμού όσο και το πεδίο τιμών είναι το σύνολο των ακεραίων (`Int`). Τέλος, στην τρίτη γραμμή, βρίσκεται ο πραγματικός ορισμός της συνάρτησης που μας ενδιαφέρει, ο οποίος εμπλέκει και μια **τυπική παράμετρο** της απόλυτης επιλογής μας, τη `x`, στην προκειμένη περίπτωση. □

Έχοντας εφοδιάσει τη Haskell με τον ορισμό της συνάρτησης `square`, μπορούμε πλέον να ζητήσουμε να εφαρμοστεί η συνάρτηση αυτή σε δεδομένες τιμές εισόδου, ως εξής:

```
*Main> square 7
49
*Main> square (-12)
144
*Main>
```

Η εφαρμογή αυτή γίνεται μέσω κατάλληλου **υπολογισμού εκφράσεων**, όπως η έκφραση `square 7`, που υπολογίζεται σε `49`, με τη βοήθεια του ορισμού της συνάρτησης `square`.[1]

Στο Παράδειγμα 10.1 είδαμε ότι βασικό συστατικό του ορισμού μιας συνάρτησης στη Haskell είναι η δήλωση των **τύπων** του πεδίου ορισμού και του πεδίου τιμών της συνάρτησης. Στη Haskell, εκτός από τον τύπο `Int`, που περιλαμβάνει τους ακέραιους αριθμούς, οι άλλοι βασικοί τύποι είναι οι `Float`, `Bool` και `Char`, για τους πραγματικούς αριθμούς, τις λογικές τιμές (`True` και `False`) και τους χαρακτήρες, αντίστοιχα. Όπως παρατηρείτε, τα ονόματα των τύπων αρχίζουν με κεφαλαίο χαρακτήρα. Αυτό επιβάλλεται συντακτικά από τη Haskell. Επίσης, τα ονόματα των συναρτήσεων, αλλά και των τυπικών παραμέτρων που χρησιμοποιούνται στους ορισμούς των συναρτήσεων, πρέπει να αρχίζουν με μικρό χαρακτήρα (π.χ. `square`, `x` κτλ.).

[1] Περικλείουμε τον αρνητικό ακέραιο `-12` μέσα σε παρενθέσεις για να αποφευχθεί η σύγχυση του προσήμου − με το σύμβολο της αφαίρεσης. Επίσης, ας μη μας προβληματίσει ιδιαίτερα αυτό το «`*Main>`», που υπάρχει πριν από την έκφραση της οποίας ζητάμε τον υπολογισμό. Μας αρκεί, για τις ανάγκες του παρόντος συγγράμματος, να το θεωρούμε προτροπή της Haskell για την υποβολή εκφράσεων προς υπολογισμό.

Παράδειγμα 10.2
Μπορούμε να ορίσουμε μια συνάρτηση που να ελέγχει αν δεδομένος χαρακτήρας είναι κεφαλαίος λατινικός, ως εξής:

```
uppercase :: Char -> Bool
uppercase c = (c >= 'A') && (c <= 'Z')
```

Η συνάρτηση αυτή επιστρέφει `True` ή `False`, δηλαδή μια λογική τιμή τύπου `Bool`, με βάση την εφαρμογή της σε ένα χαρακτήρα (τύπου `Char`) που είναι στον πίνακα χαρακτήρων μεταξύ `'A'` και `'Z'` ή όχι. □

Έτσι, με αυτόν τον ορισμό, μπορούμε να έχουμε:

```
*Main> uppercase 'F'
True
*Main> uppercase 'q'
False
*Main> uppercase '6'
False
*Main>
```

Στο Παράδειγμα 10.2 είδαμε και τη χρήση **τελεστών**, για τον ορισμό συναρτήσεων. Τα σύμβολα >= και <= είναι **σχεσιακοί τελεστές**, που χρησιμοποιούνται για τη διατύπωση εκφράσεων, με σκοπό να ελεγχθεί αν μια τιμή είναι μεγαλύτερη ή ίση (για το >=) ή μικρότερη ή ίση (για το <=) από κάποια άλλη, με δεδομένη κάποια σχέση διάταξης στα στοιχεία του εμπλεκόμενου τύπου, και την επιστροφή κατάλληλης λογικής τιμής. Άλλοι σχεσιακοί τελεστές είναι οι > (μεγαλύτερο), < (μικρότερο), == (ίσο) και /= (διάφορο). Το `&&` είναι ένας **λογικός τελεστής**, για το λογικό «ΚΑΙ», που χρησιμοποιείται για την κατασκευή σύνθετων λογικών εκφράσεων. Η λογική τιμή μιας τέτοιας σύνθετης λογικής έκφρασης είναι `True`, μόνο όταν και οι δύο (συζευγμένες) λογικές υποεκφράσεις έχουν τιμή `True`. Οι άλλοι λογικοί τελεστές στη Haskell είναι το `||` (για το λογικό «Ή»), που δίνει `True`, όταν τουλάχιστον μία από τις (διαζευγμένες) λογικές υποεκφράσεις έχει τιμή `True`, και το `not` (για τη λογική άρνηση), που αντιστρέφει τη λογική τιμή της έκφρασης στην οποία εφαρμόζεται (το `True` γίνεται `False` και το `False` γίνεται `True`).

Παράδειγμα 10.3
Ας δούμε, τώρα, πώς θα ορίζαμε στη Haskell τις συναρτήσεις $max2$ και $max3$ του Παραδείγματος 9.2. Εδώ έχουμε δύο θέματα να προσέξουμε, τον τρόπο ορισμού συναρτήσεων που να εφαρμόζονται σε περισσότερες της μιας εισόδους και τον τρόπο δημιουργίας ορισμών στους οποίους πρέπει να λάβουμε υπόψη μας διάφορες περιπτώσεις για τις εισόδους μιας συνάρτησης. Έχουμε, λοιπόν:

```
max2 :: Float -> Float -> Float
max2 x y
  | x > y      = x
  | otherwise  = y

max3 :: Float -> Float -> Float -> Float
max3 x y z = max2 x (max2 y z)
```

□

Μια συνάρτηση, έστω με όνομα `fun`, που έχει `k` τυπικές παραμέτρους για την αναπαράσταση των εισόδων της, έστω τύπων `t1`, `t2`, ..., `tk`, και δίνει αποτέλεσμα τύπου `t`, δηλώνεται στη Haskell ως:

```
fun :: t1 -> t2 -> ... -> tk -> t
```

όπως φαίνεται στους ορισμούς του Παραδείγματος 10.3. Επίσης, πολύ συχνά, έχουμε ορισμούς συναρτήσεων που πρέπει να καλύψουν διάφορες περιπτώσεις δεδομένων εισόδου. Αυτό επιτυγχάνεται στη Haskell μέσω των λεγόμενων **φρουρών**. Στον ορισμό της συνάρτησης `max2`, στο Παράδειγμα 10.3, υπάρχει ο φρουρός `x > y`, ο οποίος αποτελεί μια τέτοια συνθήκη ώστε, όταν αυτή είναι `True`, η τιμή της συνάρτησης να ισούται με `x`. Ένας ειδικός φρουρός είναι αυτός που μπορούμε να βάλουμε τελευταίο σε μια σειρά από φρουρούς, μέσω της ειδικής λέξης `otherwise`, ο οποίος γίνεται `True` όταν όλοι οι προηγούμενοι είναι `False`, για τις δεδομένες τιμές εισόδου της συνάρτησης.

Δοκιμάστε τώρα να αντιμετωπίσετε τις ασκήσεις που ακολουθούν, για να ελέγξετε αν έχετε κατανοήσει όσα συζητήσαμε μέχρι στιγμής.

Άσκηση 10.1

Στο Παράδειγμα 10.3 είδαμε τον ορισμό στη Haskell της συνάρτησης `max3`, για το μέγιστο τριών πραγματικών αριθμών, διατυπωμένο μέσω της αντίστοιχης συνάρτησης για το μέγιστο δύο αριθμών, της `max2`. Θα μπορούσαμε, όμως, να είχαμε ορίσει τη `max3` απευθείας και όχι με τη βοήθεια της `max2`. Συμπληρώστε τον ακόλουθο ορισμό, σύμφωνα με αυτήν την προσέγγιση:

```
max3 :: Float -> Float -> Float -> Float
max3 x y z
  | x >= y && x >= [ ]   =  x
  | [ ] >= [ ]           =  y
  | otherwise            =  [ ]
```

Έχοντας δεδομένο αυτόν τον ορισμό, εξηγήστε πώς είναι δυνατόν να υπολογιστούν οι εκφράσεις `max3 2.7 1.2 1.8` και `max3 1.3 1.9 3.1`.

Άσκηση 10.2

Η αποκλειστική διάζευξη είναι μια λογική πράξη επάνω σε δύο εισόδους, που δίνει αποτέλεσμα «αληθές», όταν τουλάχιστον ένας από τους δύο διαζευκτέους είναι «αληθής», αλλά όχι και οι δύο. Συμπληρώστε κατάλληλα τον επόμενο ορισμό της συνάρτησης `exOr`, έτσι ώστε αυτός να υλοποιεί σε Haskell την αποκλειστική διάζευξη:

```
exOr :: [          ] -> Bool
exOr x y = (x [ ] y) && [ ] ( [      ] )
```

Ποιες είναι οι τιμές των εκφράσεων `exOr True True` και `exOr False True`, και γιατί;

Άσκηση 10.3

Ορίστε σε Haskell μια συνάρτηση `threeDifferent` που να παίρνει την τιμή `True` όταν τρεις δεδομένοι ακέραιοι, τους οποίους δέχεται ως εισόδους, είναι διάφοροι μεταξύ τους.

Άσκηση 10.4

Αναδιατυπώστε με απλούστερο τρόπο τον ορισμό της συνάρτησης `complicated` που δίνεται στη συνέχεια, χωρίς να χρησιμοποιήσετε φρουρούς:

```
complicated :: Int -> Int -> Int -> Bool
complicated x y z
  | x > z       = True
  | y >= x      = False
  | otherwise   = True
```

Για να αντιμετωπίσετε τη συγκεκριμένη άσκηση, δείτε ποια είναι η πιο συνηθισμένη τιμή που παίρνει η συνάρτηση και πότε παραβιάζεται αυτή η γενική συμπεριφορά. □

Θα κλείσουμε αυτήν την ενότητα με ένα θέμα που είναι πολύ σημαντικό. Όπως ισχύει και για κάθε άλλη σύγχρονη γλώσσα προγραμματισμού, έτσι και στη Haskell, μας προσφέρεται η δυνατότητα να εκφράσουμε αναδρομή. Αυτό γίνεται με τη διατύπωση **αναδρομικών ορισμών**, στους οποίους, τουλάχιστον για κάποιες περιπτώσεις, η συνάρτηση που μας ενδιαφέρει ορίζεται μέσω του ίδιου της του εαυτού.

Παράδειγμα 10.4
Στην Άσκηση 9.2 σας ζητήθηκε να δώσετε τη μαθηματική συνάρτηση που ορίζει το παραγοντικό ενός μη αρνητικού ακέραιου αριθμού. Ας δούμε πώς μπορούμε να διατυπώσουμε τον ορισμό αυτό στη Haskell:

```
fact :: Int -> Int
fact n
  | n == 0   = 1
  | n > 0    = n * fact (n-1)
```

□

Έχοντας δουλέψει αρκετά με το χειρισμό της αναδρομής στην Prolog, δεν θα συναντήσετε ιδιαίτερα προβλήματα με αυτό το θέμα στο περιβάλλον της Haskell. Παρότι στην Prolog η αναδρομή εμφανίζεται στη διατύπωση συνεπαγωγών, ενώ στη Haskell στον ορισμό συναρτήσεων, δεν υπάρχουν ουσιώδεις διαφορές ανάμεσα στις δύο προγραμματιστικές φιλοσοφίες, παρά μόνο παραλλαγές συντακτικής φύσης. Αυτό, άλλωστε, θα έπρεπε να ήταν αναμενόμενο, αφού τόσο ο λογικός όσο και ο συναρτησιακός προγραμματισμός εμπίπτουν στο γενικότερο πλαίσιο του δηλωτικού προγραμματισμού, ο οποίος διέπεται από μια ενιαία αντίληψη αντιμετώπισης του κόσμου.

Άσκηση 10.5
Διατυπώστε στη Haskell τους ορισμούς για την ακολουθία Fibonacci, του Παραδείγματος 9.3, και για τον μέγιστο κοινό διαιρέτη δύο θετικών ακεραίων, που αναφέραμε, μεταξύ άλλων ορισμών, στην Άσκηση 9.1.

10.2 Πλειάδες και λίστες

Μέχρι στιγμής, είδαμε ότι τα δεδομένα που μπορούμε να χρησιμοποιήσουμε στη Haskell είναι ακέραιοι και πραγματικοί αριθμοί (τύπου `Int` και `Float`, αντίστοιχα), οι λογικές τιμές `True` και `False` (τύπου `Bool`) και οι χαρακτήρες (τύπου `Char`), οι οποίοι, συντακτικά, περικλείονται σε απλά εισαγωγικά. Σε πολλές εφαρμογές, όμως, είναι απαραίτητο να έχουμε στη διάθεσή μας και πιο σύνθετα δεδομένα. Αυτή τη δυνατότητα μας την παρέχει η Haskell, μέσω των **πλειάδων** και των **λιστών**.

Τόσο οι πλειάδες όσο και οι λίστες είναι συλλογές από πιο στοιχειώδη δεδομένα, αλλά έχουν διαφορετική χρησιμότητα και διαφορετικές ιδιότητες. Μια πλειάδα έχει προκαθορι-

σμένο αριθμό συστατικών, καθένα από τα οποία είναι δεδομένου τύπου, όχι κατ' ανάγκη ίδιου για όλα τα συστατικά. Από την άλλη πλευρά, μια λίστα έχει αυθαίρετο αριθμό από μέλη, εκ των οποίων όλα είναι, όμως, του ίδιου τύπου.

Παράδειγμα 10.5
Ένα σημείο στον δισδιάστατο χώρο μπορεί να παρασταθεί από μια πλειάδα που αποτελείται από δύο πραγματικούς αριθμούς, τις x και y συντεταγμένες του σημείου. Ένα τέτοιο σημείο είναι, για παράδειγμα, αυτό που παριστάνεται από την πλειάδα `(2.3,1.7)`, η οποία είναι δεδομένο του τύπου `(Float,Float)` στη Haskell. Είναι καλή ιδέα να ορίζουμε στα προγράμματά μας ονόματα για τους τύπους πλειάδων που σκοπεύουμε να χρησιμοποιήσουμε, γιατί έτσι βελτιώνουμε την αναγνωσιμότητά τους. Μπορούμε, δηλαδή, να γράψουμε στη Haskell:

```
type Point_2D = (Float,Float)
```

και να χρησιμοποιούμε πλέον τον τύπο `Point_2D`, σε ορισμούς συναρτήσεων, για παράδειγμα.

Στις πλειάδες του τύπου `Point_2D`, και τα δύο συστατικά τους είναι τύπου `Float`. Αν όμως τα σημεία στο επίπεδο ήταν άσπρα ή μαύρα και μας ενδιέφερε να αναφέρουμε, για κάθε σημείο, και αυτήν την ιδιότητα, θα μπορούσαμε να ορίσουμε και να χρησιμοποιήσουμε έναν τύπο `BW_Point_2D`, με τρία συστατικά, ως εξής:

```
type BW_Point_2D = (Float,Float,Bool)
```

Είναι δυνατόν να έχουμε καθορίσει ότι το τρίτο συστατικό (τύπου `Bool`) σε μια πλειάδα τέτοιου τύπου έχει τιμή `True`, όταν το σημείο είναι άσπρο, ή `False`, όταν το σημείο είναι μαύρο.

Παράδειγμα 10.6
Εάν θέλαμε να παραστήσουμε στη Haskell ένα μονοπάτι, δηλαδή μια ακολουθία, από (άσπρα ή μαύρα) σημεία, όπως αυτά του Παραδείγματος 10.5, θα μπορούσαμε να χρησιμοποιήσουμε μια λίστα από τέτοια σημεία. Ορίζοντας και τον τύπο:

```
type BW_Path = [BW_Point_2D]
```

μπορούμε να χρησιμοποιήσουμε, για παράδειγμα, και το δεδομένο:

```
[(1.4,-3.3,True),(0.0,0.0,False),(7.1,1.1,False),(-1.2,2.1,True)]
```

το οποίο παριστάνει ένα μονοπάτι (τύπου `BW_Path`) από τέσσερα σημεία στον δισδιάστατο χώρο. Είναι δυνατόν μια λίστα να μην περιέχει κανένα μέλος, οπότε είναι η **κενή λίστα** `[]`. □

Ένας πολύ χρήσιμος τύπος είναι η λίστα από χαρακτήρες, που αναφέρεται, πολλές φορές, και ως **συμβολοσειρά**. Γι' αυτόν το λόγο, ο συγκεκριμένος τύπος είναι ήδη ορισμένος στη Haskell, με την ονομασία `String`. Δηλαδή, μπορούμε να θεωρούμε ότι σε κάθε σύστημα Haskell είναι ενσωματωμένη και η δήλωση:

```
type String = [Char]
```

Μάλιστα, η Haskell δίνει και τη δυνατότητα μιας πιο συνεπτυγμένης γραφής για τα δεδομένα του τύπου `String`. Για παράδειγμα, η λίστα `['p','o','t','a','t','o']` μπορεί, ισοδύναμα, να γραφεί και ως η συμβολοσειρά `"potato"`.

Έχοντας τώρα εισαγάγει τις έννοιες των πλειάδων και των λιστών, μπορούμε, πλέον, να ορίζουμε και συναρτήσεις που εμπλέκουν δεδομένα τέτοιας μορφής.

Παράδειγμα 10.7
Μπορούμε να ορίσουμε μια συνάρτηση `swap`, η οποία να αντιστρέφει τη σειρά στα μέλη μιας πλειάδας με συστατικά δύο ακεραίους:

```
swap :: (Int,Int) -> (Int,Int)
swap (x,y) = (y,x)
```

Παράδειγμα 10.8
Μπορούμε, επίσης, να ορίσουμε μια συνάρτηση `sumdiff`, η οποία να δέχεται ορίσματα δύο πραγματικούς αριθμούς και να υπολογίζει το άθροισμα και τη διαφορά τους, τοποθετώντας τα αποτελέσματα ως συστατικά μιας πλειάδας:

```
sumdiff :: Float -> Float -> (Float,Float)
sumdiff x y = (x+y,x-y)
```

Προσέξτε τη διαφορά ανάμεσα στη συνάρτηση `sumdiff` και στην `other_sumdiff`, που ορίζουμε στη συνέχεια:

```
other_sumdiff :: (Float,Float) -> (Float,Float)
other_sumdiff (x,y) = (x+y,x-y)
```

Η `sumdiff` εφαρμόζεται σε δύο εισόδους, που είναι πραγματικοί αριθμοί, ενώ η δεύτερη συνάρτηση, η `other_sumdiff`, εφαρμόζεται σε μία είσοδο, που είναι μια πλειάδα, η οποία έχει συστατικά δύο πραγματικούς αριθμούς. Οι δύο αυτές προσεγγίσεις είναι εντελώς διαφορετικές μεταξύ τους. □

Ας δούμε, όμως, πώς θα μπορούσαμε να χρησιμοποιήσουμε την ιδέα της αναδρομής, για να επεξεργαστούμε τα στοιχεία μιας λίστας.

Παράδειγμα 10.9
Έστω ότι θέλουμε να υπολογίσουμε το άθροισμα των στοιχείων μιας λίστας, τα οποία είναι ακέραιοι αριθμοί. Αυτό μπορούμε να το κάνουμε ορίζοντας τη συνάρτηση `sum`, ως εξής:

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Στο παράδειγμα αυτό έχουμε δύο ενδιαφέροντα θέματα να συζητήσουμε. Πρώτον, βλέπουμε ότι η συνάρτηση ορίζεται με δύο τρόπους και, δεύτερον, χρησιμοποιούμε μια έκφραση, την `x:xs`, την οποία δεν έχουμε ξανασυναντήσει μέχρι τώρα, τουλάχιστον με αυτήν τη σύνταξη.[2] Η συνάρτηση `sum` ορίζεται, αφενός, για την περίπτωση της κενής λίστας, όποτε το άθροισμα των στοιχείων της είναι `0`, και αφετέρου, για την περίπτωση μιας λίστας

[2]Η έκφραση αυτή είναι αντίστοιχη της λίστας `[X|Xs]` στην Prolog.

που έχει τη μορφή `x:xs`. Αυτό είναι μια έκφραση που παριστάνει τη λίστα που έχει πρώτο στοιχείο (**κεφαλή**) το `x` και υπόλοιπα στοιχεία αυτά της λίστας `xs` (**ουρά**).

Πώς όμως αποφασίζει η Haskell ποιον από τους δύο τύπους θα χρησιμοποιήσει, για να υπολογίσει το άθροισμα των στοιχείων δεδομένης λίστας; Η Haskell χρησιμοποιεί μια διαδικασία που ονομάζεται **ταίριασμα προτύπων**, με βάση την οποία γίνεται απόπειρα να ταιριάξει η συγκεκριμένη είσοδος με την οποία καλείται μια συνάρτηση με τα πρότυπα που περιγράφονται από τις τυπικές παραμέτρους στους εναλλακτικούς τύπους του ορισμού της συνάρτησης. Αυτοί οι εναλλακτικοί τύποι εξετάζονται ένας προς έναν, με τη σειρά που είναι γραμμένοι, και ο πρώτος εφαρμόσιμος, με βάση το αποτέλεσμα του ταιριάσματος προτύπων, είναι αυτός που χρησιμοποιείται τελικά.

Έτσι, στην περίπτωση της συνάρτησης `sum`, αν αυτή κληθεί με είσοδο την κενή λίστα `[]`, θα εφαρμοστεί ο τύπος `sum [] = 0` και θα πάρουμε αποτέλεσμα το `0`. Αν αυτή κληθεί με είσοδο κάποια λίστα που είναι μη κενή, επειδή δεν θα είναι δυνατόν να ταιριάξει η συγκεκριμένη λίστα με το πρότυπο `[]` του πρώτου τύπου, θα γίνει απόπειρα να ταιριάξει με το πρότυπο `x:xs` του δεύτερου τύπου. Αυτή η απόπειρα θα είναι επιτυχής, οπότε θα εφαρμοστεί, για τον υπολογισμό του αποτελέσματος, ο δεύτερος τύπος, έχοντας ως `x` την κεφαλή της λίστας εισόδου και ως `xs` την ουρά της. Τελικά, το ζητούμενο αποτέλεσμα θα είναι το άθροισμα της κεφαλής με το άθροισμα των στοιχείων της ουράς (αναδρομικός υπολογισμός).

Άσκηση 10.6
Αντιστοιχίστε τα δεδομένα του πίνακα **Δ** στους τύπους του πίνακα **Τ**.

Δ	
α	`(2,True)`
β	`["ab","cde"]`
γ	`([2.3,5.6],'a',(1,2))`
δ	`("ab","cde")`
ε	`[('a',False),('b',True)]`

Τ	
1	`([Float],Char,(Int,Int))`
2	`([Char],[Char])`
3	`(Int,Bool)`
4	`[(Char,Bool)]`
5	`[String]`

Άσκηση 10.7
Ορίστε σε Haskell δύο συναρτήσεις `count_BPoints` και `count_WPoints`, οι οποίες να υπολογίζουν πόσα μαύρα και πόσα άσπρα σημεία, αντίστοιχα, υπάρχουν σε ένα μονοπάτι (τύπου `BW_Path`, όπως αυτός ορίστηκε στο Παράδειγμα 10.6) από σημεία.

10.3 Πολυμορφισμός

Έχοντας εισαγάγει την έννοια των λιστών στη Haskell και δεδομένου ότι μια λίστα δεν έχει προκαθορισμένο αριθμό στοιχείων, καταλαβαίνουμε πως θα ήταν ιδιαίτερα χρήσιμο αν ορίζαμε μια συνάρτηση που να υπολογίζει το πλήθος των στοιχείων μιας λίστας.

Παράδειγμα 10.10
Έστω ότι μας ενδιαφέρει να μετράμε πόσα στοιχεία έχουν λίστες από ακεραίους. Μπορούμε, γι' αυτόν το σκοπό, να ορίσουμε τη συνάρτηση `listlen` πολύ απλά, ως εξής:

```
listlen :: [Int] -> Int
listlen []     = 0
listlen (_:xs) = 1 + listlen xs
```

Φυσικά, αν θέλαμε η συνάρτησή μας να μετράει τα στοιχεία λιστών από πραγματικούς αριθμούς, θα δίναμε τον ίδιο ορισμό, εκτός από τον τύπο της συνάρτησης, που θα ήταν πλέον:

```
listlen :: [Float] -> Int
```
□

Από το Παράδειγμα 10.10, συμπεραίνουμε ότι, για να ορίσουμε τη συνάρτηση `listlen`, θα έπρεπε να δηλώσουμε ρητά τον τύπο των στοιχείων των λιστών στις οποίες εφαρμόζεται. Αν θέλαμε να δουλέψουμε με περισσότερα του ενός είδη λιστών, όσον αφορά τον τύπο των στοιχείων τους, δηλαδή θα ήμασταν αναγκασμένοι να ορίσουμε διαφορετικές (επί της ουσίας, όμως, ίδιες) συναρτήσεις για κάθε περίπτωση;

Η απάντηση στην προηγούμενη ερώτηση είναι, ευτυχώς, αρνητική. Η Haskell υποστηρίζει την ιδέα του **πολυμορφισμού**, δηλαδή τη δυνατότητα να ορίζουμε συναρτήσεις με **παραμετρικούς τύπους**. Έτσι, μπορούμε να γράφουμε ορισμούς συναρτήσεων που είναι ευρύτερα εφαρμόσιμοι, όπως ο ορισμός της συνάρτησης `length`, για τον υπολογισμό του πλήθους των στοιχείων μιας λίστας, ανεξάρτητα από τον τύπο των στοιχείων της λίστας, που δίνεται στο επόμενο παράδειγμα.

Παράδειγμα 10.11
Μπορούμε να ορίσουμε στη Haskell μια συνάρτηση `length` που να υπολογίζει πόσα στοιχεία έχει μια λίστα, ως εξής:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Το σύμβολο `a` σε αυτόν τον ορισμό είναι μια **μεταβλητή τύπου**, που παριστάνει οποιονδήποτε τύπο.[3] Συνεπώς, ο ορισμός αυτός μπορεί να εφαρμοστεί για οποιαδήποτε λίστα, χωρίς να έχει σημασία αν τα στοιχεία της είναι τύπου `Int`, `Float`, `Bool`, `Char`, ή πλειάδες με συστατικά οποιωνδήποτε τύπων, ή ακόμα και λίστες, όπως τύπου `String`. □

Εδώ θα πρέπει να σημειώσουμε ότι είναι δυνατόν να έχουμε περισσότερες της μιας μεταβλητές τύπου στον ορισμό του τύπου μιας συνάρτησης. Προσέξτε όμως τη διαφορά ανάμεσα σε μια συνάρτηση `fun1` και μια `fun2`, οι οποίες έχουν δηλωθεί ως:

```
fun1 :: [a] -> [b] -> (a,b)
fun2 :: [a] -> [a] -> (a,a)
```

Ενώ και οι δύο συναρτήσεις εφαρμόζονται σε δύο εισόδους, που είναι λίστες, και δίνουν αποτέλεσμα μια πλειάδα (με δύο συστατικά), στη `fun1` μπορούν οι δύο είσοδοι να είναι και διαφορετικών τύπων, αλλά στη `fun2` πρέπει υποχρεωτικά να είναι του ίδιου τύπου.

Τέλος, είναι χρήσιμο εδώ να κάνουμε μια επισήμανση για τη σύγχυση που προκαλείται μερικές φορές μεταξύ της έννοιας του πολυμορφισμού και μιας άλλης σχετικής έννοιας, αυτής της **υπερφόρτωσης**. Μια συνάρτηση λέγεται ότι είναι υπερφορτωμένη όταν μπορεί να εφαρμοστεί σε εισόδους διαφόρων τύπων, αλλά οι ακριβείς ορισμοί της συνάρτησης για κάθε πιθανό συνδυασμό τύπων των εισόδων της ενδέχεται να είναι διαφορετικοί μεταξύ τους, αν και νοηματικά παρόμοιοι. Για παράδειγμα, οι σχεσιακοί τελεστές, που είδαμε στην

[3] Οι μεταβλητές τύπου, όπως και οι τυπικές παράμετροι και τα ονόματα συναρτήσεων, πρέπει, σύμφωνα με τη σύνταξη της Haskell, να αρχίζουν με μικρό χαρακτήρα.

Ενότητα 10.1, είναι υπερφορτωμένες συναρτήσεις.[4] Στο Παράδειγμα 10.2, οι σχεσιακοί τελεστές >= και <= εφαρμόζονται σε χαρακτήρες, αλλά μπορούν, βέβαια, να εφαρμοστούν και μεταξύ αριθμών (ακεραίων ή πραγματικών). Οι τελεστές αυτοί είναι ορισμένοι εσωτερικά στη Haskell, αλλά οι ορισμοί τους θα πρέπει να είναι λίγο διαφορετικοί, ανάλογα με το αν εφαρμόζονται σε χαρακτήρες ή σε αριθμούς. Η Haskell δίνει τη δυνατότητα και στους ίδιους τους χρήστες να ορίσουν υπερφορτωμένες συναρτήσεις. Όμως, στο παρόν σύγγραμμα, δεν θα μας απασχολήσει ο τρόπος με τον οποίο γίνεται αυτό.

Άσκηση 10.8
Συμπληρώστε κατάλληλα τους ορισμούς των συναρτήσεων `fst` και `snd`, που δίνονται στη συνέχεια, έτσι ώστε, όταν αυτές εφαρμόζονται σε μια πλειάδα με δύο συστατικά, η μεν `fst` να επιστρέφει το πρώτο από αυτά, η δε `snd` το δεύτερο.

```
fst :: (      ) -> a
fst       = x

snd :: (a,b)
snd
```

Άσκηση 10.9
Στην Άσκηση 10.7 σας ζητήθηκε να ορίσετε δύο συναρτήσεις, τις `count_BPoints` και `count_WPoints`, που να μετρούν πόσα μαύρα και πόσα άσπρα σημεία, αντίστοιχα, υπάρχουν σε ένα μονοπάτι. Είναι δυνατόν να ορίσετε μία και μοναδική συνάρτηση, έστω με την ονομασία `count_BWPoints`, που να κάνει αυτούς τους υπολογισμούς; Μπορείτε να χρησιμοποιήσετε, αν θέλετε, τις συναρτήσεις `fst` και `snd` της Άσκησης 10.8.

10.4 Το πρελούδιο

Με βάση όσα έχουμε εξετάσει μέχρι τώρα σχετικά με τη Haskell, θα ήταν, πιστεύουμε, επιθυμητό να είχαμε εκ των προτέρων ορισμένο ένα σύνολο από συναρτήσεις, οι οποίες να εκτελούν χρήσιμες λειτουργίες και να είναι διαθέσιμες στα προγράμματα που γράφουμε. Πράγματι, η Haskell είναι εφοδιασμένη με διάφορες βιβλιοθήκες, καθεμία από τις οποίες περιλαμβάνει συναρτήσεις που υποστηρίζουν συγκεκριμένη λειτουργικότητα της γλώσσας.

Μία από τις βιβλιοθήκες της Haskell είναι το λεγόμενο **πρελούδιο**, το οποίο περιλαμβάνει πολύ βασικές και συχνότατα χρησιμοποιούμενες συναρτήσεις. Έχει ονομαστεί έτσι γιατί φορτώνεται αυτόματα σε ένα σύστημα Haskell, κατά την ενεργοποίηση του συστήματος, με αποτέλεσμα να είναι διαθέσιμο σε οποιοδήποτε πρόγραμμα του χρήστη. Έτσι, στα προγράμματα που γράφουμε, μπορούμε να θεωρούμε δεδομένες τις συναρτήσεις οι οποίες ορίζονται στο πρελούδιο και να τις χρησιμοποιούμε, σύμφωνα, φυσικά, με τις προδιαγραφές τους.

Μερικές από τις σημαντικότερες (κατά την κρίση του συγγραφέα) συναρτήσεις που είναι ορισμένες στο πρελούδιο φαίνονται στον Πίνακα 10.1. Παρατηρήστε ότι ορισμένες από

[4] Μη σας ξενίζει το γεγονός ότι χαρακτηρίζουμε τους σχεσιακούς τελεστές συναρτήσεις, παρότι φαίνεται ότι τους χρησιμοποιούμε, από συντακτικής πλευράς, με διαφορετικό τρόπο από τον καθιερωμένο. Ουσιαστικά, για τους τελεστές αυτούς, αλλά και για κάποιους άλλους, είναι πιο βολικό να έχουμε μια ενδοθεματική σύνταξη. Μπορούμε και συναρτήσεις που ορίζουμε εμείς οι ίδιοι να τις χρησιμοποιήσουμε με ενδοθεματικό τρόπο, αν περικλείσουμε το όνομά τους σε `. Για παράδειγμα, η έκφραση `f 2 5` είναι ισοδύναμη με την `2 `f` 5`.

τις συναρτήσεις που ορίσαμε σε αυτό το κεφάλαιο, όπως η `sum` του Παραδείγματος 10.9, η `length` του Παραδείγματος 10.11 και οι `fst` και `snd` της Άσκησης 10.8, είναι ήδη ορισμένες στο πρελούδιο.

Άσκηση 10.10
Ζητήστε από ένα σύστημα Haskell τον υπολογισμό διαφόρων εκφράσεων, έτσι ώστε να δείτε στην πράξη τη λειτουργικότητα των συναρτήσεων του πρελουδίου που βρίσκονται στον Πίνακα 10.1. Θυμηθείτε ότι στις συναρτήσεις που διαχειρίζονται λίστες μπορείτε να χρησιμοποιείτε και συμβολοσειρές (δεδομένα τύπου `String`), αφού αυτές είναι στην πραγματικότητα λίστες από χαρακτήρες (δεδομένα τύπου `Char`).

10.5 Εφαρμογές της Haskell

Η Haskell είναι ένας τυπικός εκπρόσωπος της έννοιας του συναρτησιακού προγραμματισμού, δηλαδή εκείνης της προγραμματιστικής φιλοσοφίας που, μαζί με τον λογικό προγραμματισμό, υποστηρίζει, κατά κύριο λόγο, μια δηλωτική αντιμετώπιση προβλημάτων του πραγματικού κόσμου. Υπ' αυτήν την έννοια, οι περιοχές εφαρμογών του συναρτησιακού προγραμματισμού, άρα και της Haskell, δεν διαφέρουν ουσιαστικά από αυτές του λογικού προγραμματισμού, τις κυριότερες εκ των οποίων εξετάσαμε αναλυτικά στην Ενότητα 3.4.

Επιγραμματικά, μπορούμε να αναφερθούμε στις ακόλουθες περιοχές εφαρμογών, για τις οποίες θα ήταν αρκετά καλή ιδέα να επιλεγεί η Haskell ως γλώσσα υποστήριξης. Για όλες αυτές τις περιοχές, έχουν υλοποιηθεί πραγματικές εφαρμογές σε Haskell, αλλά και σε άλλες γλώσσες συναρτησιακού προγραμματισμού:

- Απόδειξη θεωρημάτων
- Επεξεργασία φυσικής γλώσσας
- Αναγνώριση προφορικού λόγου
- Διερμηνείς και μεταγλωττιστές
- Επαλήθευση κατανεμημένων συστημάτων
- Ρομποτική
- Συμβολική επεξεργασία

Για μια περισσότερο αναλυτική παρουσίαση συγκεκριμένων εφαρμογών υλοποιημένων σε Haskell, βλ. την επίσημη ιστοσελίδα της γλώσσας:

```
http://www.haskell.org
```

Απαντήσεις ασκήσεων

Απάντηση άσκησης 10.1
Ο ορισμός της `max3` πρέπει να συμπληρωθεί ως εξής:

```
max3 :: Float -> Float -> Float -> Float
max3 x y z
  | x >= y && x >= z    = x
  | y >= z              = y
  | otherwise           = z
```

Συνάρτηση	Τύπος	Περιγραφή
`:`	`a -> [a] -> [a]`	Προσθήκη ενός στοιχείου στην αρχή μιας λίστας (ενδοθεματική σύνταξη)
`++`	`[a] -> [a] -> [a]`	Συνένωση δύο λιστών (ενδοθεματική σύνταξη)
`!!`	`[a] -> Int -> a`	Επιλογή στοιχείου δεδομένης τάξης από λίστα (ενδοθεματική σύνταξη)
`concat`	`[[a]] -> [a]`	Συνένωση λιστών που είναι στοιχεία λίστας
`length`	`[a] -> Int`	Μήκος λίστας
`head`	`[a] -> a`	Πρώτο στοιχείο της λίστας
`last`	`[a] -> a`	Τελευταίο στοιχείο της λίστας
`tail`	`[a] -> [a]`	Διαγραφή του πρώτου στοιχείου της λίστας
`init`	`[a] -> [a]`	Διαγραφή του τελευταίου στοιχείου της λίστας
`null`	`[a] -> Bool`	Έλεγχος για κενή λίστα
`replicate`	`Int -> a -> [a]`	Δημιουργία λίστας από δεδομένο αριθμό αντιγράφων στοιχείου
`take`	`Int -> [a] -> [a]`	Επιλογή δεδομένου αριθμού στοιχείων από την αρχή της λίστας
`drop`	`Int -> [a] -> [a]`	Διαγραφή δεδομένου αριθμού στοιχείων από την αρχή της λίστας
`splitAt`	`Int -> [a] -> ([a],[a])`	Διάσπαση λίστας σε δεδομένη θέση
`reverse`	`[a] -> [a]`	Αντιστροφή λίστας
`zip`	`[a] -> [b] -> [(a,b)]`	Σύμπτυξη των στοιχείων δύο λιστών σε λίστα ζευγαριών των αντίστοιχων στοιχείων
`unzip`	`[(a,b)] -> ([a],[b])`	Διάσπαση λίστας ζευγαριών σε ζευγάρι λιστών
`fst`	`(a,b) -> a`	Επιλογή πρώτου συστατικού από ζευγάρι
`snd`	`(a,b) -> b`	Επιλογή δεύτερου συστατικού από ζευγάρι
`words`	`String -> [String]`	Διάσπαση συμβολοσειράς σε λίστα από λέξεις (συμβολοσειρές)
`and`	`[Bool] -> Bool`	Σύζευξη στοιχείων λίστας από λογικές τιμές
`or`	`[Bool] -> Bool`	Διάζευξη στοιχείων λίστας από λογικές τιμές
`sum`	`[Int] -> Int` `[Float] -> Float`	Υπολογισμός αθροίσματος στοιχείων λίστας
`product`	`[Int] -> Int` `[Float] -> Float`	Υπολογισμός γινομένου στοιχείων λίστας
`pi`	`Float`	Το γνωστό π ($= 3.14\ldots$)

Πίνακας 10.1: Μερικές συναρτήσεις από το πρελούδιο της Haskell.

Η έκφραση `max3 2.7 1.2 1.8` έχει τιμή `2.7`, επειδή γι' αυτήν αληθεύει ο πρώτος φρουρός του ορισμού, ενώ η έκφραση `max3 1.3 1.9 3.1` έχει τιμή `3.1`, λόγω της εφαρμογής του φρουρού `otherwise`, αφού οι δύο πρώτοι είναι ψευδείς.

Απάντηση άσκησης 10.2
Ο ορισμός της συνάρτησης `exOr` μπορεί να γραφεί σωστά, αν περιγράψουμε στη Haskell την πράξη της αποκλειστικής διάζευξης:

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

Τι δηλώνεται με αυτόν τον ορισμό; Πολύ απλά, ότι η αποκλειστική διάζευξη ταυτίζεται με την απλή διάζευξη (`||`), αρκεί να μην είναι αληθείς ταυτόχρονα και οι δύο διαζευκτέοι (`&& not ...`). Η έκφραση `exOr True True` υπολογίζεται ως εξής:

```
exOr True True = (True || True) && not (True && True) =
                 True && not True = True && False = False
```

Εφαρμόζοντας τον ορισμό στην έκφραση `exOr False True`, έχουμε:

```
exOr False True = (False || True) && not (False && True) =
                  True && not False = True && True = True
```

Απάντηση άσκησης 10.3
Θα μπορούσαμε να ορίσουμε τη ζητούμενη συνάρτηση `threeDifferent` ως εξής:

```
threeDifferent :: Int -> Int -> Int -> Bool
threeDifferent x y z = (x /= y) && (y /= z) && (z /= x)
```

Προσέξτε ότι χρειαζόμαστε στον ορισμό και τη συνθήκη `z /= x`. Δεν αρκούν μόνο οι δύο πρώτες, γιατί τότε η έκφραση `threeDifferent 3 7 3` δεν θα μπορούσε να υπολογιστεί σωστά ως `False`.

Απάντηση άσκησης 10.4
Η συνάρτηση `complicated` που δίνεται είναι τελικά πιο απλή απ' ό,τι φαίνεται. Θα μπορούσαμε να την ορίσουμε εντελώς ισοδύναμα και ως:

```
complicated :: Int -> Int -> Int -> Bool
complicated x y z = (x > y) || (x > z)
```

Εύκολα μπορούμε να παρατηρήσουμε ότι η μόνη περίπτωση να δώσει τιμή `False` η συνάρτηση είναι να αληθεύει ο δεύτερος φρουρός (`y >= x`), αφού έχει αποδειχθεί ψευδής ο πρώτος (δηλαδή, ισχύει `x <= z`). Συνεπώς, η συνάρτηση είναι αληθής σε όλες τις άλλες περιπτώσεις, δηλαδή όταν ισχύει:

```
not ((y >= x) && (x <= z))
```

ή αλλιώς:

```
not (y >= x) || not (x <= z)
```

ή, τελικά:

```
    (x > y) || (x > z)
```

Απάντηση άσκησης 10.5
Η ακολουθία Fibonacci θα μπορούσε να οριστεί στη Haskell ως εξής:

```
fib :: Int -> Int
fib n
  | n == 1    = 1
  | n == 2    = 1
  | n > 2     = fib (n-1) + fib (n-2)
```

Η συνάρτηση για τον μέγιστο κοινό διαιρέτη μπορεί να γραφεί έτσι:

```
gcd :: Int -> Int -> Int
gcd x y
  | x == y   = x
  | x < y    = gcd x (y-x)
  | x > y    = gcd (x-y) y
```

Βλέπουμε ότι τόσο η συνάρτηση `fib`, στην τρίτη περίπτωση, όσο και η συνάρτηση `gcd`, στη δεύτερη και στην τρίτη περίπτωση, ορίζονται μέσω των ίδιων τους των εαυτών.

Απάντηση άσκησης 10.6
Η σωστή αντιστοίχιση είναι: α-3, β-5, γ-1, δ-2, ε-4.

Απάντηση άσκησης 10.7
Οι συναρτήσεις `count_BPoints` και `count_WPoints` μπορούν να οριστούν ως εξής:

```
type BW_Point_2D = (Float,Float,Bool)
type BW_Path = [BW_Point_2D]

count_BPoints :: BW_Path -> Int
count_BPoints []                  = 0
count_BPoints ((_,_,False):path) = 1 + count_BPoints path
count_BPoints ((_,_,True):path)  = count_BPoints path

count_WPoints :: BW_Path -> Int
count_WPoints []                  = 0
count_WPoints ((_,_,True):path)  = 1 + count_WPoints path
count_WPoints ((_,_,False):path) = count_WPoints path
```

Η ιδέα των ορισμών βασίζεται στο ταίριασμα προτύπων. Σαρώνοντας αναδρομικά τη λίστα από τα σημεία, δηλαδή το μονοπάτι, όταν συναντάμε μαύρο σημείο (`False`), έχουμε συνεισφορά κατά 1 στο αποτέλεσμα της συνάρτησης `count_BPoints`, ενώ τα άσπρα σημεία δεν επηρεάζουν το αποτέλεσμά της. Ακριβώς αντίστροφη είναι η κατάσταση με τη συνάρτηση `count_WPoints`.

Επίσης, προσέξτε ότι στους ορισμούς των δύο συναρτήσεων χρησιμοποιήσαμε το σύμβολο _ στη θέση των τυπικών παραμέτρων που παριστάνουν τις συντεταγμένες των σημείων του μονοπατιού. Αυτό το κάναμε γιατί δεν μας ενδιαφέρει να δώσουμε συγκεκριμένα ονόματα σε αυτές τις τυπικές παραμέτρους, αφού δεν τις χρησιμοποιούμε στα δεξιά μέλη των τύπων. Το σύμβολο αυτό δεν το είχαμε αναφέρει μέχρι στιγμής. Αν γράψατε τους ορισμούς δίνοντας συγκεκριμένα ονόματα στις τυπικές παραμέτρους, φυσικά δεν είναι λάθος.

Απάντηση άσκησης 10.8
Οι ορισμοί των συναρτήσεων `fst` και `snd` θα πρέπει να συμπληρωθούν ως εξής:

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

Φυσικά, θα μπορούσατε να είχατε χρησιμοποιήσει κάποια άλλη μεταβλητή τύπου, αντί της `b`, στον ορισμό της `fst` ή και κάποια άλλη τυπική παράμετρο, αντί της `y`, στον ορισμό της `snd`. Και στους δύο ορισμούς, θα μπορούσατε επίσης να είχατε βάλει συγκεκριμένη τυπική παράμετρο στο συστατικό της πλειάδας που δεν μας ενδιαφέρει σε κάθε περίπτωση, αντί του συμβόλου _.

Απάντηση άσκησης 10.9
Θα μπορούσαμε να ορίσουμε τη συνάρτηση `count_BWPoints` ως εξής:

```
type BW_Point_2D = (Float,Float,Bool)
type BW_Path = [BW_Point_2D]

count_BWPoints :: BW_Path -> (Int,Int)
count_BWPoints [] = (0,0)
count_BWPoints ((_,_,False):path) = (1 + fst (count_BWPoints path),
                                     snd (count_BWPoints path))
count_BWPoints ((_,_,True):path)  = (fst (count_BWPoints path),
                                     1 + snd (count_BWPoints path))
```

Η συνάρτηση `count_BWPoints` επιστρέφει ένα ζευγάρι ακεραίων, εκ των οποίων ο πρώτος είναι το πλήθος των μαύρων (`False`) σημείων του μονοπατιού και ο δεύτερος το πλήθος των άσπρων (`True`). Εξερευνώντας αναδρομικά τη λίστα των σημείων, αυξάνουμε κατά 1 το κατάλληλο συστατικό του ζευγαριού, ανάλογα με το χρώμα του σημείου που συναντάμε.

Όμως, ο ορισμός που δώσαμε, παρότι απόλυτα σωστός, δεν είναι διατυπωμένος με τον καλύτερο δυνατό τρόπο. Το πρόβλημα βρίσκεται στους δύο αναδρομικούς τύπους του ορισμού. Στον καθένα από αυτούς, για να υπολογιστούν και τα δύο συστατικά του ζευγαριού στο αποτέλεσμα, ζητάμε **δύο** φορές να υπολογιστεί η έκφραση `count_BWPoints path`. Αυτό είναι εντελώς περιττό, και, ανάλογα με το μήκος της λίστας `path`, ίσως να είναι και αρκετά επιβαρυντικό για την αποδοτικότητα του ορισμού. Μας δίνει, άραγε, η Haskell τη δυνατότητα για καλύτερη διατύπωση; Ευτυχώς, ναι! Δείτε τον επόμενο ορισμό:

```
fast_count_BWPoints :: BW_Path -> (Int,Int)
fast_count_BWPoints [] = (0,0)
fast_count_BWPoints ((_,_,False):path) = (1 + fst cBWPpath,
                                          snd cBWPpath)
                     where cBWPpath = fast_count_BWPoints path
fast_count_BWPoints ((_,_,True):path)  = (fst cBWPpath,
                                          1 + snd cBWPpath)
                     where cBWPpath = fast_count_BWPoints path
```

Εδώ δώσαμε τοπικούς ορισμούς, μέσω της λέξης-κλειδιού `where`, για το `cBWPpath`, που χρησιμοποιούνται στη συνάρτηση. Έτσι, η έκφραση `fast_count_BWPoints path`, σε κάθε αναδρομικό βήμα που γίνεται κατά τον υπολογισμό, υπολογίζεται μία μόνο φορά.

Με αφορμή την άσκηση αυτή, είναι χρήσιμο να θίξουμε και ένα άλλο θέμα στο οποίο δεν αναφερθήκαμε μέχρι τώρα. Πότε τελειώνει ένας ορισμός και αρχίζει κάποιος άλλος; Αφού δεν υπάρχει ειδικό σύμβολο που να δείχνει το τέλος ενός ορισμού, πώς το καταλαβαίνει ένα σύστημα Haskell; Η απάντηση βρίσκεται στη στοίχιση. Προσέξτε ότι στις συναρτήσεις που ορίσαμε στην άσκηση αυτή, κάποιοι ορισμοί συνέχιζαν και σε επόμενη γραμμή. Όμως, οι επόμενες γραμμές πάντοτε άρχιζαν πιο δεξιά από την πρώτη γραμμή του ορισμού. Αυτό δεν ήταν τυχαίο και σίγουρα δεν έγινε μόνο για λόγους καλαισθησίας. Έπρεπε να γίνει έτσι, για να ξεχωρίσουν οι ορισμοί μεταξύ τους.

Απάντηση άσκησης 10.10
Στη συνέχεια, μέσα από το περιβάλλον της Haskell φαίνονται τα αποτελέσματα του υπολογισμού διαφόρων εκφράσεων που εμπλέκουν τις συναρτήσεις του πρελούδιου από τον Πίνακα 10.1.

```
Prelude> "miranda"!!2
'r'
Prelude> length (concat [[1,2],[3,4,5],[6],[7,8,9,0]])
10
Prelude> tail "haskell" ++ init "curry"
"askellcurr"
Prelude> replicate 5 (last (3:[4,5,6]))
[6,6,6,6,6]
Prelude> head (reverse "word")
'd'
Prelude> fst (unzip [(1,2),(3,4),(5,6)])
[1,3,5]
Prelude> zip (take 3 "alonzo") (drop 2 "church")
[('a','u'),('l','r'),('o','c')]
Prelude> snd (splitAt 4 [9,8,7,6,5,4,3])
[5,4,3]
Prelude> words "this is a test string"
["this","is","a","test","string"]
Prelude> and [null [], null [1]]
False
Prelude> or [null [], null [1]]
True
Prelude> product [pi, (sum [pi,1])]
13.0112
Prelude>
```

Ας κάνουμε, όμως, μερικά σχόλια:

- Ας θεωρήσουμε ότι το «`Prelude>`», που βλέπουμε σε αρκετά σημεία, αποτελεί προτροπή του συστήματος να δώσουμε μια έκφραση για υπολογισμό. Δεν είναι «`*Main>`» εδώ, όπως σε προηγούμενα παραδείγματα, επειδή τώρα δεν έχουμε φορτώσει στο σύστημα συγκεκριμένο αρχείο με ορισμούς συναρτήσεων.
- Στη συνάρτηση `!!`, το πρώτο στοιχείο της λίστας είναι τάξης 0. Η ίδια σύμβαση ισχύει και στη συνάρτηση `splitAt`. Για τον υπολογισμό της θέσης διάσπασης, θεωρούμε ότι το πρώτο στοιχείο της λίστας είναι στη θέση 0 και ότι το στοιχείο στη θέση διάσπασης είναι το πρώτο στοιχείο της δεύτερης λίστας που προκύπτει από τη διάσπαση.
- Η «συνάρτηση» `pi` είναι ουσιαστικά μια σταθερά. Μπορούμε και εμείς να ορίζουμε στα προγράμματά μας σταθερές με συγκεκριμένα ονόματα (και συγκεκριμένους τύπους), τις οποίες να χρησιμοποιούμε κατά βούληση. Προσέξτε, όμως, ότι αυτά τα

ονόματα δεν αντιστοιχούν με εκείνα τα οποία χρησιμοποιούμε στις εντολές αντικατάστασης των διαδικαστικών γλωσσών προγραμματισμού, που παριστάνουν μεταβλητές των οποίων η τιμή μπορεί να αλλάζει στη διάρκεια της εκτέλεσης ενός προγράμματος. Στη Haskell, τα ονόματα αυτά είναι σταθερές, με την κυριολεκτική έννοια του όρου. Έχουν, δηλαδή, την ίδια τιμή για όλο το πρόγραμμα.

Προβλήματα

Πρόβλημα 10.1
Ορίστε στη Haskell μια συνάρτηση `sign`, η οποία, όταν εφαρμόζεται σε έναν πραγματικό αριθμό, να επιστρέφει 1, 0 ή -1, ανάλογα με το αν ο αριθμός είναι θετικός, μηδέν ή αρνητικός, αντίστοιχα.

Πρόβλημα 10.2
Ορίστε στη Haskell τη συνάρτηση:

```
nAnd :: Bool -> Bool -> Bool
```

η οποία υλοποιεί τη λογική πράξη NAND (NOT AND).

Πρόβλημα 10.3
Ορίστε στη Haskell τη συνάρτηση `isPalindrome`, η οποία, όταν εφαρμόζεται σε μια λίστα, να επιστρέφει `True` ή `False`, ανάλογα με το αν η λίστα είναι παλινδρομική, δηλαδή ταυτίζεται με την αντίστροφή της, ή όχι.

Πρόβλημα 10.4
Ορίστε στη Haskell τη συνάρτηση `capitalize`, η οποία, όταν εφαρμόζεται σε μια συμβολοσειρά (τύπου `String`), να επιστρέφει μια νέα συμβολοσειρά, στην οποία τα πεζά γράμματα της αρχικής να έχουν αντικατασταθεί με τα αντίστοιχα κεφαλαία, ενώ ό,τι δεν είναι πεζό γράμμα να μεταφέρεται στη νέα συμβολοσειρά ως έχει. Για παράδειγμα:

```
*Main> capitalize "ab12-CDe*&fG34hI"
"AB12-CDE*&FG34HI"
*Main>
```

Πρόβλημα 10.5
Ορίστε στη Haskell τη συνάρτηση `insertAt`, η οποία να εισάγει δεδομένο στοιχείο σε δεδομένη θέση δεδομένης λίστας.

Πρόβλημα 10.6
Η συνάρτηση Ackermann ορίζεται ως εξής:

$$ackermann(M,N) = \begin{cases} N+1 & \text{αν } M = 0 \text{ και } N \geq 0 \\ ackermann(M-1,1) & \text{αν } M > 0 \text{ και } N = 0 \\ ackermann(M-1, ackermann(M, N-1)) & \text{αν } M > 0 \text{ και } N > 0 \end{cases}$$

Ορίστε συνάρτηση Haskell που να την υλοποιεί.

Πρόβλημα 10.7
Ορίστε στη Haskell τη συνάρτηση:

```
dropEvery :: [a] -> Int -> [a]
```

η οποία να δέχεται μια λίστα και έναν ακέραιο n, και να επιστρέφει μια νέα λίστα που προκύπτει από την αρχική, αν διαγραφεί από αυτήν κάθε n-οστό στοιχείο της. Για παράδειγμα:

```
*Main> dropEvery [12,345,67,8,90,987,6,5,432,1] 4
[12,345,67,90,987,6,432,1]
*Main> dropEvery "abcdefghijklmnopqrstuvwxyz" 7
"abcdefhijklmopqrstvwxyz"
*Main>
```

Πρόβλημα 10.8
Ορίστε στη Haskell τη συνάρτηση:

```
rotate :: [a] -> Int -> [a]
```

η οποία να δέχεται μια λίστα και έναν ακέραιο n, και να επιστρέφει μια νέα λίστα που προκύπτει από την περιστροφή της αρχικής κατά n θέσεις αριστερά. Για παράδειγμα:

```
*Main> rotate "abcdefghijk" 5
"fghijkabcde"
*Main> rotate "abcdefghijk" 0
"abcdefghijk"
*Main> rotate "abcdefghijk" 24
"cdefghijkab"
*Main>
```

Μήπως ο ορισμός που δώσατε μπορεί να δουλέψει και έτσι;

```
*Main> rotate "abcdefghijk" (-2)
"jkabcdefghi"
*Main>
```

Πρόβλημα 10.9
Ορίστε στη Haskell τη συνάρτηση:

```
isMember :: Char -> String -> Bool
```

η οποία να ελέγχει αν δεδομένος χαρακτήρας περιέχεται σε μια συμβολοσειρά. Για παράδειγμα:

```
*Main> isMember 'c' "abcdef"
True
*Main> isMember 'g' "abcdef"
False
*Main>
```

Πρόβλημα 10.10
Ορίστε στη Haskell τη συνάρτηση:

```
disjoint :: String -> String -> Bool
```

η οποία να ελέγχει αν δύο συμβολοσειρές δεν έχουν κανέναν κοινό χαρακτήρα. Για παράδειγμα:

```
*Main> disjoint "abcd" "efg"
True
*Main> disjoint "abcd" "efbg"
False
*Main>
```

Βιβλιογραφικές αναφορές

[1] S. Thompson, *Haskell: The Craft of Functional Programming*, Addison Wesley, 2011.

[2] R. Bird, *Thinking Functionally with Haskell*, Cambridge University Press, 2015.

Κεφάλαιο 11
Εκφραστικές δυνατότητες της Haskell

Σύνοψη

Στο κεφάλαιο αυτό παρουσιάζονται οι περιφραστικές λίστες, μια δυνατότητα που παρέχει η Haskell και με την οποία μπορούμε πολύ εύκολα να ορίσουμε διάφορες συναρτήσεις που εφαρμόζονται σε λίστες. Επίσης, γίνεται αναφορά σε ένα πολύ βασικό χαρακτηριστικό του συναρτησιακού προγραμματισμού, συνεπώς και της Haskell, τη δυνατότητα να ορίζουμε συναρτήσεις ανώτερης τάξης, δηλαδή συναρτήσεις που δέχονται στην είσοδό τους συναρτήσεις ή δίνουν αποτέλεσμα συναρτήσεις (ή και τα δύο). Ακόμα, παρουσιάζεται πώς μπορούμε, με την τεχνική της σύνθεσης συναρτήσεων, να ορίζουμε περισσότερο πολύπλοκες συναρτήσεις με τη βοήθεια απλούστερων, που λειτουργούν ως στοιχειώδεις δομικοί λίθοι, αλλά και εξηγείται γιατί τελικά βλέπουμε τις συναρτήσεις στον συναρτησιακό προγραμματισμό, ανεξάρτητα από το πλήθος των ορισμάτων επάνω στα οποία εφαρμόζονται, ως συναρτήσεις ενός μόνο ορίσματος. Επιπλέον, εξετάζεται η δυνατότητα την οποία έχουμε στον συναρτησιακό προγραμματισμό να προβαίνουμε σε τυπικές αποδείξεις ιδιοτήτων για τα προγράμματα που γράφουμε και αναφέρονται επιγραμματικά άλλα χαρακτηριστικά της Haskell, για τα οποία, αν και είναι ιδιαιτέρως σημαντικά, δεν έχουμε τη δυνατότητα να κάνουμε αναλυτικότερη παρουσίαση στο πλαίσιο του παρόντος συγγράμματος.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει μελετήσει το Κεφάλαιο 10.

11.1 Περιφραστικές λίστες

Η Haskell [1, 2] μας επιτρέπει να διατυπώσουμε μια λίστα όχι μόνο με ρητή απαρίθμηση των στοιχείων της, αλλά και με έμμεσο τρόπο, μέσω των στοιχείων μιας άλλης λίστας. Έτσι, μπορούμε να κατασκευάσουμε τις λεγόμενες **περιφραστικές λίστες**, οι οποίες είναι πολύ εξυπηρετικές στον συναρτησιακό προγραμματισμό. Ας δούμε όμως, μέσω συγκεκριμένων παραδειγμάτων, πώς διατυπώνονται οι περιφραστικές λίστες.

Παράδειγμα 11.1
Έστω ότι θέλουμε να ορίσουμε μια συνάρτηση `squareList`, η οποία να δέχεται είσοδο μια λίστα από ακεραίους και να κατασκευάζει τη λίστα από τα τετράγωνα των ακεραίων αυτών. Η συνάρτηση αυτή θα μπορούσε να υλοποιηθεί πολύ απλά, μέσω αναδρομής, ως εξής:

```
squareList :: [Int] -> [Int]
squareList []     = []
```

```
squareList (x:xs) = x^2 : (squareList xs)
```

Θα μπορούσαμε όμως να ορίσουμε τη `squareList` πολύ απλούστερα, με τη χρήση μιας περιφραστικής λίστας. Ας δούμε τον τρόπο (ονομάζοντας τη νέα συνάρτηση `csquareList`):

```
csquareList :: [Int] -> [Int]
csquareList xs = [ x^2 | x <- xs ]
```

Οπότε, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `csquareList`, ως εξής:

```
*Main> csquareList [4,2,6,1,7]
[16,4,36,1,49]
*Main>
```

Η λεκτική διατύπωση του ορισμού που δώσαμε για τη `csquareList` θα μπορούσε να ήταν: «*Η λίστα των τετραγώνων των στοιχείων της λίστας* `xs` *αποτελείται από όλα τα* x^2*, για κάθε* `x` *που ανήκει στη* `xs`». Η έκφραση `[ x^2 | x <- xs ]` είναι η περιφραστική λίστα. Το `x <- xs` ονομάζεται **γεννήτορας** της περιφραστικής λίστας. Ουσιαστικά, μπορούμε να φανταζόμαστε το σύμβολο `<-` ως το σύμβολο $\in$ του «ανήκειν» στα μαθηματικά. □

Είναι δυνατόν να συνδυάσουμε έναν γεννήτορα με κάποια συνθήκη, στη διατύπωση μιας περιφραστικής λίστας, ώστε να περιορίσουμε κατάλληλα τα στοιχεία που αυτός παράγει και χρησιμοποιούνται για τον ορισμό των στοιχείων της περιφραστικής λίστας. Επίσης, δεν απαγορεύεται, και μάλιστα σε πολλές περιπτώσεις είναι εξαιρετικά χρήσιμο, να έχουμε περισσότερους του ενός γεννήτορες. Δείτε το επόμενο παράδειγμα, το οποίο επιδεικνύει αυτές τις δύο δυνατότητες.

Παράδειγμα 11.2
Έστω ότι θέλουμε να ορίσουμε μια συνάρτηση `givenSum`, η οποία να δέχεται δύο λίστες από ακεραίους και να κατασκευάζει μια λίστα από ζευγάρια, με πρώτο συστατικό ένα στοιχείο από την πρώτη λίστα, δεύτερο συστατικό ένα στοιχείο από τη δεύτερη λίστα, αλλά με δεδομένο άθροισμα των δύο συστατικών, το οποίο επίσης να δίνεται ως είσοδος στη συνάρτηση. Η συνάρτηση `givenSum` θα μπορούσε να οριστεί ως εξής:

```
givenSum :: [Int] -> [Int] -> Int -> [(Int,Int)]
givenSum xs ys s = [(x,y) | x <- xs, y <- ys, x+y == s ]
```

Ιδού ο υπολογισμός μιας έκφρασης από τη `givenSum`:

```
*Main> givenSum [1,3,4,6] [1,2,3,5,6] 7
[(1,6),(4,3),(6,1)]
*Main>
```

Όταν έχουμε περισσότερους του ενός γεννήτορες στη διατύπωση μιας περιφραστικής λίστας, αυτό αντιστοιχεί στο καρτεσιανό γινόμενο των αντίστοιχων λιστών. □

Ας δούμε όμως κι άλλο ένα παράδειγμα, που θα είναι καταλυτικό στο να πειστούμε για τη χρησιμότητα των περιφραστικών λιστών, εάν αυτό δεν έχει γίνει μέχρι τώρα.

Παράδειγμα 11.3
Έστω ότι θέλουμε να βρούμε όλα τα πυθαγόρεια τρίγωνα[1] με πλευρές όχι μεγαλύτερες δεδομένου μήκους. Μπορούμε να ορίσουμε μια συνάρτηση `pythTriangles`, στην οποία να δίνουμε έναν ακέραιο (το μέγιστο μήκος των πλευρών του τριγώνου) και αυτή να κατασκευάζει μια λίστα με όλες τις τριάδες από ακεραίους που είναι τα μήκη πλευρών πυθαγόρειων τριγώνων, μικρότερα ή ίσα όμως από το δεδομένο μέγιστο. Ο ορισμός της συνάρτησης δίνεται στη συνέχεια. Προσέξτε, επίσης, στον ορισμό τις εκφράσεις της μορφής `[n .. m]`, που βλέπουμε για πρώτη φορά. Μια τέτοια έκφραση παριστάνει τη λίστα με όλους τους ακεραίους, από τον `n` έως τον `m`, δηλαδή την `[n, n+1, n+2, ..., m-1, m]`:

```
pythTriangles n = [ (x,y,z) | x <- [1 .. n-2], y <- [x+1 .. n-1],
                              z <- [y+1 .. n], x^2 + y^2 == z^2 ]
```

Προσέξτε ότι, για την πλευρά `x` του τριγώνου, επιτρέπουμε να κυμαίνεται από `1` έως `n-2`, επειδή θεωρούμε πως θα είναι η μικρότερη κάθετη πλευρά του. Επίσης, είναι γνωστό, και το θεωρούμε δεδομένο, ότι δεν υπάρχουν ισοσκελή πυθαγόρεια τρίγωνα. Στην πλευρά `y`, που είναι η μεγαλύτερη κάθετη, βάζουμε κάτω όριο το `x+1`, για να αποφύγουμε την κατασκευή συμμετρικών λύσεων, π.χ. `(3,4,5)` και `(4,3,5)`, και πάνω όριο το `n-1`, επειδή είναι σίγουρα μικρότερη από την υποτείνουσα. Για την υποτείνουσα `z`, βάζουμε κάτω όριο το `y+1`, αφού γνωρίζουμε ότι σίγουρα είναι μεγαλύτερη από τις κάθετες πλευρές, με σκοπό να αυξήσουμε την αποδοτικότητα του ορισμού μας, και πάνω όριο, φυσικά, το `n`.

Ορίστε, λοιπόν, τα πυθαγόρεια τρίγωνα με πλευρές όχι μεγαλύτερες του 30:

```
*Main> pythTriangles 30
[(3,4,5),(5,12,13),(6,8,10),(7,24,25),(8,15,17),(9,12,15),
 (10,24,26),(12,16,20),(15,20,25),(18,24,30),(20,21,29)]
*Main>
```

Άσκηση 11.1
Γνωρίζουμε ότι το εσωτερικό γινόμενο δύο διανυσμάτων ισούται με το άθροισμα των γινομένων των ομόλογων στοιχείων τους. Αν παραστήσουμε ένα διάνυσμα ως μια λίστα από πραγματικούς αριθμούς (τις συντεταγμένες του διανύσματος), δεν θα ήταν δύσκολο να ορίσουμε μια συνάρτηση στη Haskell που να υπολογίζει το εσωτερικό γινόμενο δύο δεδομένων διανυσμάτων. Στη συνέχεια, δίνουμε δύο υποψήφιους ορισμούς για το εσωτερικό γινόμενο, για τις συναρτήσεις `scalProd1` και `scalProd2`. Ο ένας από αυτούς είναι σωστός, ενώ ο άλλος είναι λανθασμένος. Ποιος είναι σωστός και ποιος λανθασμένος;

```
scalProd1 :: [Float] -> [Float] -> Float
scalProd1 xs ys = sum [ x*y | x <- xs, y <- ys ]

scalProd2 :: [Float] -> [Float] -> Float
scalProd2 xs ys = sum [ x*y | (x,y) <- zip xs ys ]
```

Παρατηρήστε ότι στους ορισμούς αυτούς εφαρμόζουμε μια συνάρτηση, τη `sum`, επάνω σε περιφραστικές λίστες. Επίσης, υπενθυμίζεται ότι η συνάρτηση `zip`, που είναι ορισμένη στο πρελούδιο της Haskell, συμπτύσσει δύο λίστες από στοιχεία σε μια λίστα από ζευγάρια.

[1] Πυθαγόρειο τρίγωνο είναι ένα ορθογώνιο τρίγωνο με πλευρές ακέραιου μήκους, π.χ. 3, 4 και 5 (αφού $3^2 + 4^2 = 5^2$).

Άσκηση 11.2
Πολλές φορές θα ήταν χρήσιμο να είχαμε στη διάθεσή μας μια συνάρτηση που να ταξινομεί μια λίστα από ακεραίους. Θα μπορούσαμε να ορίσουμε διάφορες συναρτήσεις γι' αυτόν το λόγο, οι οποίες να υλοποιούν γνωστές μεθόδους ταξινόμησης. Μία από αυτές τις μεθόδους είναι η «γρήγορη ταξινόμηση», η οποία βασίζεται στην εξής ιδέα: «*Για να ταξινομήσουμε μια λίστα σε αύξουσα σειρά, παίρνουμε το πρώτο στοιχείο της, έστω* `x`*, διασπάμε την υπόλοιπη λίστα σε δύο άλλες, εκ των οποίων η μια περιέχει τα μικρότερα ή ίσα με το* `x` *στοιχεία και η άλλη τα μεγαλύτερα από το* `x`*, ταξινομούμε τις δύο αυτές λίστες και κατασκευάζουμε την τελική ταξινομημένη λίστα, με συνένωση των δύο αυτών ταξινομημένων λιστών, αφού παρεμβάλουμε στο σημείο συνένωσης και το στοιχείο* `x`». Ορίστε στη Haskell μια συνάρτηση `quicksort` που να υλοποιεί τη μέθοδο της γρήγορης ταξινόμησης.

11.2 Συναρτήσεις ανώτερης τάξης

Με βάση ό,τι έχουμε συζητήσει μέχρι τώρα, όταν ορίζουμε μια συνάρτηση στη Haskell, περιγράφουμε τον τρόπο με τον οποίο θέλουμε να απεικονίσουμε κάποια δεδομένα εισόδου (ακεραίους, πραγματικούς, λογικές τιμές, χαρακτήρες, συμβολοσειρές, λίστες από ακεραίους, πλειάδες με συστατικά ό,τι τύπου θέλουμε κτλ.) σε κάποιο αποτέλεσμα. Το αποτέλεσμα αυτό μπορεί επίσης να είναι συγκεκριμένου τύπου, απλούστερου ή πιο σύνθετου, όπως αυτούς που αναφέραμε ήδη. Θα μπορούσαμε όμως να είχαμε και συναρτήσεις που να δέχονται στην είσοδό τους συναρτήσεις ή να δίνουν αποτέλεσμα κάποια συνάρτηση ή και τα δύο; Η απάντηση στο ερώτημα αυτό είναι καταφατική. Μια τέτοια συνάρτηση, που είτε δέχεται είτε δίνει κάποια συνάρτηση (είτε και τα δύο), ονομάζεται **συνάρτηση ανώτερης τάξης**. Πώς μπορούμε, όμως, να ορίσουμε τέτοιες συναρτήσεις; Και εν πάση περιπτώσει, έχουν κάποια χρησιμότητα οι συναρτήσεις ανώτερης τάξης; Για να μπορέσουμε να δώσουμε απαντήσεις σε αυτά τα ερωτήματα, ας δούμε τα παραδείγματα που ακολουθούν.

Παράδειγμα 11.4
Όταν προγραμματίζουμε στη Haskell, πολύ συχνά χρειάζεται να εφαρμόσουμε μια συνάρτηση σε όλα τα στοιχεία μιας λίστας και να πάρουμε τα αποτελέσματα σε μια νέα λίστα. Κάτι τέτοιο κάναμε στο Παράδειγμα 11.1 με τις συναρτήσεις `squareList` και `csquareList`. Θα ήταν ενδιαφέρον αν μπορούσαμε να γράψουμε μια συνάρτηση η οποία να εφαρμόζει στα στοιχεία μιας λίστας μια συνάρτηση, όταν της δίνονται ως είσοδοι τόσο η λίστα αυτή, όσο και η συνάρτηση προς εφαρμογή. Μια συνάρτηση που λειτουργεί με αυτόν τον τρόπο είναι η `map`, η οποία ορίζεται ως εξής:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

Βλέπετε ότι η `map` δέχεται στην είσοδό της μια συνάρτηση (τύπου `a -> b`) και μια λίστα (τύπου `[a]`), εφαρμόζει τη συνάρτηση σε κάθε στοιχείο της λίστας και παράγει μια λίστα τύπου `[b]`. Η εφαρμογή αυτή καθαυτή ορίζεται με τη βοήθεια μιας περιφραστικής λίστας. Έτσι, θα μπορούσαμε να ορίσουμε τον τετραγωνισμό των στοιχείων λίστας και με τη βοήθεια της συνάρτησης ανώτερης τάξης `map` και της συνάρτησης τετραγωνισμού ακεραίων `square` (Παράδειγμα 10.1), αντί να χρησιμοποιήσουμε αναδρομή ή μια περιφραστική λίστα, όπως κάναμε στο Παράδειγμα 11.1, ως εξής:

```
hsquareList :: [Int] -> [Int]
hsquareList = map square
```

Η συνάρτηση ανώτερης τάξης `map` είναι χρήσιμη σε πάρα πολλές περιπτώσεις και γι' αυτόν το λόγο είναι ήδη ορισμένη στο πρελούδιο της Haskell (αν και δεν την περιλάβαμε στον Πίνακα 10.1).

Παράδειγμα 11.5
Μια άλλη πολύ χρήσιμη συνάρτηση ανώτερης τάξης στη Haskell είναι η `filter`, η οποία επίσης είναι ορισμένη μέσα στο πρελούδιο. Η συνάρτηση αυτή δέχεται στην είσοδό της μια συνάρτηση `p` (τύπου `a -> Bool`) και μια λίστα από στοιχεία (τύπου `[a]`). Η συνάρτηση `p` επιστρέφει `True` όταν κάποιο στοιχείο τύπου `a` που της δίνεται έχει συγκεκριμένη ιδιότητα και `False` όταν δεν την έχει. Η `filter` κατασκευάζει μια λίστα από εκείνα τα στοιχεία της λίστας που δέχτηκε στην είσοδό της, τα οποία έχουν την ιδιότητα `p`. Ορίστε ο ορισμός της `filter`, όπως αυτός περιέχεται στο πρελούδιο:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]
```

Έτσι, αν χρησιμοποιήσουμε και τη συνάρτηση `isDigit`, που ορίζεται στο πρελούδιο, είναι τύπου `Char -> Bool` και, όταν εφαρμόζεται σε ένα χαρακτήρα, επιστρέφει `True` όταν ο χαρακτήρας αυτός είναι ψηφίο, μπορούμε να ζητήσουμε τον εξής υπολογισμό:

```
Prelude> filter isDigit "p325-q2157"
"3252157"
Prelude>
```

Θυμηθείτε ότι οι συμβολοσειρές είναι λίστες από χαρακτήρες.

Άσκηση 11.3
Συμπληρώστε τους ορισμούς που ακολουθούν, έτσι ώστε η συνάρτηση `hlength` να συμπεριφέρεται όπως η γνωστή σας συνάρτηση `length` από το πρελούδιο, δηλαδή να υπολογίζει το πλήθος των στοιχείων μιας λίστας:

```
const1 :: a -> Int
const1 x = [ ]

hlength :: [    ] -> Int
hlength xs = sum ( [      ] const1 [    ] )
```

Άσκηση 11.4
Ορίστε μια συνάρτηση `iter`, η οποία να παίρνει στην είσοδό της έναν ακέραιο `n`, μια συνάρτηση `f` και κάποιο `x`, και να εφαρμόζει `n` φορές την `f` επάνω στο `x`. Δηλαδή, θέλουμε, με κάποια άτυπη διατύπωση, η `iter` να κάνει το εξής:

```
iter n f x = f (f (f ... (f x) ... ))
```

Η `iter` εφαρμόζεται `n` φορές στο δεξιό μέλος της ισότητας, για παράδειγμα:

```
iter 4 f x = f (f (f (f x)))
```

Αν το `n` είναι `0`, θέλουμε το `iter 0 f x` να ισούται με `x`.

Αφού ορίσετε τη συνάρτηση `iter`, συνδυάστε τη με κάποιον τρόπο με τη συνάρτηση `double`, όπου:

```
double :: Int -> Int
double x = 2*x
```

για να ορίσετε τη συνάρτηση `powertwo`, η οποία, όταν της δίνεται ένας ακέραιος `n`, επιστρέφει το 2^n.

Αν έχουμε και τη συνάρτηση `square` του Παραδείγματος 10.1, τι επιστρέφεται ως τιμή της έκφρασης `iter n square 3`;

11.3 Σύνθεση συναρτήσεων

Στον συναρτησιακό προγραμματισμό, άρα και στη Haskell, ο βασικός μηχανισμός υπολογισμού είναι η εφαρμογή συναρτήσεων επάνω σε δεδομένα εισόδου. Έτσι, αν έχουμε μια συνάρτηση `g`, η οποία εφαρμόζεται επάνω σε κάποιο `x`, στη Haskell γράφουμε την έκφραση `g x`, για να αναφερθούμε στο αποτέλεσμα της εφαρμογής αυτής. Αν το αποτέλεσμα αυτό έχει τύπο συμβατό με τον τύπο των δεδομένων εισόδου μιας συνάρτησης `f`, είναι αυτονόητο ότι μπορούμε να εφαρμόσουμε την `f` επάνω στο αποτέλεσμα της εφαρμογής της `g` επάνω στο `x`, εκφράζοντας το αποτέλεσμα της εφαρμογής αυτής ως `f (g x)`. Αυτή η διαδοχική εφαρμογή συναρτήσεων δεν είναι τίποτε άλλο από αυτό που ονομάζουμε στον συναρτησιακό προγραμματισμό, αλλά και στα μαθηματικά γενικότερα, **σύνθεση συναρτήσεων**.

Όταν συνθέτουμε συναρτήσεις, δημιουργούμε απλώς νέες συναρτήσεις, η εφαρμογή των οποίων έχει το ίδιο ακριβώς αποτέλεσμα με τη διαδοχική εφαρμογή των αρχικών συναρτήσεων. Έτσι, η διαδοχική εφαρμογή των συναρτήσεων `g` και `f` είναι ισοδύναμη με μια νέα συνάρτηση, την οποία στη Haskell συμβολίζουμε με «`f . g`». Ο τελεστής «`.`» της σύνθεσης συναρτήσεων δεν έχει κάτι το ιδιαίτερο. Είναι και αυτός μια συνάρτηση (ανώτερης τάξης), με ενδοθεματική σύνταξη, η οποία εφαρμόζεται σε μια συνάρτηση `f` (τύπου `b -> c`) και σε μια συνάρτηση `g` (τύπου `a -> b`), για να δώσει αποτέλεσμα μια συνάρτηση `f . g` (τύπου `a -> c`). Δηλαδή:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Προσέξτε ότι ο τύπος των αποτελεσμάτων της `g`, που είναι `b`, ταυτίζεται με τον τύπο των δεδομένων εισόδου της `f`. Επίσης, η συνάρτηση `f . g`, που είναι το αποτέλεσμα της σύνθεσης, έχει τύπο δεδομένων εισόδου τον `a`, που είναι και ο τύπος των δεδομένων εισόδου της `g`, και τύπο αποτελεσμάτων το `c`, που είναι και ο τύπος αποτελεσμάτων της `f`. Τέλος, στον ορισμό του τύπου του τελεστή σύνθεσης «`.`» τον έχουμε περικλείσει σε παρενθέσεις, για να δείξουμε ότι είναι συνάρτηση-τελεστής, που θα χρησιμοποιηθεί με ενδοθεματική σύνταξη.

Παράδειγμα 11.6
Έστω ότι θέλουμε να ορίσουμε μια συνάρτηση `twice`, η οποία, όταν εφαρμόζεται επάνω σε μια συνάρτηση `f`, να δίνει αποτέλεσμα μια άλλη συνάρτηση, που να ισοδυναμεί με δύο διαδοχικές εφαρμογές της `f`. Η `twice` μπορεί να οριστεί ως εξής:

```
twice :: (a -> a) -> (a -> a)
twice f = (f . f)
```

Δηλαδή, η συνάρτηση `twice` εφαρμόζεται επάνω σε μια συνάρτηση `f` τύπου `a -> a` και δίνει αποτέλεσμα συνάρτηση του ίδιου τύπου. Η `f` πρέπει να είναι συνάρτηση με το ίδιο πεδίο ορισμού και πεδίο τιμών, γιατί, λόγω της διαδοχικής εφαρμογής της, θα πρέπει το αποτέλεσμα που παράγει να είναι συμβατό με τον τύπο των δεδομένων εισόδου της.

Έτσι, αν έχουμε και μια συνάρτηση, τη `threetimes`, η οποία δέχεται είσοδο έναν ακέραιο και τον τριπλασιάζει:

```
threetimes :: Int -> Int
threetimes x = 3*x
```

η `twice threetimes` θα πρέπει να εννεαπλασιάζει την είσοδό της. Όντως:

```
*Main> twice threetimes 5
45
*Main>
```

Άσκηση 11.5
Έστω η ταυτοτική συνάρτηση `id`, που ορίζεται ως:[2]

```
id :: a -> a
id x = x
```

1. Με τι ισούνται οι εκφράσεις `(id . f)`, `(f . id)` και `id f`, για κάποια τυχαία συνάρτηση `f`;
2. Αν η `f` είναι τύπου `Char -> Int`, με ποια τιμή στη μεταβλητή τύπου `a`, στον ορισμό της `id`, χρησιμοποιείται σε καθεμία από τις προηγούμενες περιπτώσεις η ταυτοτική συνάρτηση;
3. Τι τύπου πρέπει να είναι η συνάρτηση `f`, για να μην υπάρχει λάθος τύπου στην έκφραση `f id`;

Άσκηση 11.6
Στην Άσκηση 11.4, σας ζητήθηκε να ορίσετε μια συνάρτηση `iter`, η οποία, δεχόμενη στην είσοδο έναν ακέραιο `n`, μια συνάρτηση `f` (τύπου `a -> a`) και κάποιο `x` (τύπου `a`), δίνει αποτέλεσμα, δηλαδή `iter n f x`, αυτό που προκύπτει από τη διαδοχική εφαρμογή `n` φορές, αρχίζοντας από το `x`, της συνάρτησης `f`. Στη συνέχεια, δίνεται ένας ημιτελής ορισμός μιας συνάρτησης `iterf`, η οποία, εφαρμοζόμενη επάνω σε έναν ακέραιο `n` και σε μια συνάρτηση `f`, δίνει αποτέλεσμα μια νέα συνάρτηση, την `iterf n f`, η οποία, όταν εφαρμοστεί επάνω σε κάποιο `x`, να δίνει το ίδιο αποτέλεσμα με την έκφραση `iter n f x`. Συμπληρώστε κατάλληλα τον ημιτελή ορισμό της `iterf`:

```
iterf :: [    ] -> (a -> a) -> [          ]
iterf n f
  | n == 0   = [    ]
  | n > 0    = f . [              ]
```

Άσκηση 11.7
Ορίστε μια συνάρτηση `composeList`, η οποία να εφαρμόζεται επάνω σε μια λίστα από συναρτήσεις και να δίνει αποτέλεσμα μια συνάρτηση που θα είναι η σύνθεση (κατά σειρά) των συναρτήσεων αυτών.

[2] Και η συνάρτηση αυτή είναι ορισμένη στο πρελούδιο της Haskell, με αποτέλεσμα να μπορεί να χρησιμοποιείται χωρίς να οριστεί.

11.4 Συναρτήσεις πολλών ορισμάτων – Currying

Όταν ορίζουμε τον τύπο μιας συνάρτησης `f` ως:

```
f :: a -> b -> c
```

εννοούμε ότι πρόκειται για μια συνάρτηση η οποία εφαρμόζεται επάνω σε δύο ορίσματα (τύπων `a` και `b`, αντίστοιχα) και δίνει αποτέλεσμα τύπου `c`. Βλέποντας αυτήν τη δήλωση, θα αναρωτιέστε εάν το σύμβολο `->` έχει προκαθορισμένη προσεταιριστικότητα. Δηλαδή, υπάρχει ενδεχόμενο το `a -> b -> c` να είναι το ίδιο με κάποιο από τα `(a -> b) -> c` ή `a -> (b -> c)` (ή ίσως και με τα δύο); Επειδή το σύμβολο `->` είναι μόνο δεξιά προσεταιριστικό, ο τύπος `a -> b -> c` ταυτίζεται με τον `a -> (b -> c)`, αλλά όχι με τον `(a -> b) -> c`.

Η δεξιά προσεταιριστικότητα του συμβόλου `->` μας οδηγεί στο συμπέρασμα ότι θα μπορούσαμε κάλλιστα να θεωρήσουμε ισοδύναμα μια συνάρτηση `f`, με τον τύπο που ορίσαμε προηγουμένως, και ως συνάρτηση που εφαρμόζεται σε **ένα** μόνο όρισμα (τύπου `a`), επιστρέφοντας ως αποτέλεσμα μια συνάρτηση (τύπου `b -> c`). Αυτό δεν ισχύει μόνο στη Haskell, αλλά γενικότερα στον συναρτησιακό προγραμματισμό, επειδή έτσι συμβαίνει στον λάμδα λογισμό, που είναι το θεωρητικό υπόβαθρο του συναρτησιακού προγραμματισμού. Έτσι, μπορούμε να πούμε ότι στον συναρτησιακό προγραμματισμό ισχύει το εξής:

Οι συναρτήσεις είναι πάντα συναρτήσεις ενός ορίσματος. Μια συνάρτηση με n ορίσματα ($n > 1$) είναι συνάρτηση ενός ορίσματος που επιστρέφει ως αποτέλεσμα μια συνάρτηση $n-1$ ορισμάτων κ.ο.κ.

Παράδειγμα 11.7
Πολύ απλά, μπορούμε να ορίσουμε τη συνάρτηση `add`, έτσι ώστε αυτή να αθροίζει δύο ακεραίους που της δίνονται ως ορίσματα:

```
add :: Int -> Int -> Int
add x y = x+y
```

Έχει νόημα η έκφραση `add 3`; Ναι, επειδή ο τύπος της `add` μπορεί να γραφεί και ως `Int -> (Int -> Int)`, το `add 3` έχει αποτέλεσμα μια συνάρτηση (τύπου `Int -> Int`), η οποία, εφαρμοζόμενη επάνω σε έναν ακέραιο `y`, δίνει αποτέλεσμα τον ακέραιο `3+y`.

Παράδειγμα 11.8
Στο Παράδειγμα 11.6 είδαμε τον ορισμό μιας συνάρτησης `twice`, η οποία, όταν εφαρμοστεί επάνω σε μια συνάρτηση `f`, δίνει αποτέλεσμα τη σύνθεση της συνάρτησης `f` με τον εαυτό της. Θα μπορούσαμε να είχαμε ορίσει τη συνάρτηση `twice` και ως εξής:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Δηλαδή, η `twice`, όταν εφαρμόζεται επάνω σε μια συνάρτηση τύπου `a -> a` και σε ένα `x` τύπου `a`, δίνει αποτέλεσμα αυτό που προκύπτει από τη διπλή εφαρμογή της `f` επάνω στο `x`. □

Στο Παράδειγμα 10.8, ορίστηκαν δύο συναρτήσεις, οι `sumdiff` και `other_sumdiff`, οι οποίες υπολόγιζαν το άθροισμα και τη διαφορά δύο πραγματικών αριθμών. Οι συναρτήσεις αυτές, αν και έδιναν τα ίδια αποτελέσματα, ήταν διαφορετικές στη φύση τους. Ενώ

η `sumdiff` ήταν τύπου `Float -> Float -> (Float,Float)`, δηλαδή συνάρτηση με δύο πραγματικούς αριθμούς ως ορίσματα και αποτέλεσμα ένα ζευγάρι από πραγματικούς αριθμούς (ή, ισοδύναμα, συνάρτηση με έναν πραγματικό αριθμό ως όρισμα και αποτέλεσμα μια συνάρτηση που έχει πραγματικό αριθμό ως όρισμα και αποτέλεσμα ένα ζευγάρι πραγματικών αριθμών), η `other_sumdiff` ήταν τύπου `(Float,Float) -> (Float,Float)`, δηλαδή έπαιρνε στην είσοδο ένα ζευγάρι πραγματικών αριθμών και έδινε στην έξοδο επίσης ένα ζευγάρι πραγματικών αριθμών.

Κάθε συνάρτηση που εφαρμόζεται επάνω σε ένα ζευγάρι τύπου `(a,b)` και δίνει αποτέλεσμα τύπου `c`, δηλαδή είναι τύπου `(a,b) -> c`, μπορεί πολύ εύκολα να μετασχηματιστεί σε μια συνάρτηση με δύο ορίσματα (τύπων `a` και `b`, αντίστοιχα), δίνοντας επίσης αποτέλεσμα τύπου `c`. Αυτή η διαδικασία μετασχηματισμού ονομάζεται **currying**, προς τιμήν του Haskell Curry, που τη χρησιμοποίησε σε εκτεταμένο βαθμό. Μάλιστα, ο μετασχηματισμός αυτός μπορεί να οριστεί στη Haskell και με μια συνάρτηση ανώτερης τάξης, την `curry`, ως εξής:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
```

Μερικές φορές είναι χρήσιμος και ο αντίστροφος μετασχηματισμός από το currying, με τον οποίο μετασχηματίζεται μια συνάρτηση δύο ορισμάτων σε μια άλλη που εφαρμόζεται σε ζευγάρι. Ο μετασχηματισμός αυτός θα μπορούσε να υλοποιηθεί μέσω της συνάρτησης `uncurry`, ως εξής:

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

Οι συναρτήσεις `curry` και `uncurry` είναι ορισμένες στο πρελούδιο της Haskell.

Άσκηση 11.8
Επιλέξτε ως σωστή μία από τις εξής δύο δηλώσεις:

1. *Οι συναρτήσεις `iter` και `iterf` των Ασκήσεων 11.4 και 11.6, αντίστοιχα, είναι δύο ουσιαστικά διαφορετικές συναρτήσεις.*
2. *Οι συναρτήσεις `iter` και `iterf` των Ασκήσεων 11.4 και 11.6, αντίστοιχα, δεν έχουν ουσιαστική διαφορά.*

Άσκηση 11.9
Θα μπορούσαμε να ορίσουμε μια συνάρτηση `flip`, η οποία να εφαρμόζεται επάνω σε μια συνάρτηση δύο ορισμάτων και να τη μετασχηματίζει σε μια άλλη, επίσης δύο ορισμάτων, η οποία όμως να δέχεται τα ορίσματά της με αντίστροφη σειρά, ως εξής:

```
flip f x y = f y x
```

Επιλέξτε ποιος (ή ποιοι) από τους τύπους που δίνονται στη συνέχεια είναι κατάλληλος για τον ορισμό τύπου `flip :: ...` της `flip`.

1. `flip :: a -> (b -> c) -> b -> (a -> c)`
2. `flip :: (a -> a -> a) -> (a -> a -> a)`

3. `flip :: a -> b -> c -> (b -> a -> c)`
4. `flip :: (a -> b -> c) -> (b -> a -> c)`
5. `flip :: (a -> a -> b) -> a -> a -> b`
6. `flip :: a -> b -> c -> b -> a -> c`
7. `flip :: (a -> b -> c) -> b -> a -> c`
8. `flip :: (a -> b) -> c -> b -> (a -> c)`
9. `flip :: (a -> b -> a) -> b -> a -> a`

Άσκηση 11.10
Ορίστε μια συνάρτηση `total` με τύπο:

```
total :: (Int -> Int) -> (Int -> Int)
```

η οποία, όταν εφαρμόζεται σε μια συνάρτηση `f` (τύπου `Int -> Int`), τότε το αποτέλεσμα `total f` που δίνει να είναι μια συνάρτηση η οποία, όταν εφαρμόζεται σε έναν ακέραιο `n`, να υπολογίζει το `f 0 + f 1 + ... + f n`.

11.5 Αποδείξεις ιδιοτήτων

Πολλές φορές, αφού έχουμε γράψει ένα πρόγραμμα σε κάποια γλώσσα προγραμματισμού, μας ενδιαφέρει να γνωρίζουμε αν η συμπεριφορά του παρουσιάζει συγκεκριμένη ιδιότητα, για κάθε πιθανή είσοδο που μπορεί να δεχθεί. Κάτι τέτοιο είναι, εν γένει, ιδιαίτερα δύσκολο να αποδειχθεί, για την περίπτωση κάποιας τυχαίας γλώσσας προγραμματισμού. Όμως, στον συναρτησιακό προγραμματισμό, όπως και στη Haskell, οι αποδείξεις ιδιοτήτων των προγραμμάτων είναι δυνατόν να γίνουν σχετικά εύκολα. Αυτό συμβαίνει επειδή ο συναρτησιακός προγραμματισμός είναι πολύ καλά θεμελιωμένος επάνω στα μαθηματικά. Ας δούμε, όμως, μέσω κάποιων παραδειγμάτων, πώς μπορούμε να διατυπώσουμε τέτοιου είδους αποδείξεις ιδιοτήτων.

Παράδειγμα 11.9
Θυμηθείτε τους ορισμούς των συναρτήσεων `swap`, στο Παράδειγμα 10.7, και `twice`, στο Παράδειγμα 11.6. Θα ήταν ενδιαφέρον αν μπορούσαμε να αποδείξουμε ότι:

```
twice swap (i,j) = (i,j)
```

για κάθε ζευγάρι `(i,j)` από ακεραίους. Η ιδιότητα αυτή μπορεί να αποδειχθεί ως εξής:

```
twice swap (i,j) =
(swap . swap) (i,j) =
swap (swap (i,j)) =
swap (j,i) =
(i,j)
```

Έτσι, προκύπτει το ενδιαφέρον συμπέρασμα ότι, αν εφαρμόσουμε δύο φορές διαδοχικά τη συνάρτηση `swap` σε ένα ζευγάρι ακεραίων, θα καταλήξουμε να πάρουμε το ίδιο ζευγάρι ως αποτέλεσμα.[3] □

Πολλές φορές ενδιαφερόμαστε να αποδείξουμε κάποιες ιδιότητες που πρέπει να ισχύουν για κάθε τιμή μιας παραμέτρου n από τους φυσικούς (μη αρνητικούς ακεραίους) αριθμούς. Δηλαδή, θέλουμε να αποδείξουμε ότι ισχύει το $P(n)$, για κάθε φυσικό αριθμό n. Για την περίπτωση αυτή, έχουμε στη διάθεσή μας ένα πολύ ισχυρό μαθηματικό εργαλείο, την **επαγωγή**. Η μέθοδος της επαγωγής εφαρμόζεται ως εξής:

Προκειμένου να αποδειχθεί η ιδιότητα $P(n)$, για κάθε φυσικό αριθμό n, αρκεί:

- *Να αποδειχθεί ότι ισχύει το $P(0)$.*
- *Για κάθε $n > 0$, με την υπόθεση ότι ισχύει το $P(n-1)$, να αποδειχθεί ότι ισχύει και το $P(n)$.*

Ας δούμε πώς μπορούμε να εφαρμόσουμε την επαγωγή, με τη βοήθεια ενός παραδείγματος.

Παράδειγμα 11.10
Θυμηθείτε τον ορισμό του παραγοντικού, που δώσαμε στο Παράδειγμα 10.4, μέσω της συνάρτησης `fact`, καθώς και τον ορισμό της συνάρτησης `powertwo`, για τον υπολογισμό του 2^n, για κάθε φυσικό αριθμό `n`, που δώσαμε στην Άσκηση 11.4. Είναι δυνατόν να αποδείξουμε ότι ισχύει:

```
fact (n+1) >= powertwo n
```

για κάθε φυσικό αριθμό `n`; Ας εφαρμόσουμε την επαγωγή:

- Για `n = 0`, η ιδιότητα που μας ενδιαφέρει ισχύει (ως ισότητα), αφού:
  `fact (0+1) = fact 1 = 1 * fact (1-1) = 1 * fact 0 = 1 * 1 = 1`
  και:
  `powertwo 0 = iter 0 double 1 = 1`
- Έστω ότι ισχύει η ιδιότητά μας για την περίπτωση του `n-1`, όπου `n > 0`, δηλαδή:
  `fact (n-1+1) >= powertwo (n-1)`
  ή, αλλιώς:
  `fact n >= powertwo (n-1)`
  Θα πρέπει να αποδείξουμε ότι ισχύει η ιδιότητα για την περίπτωση του `n`, δηλαδή:
  `fact (n+1) >= powertwo n`
  Αφού όμως ισχύει `n > 0`, έχουμε, λόγω και της επαγωγικής μας υπόθεσης και των ορισμών των συναρτήσεων `iter` και `double` από την Άσκηση 11.4:
  ```
  fact (n+1) =
  (n+1) * fact (n+1-1) =
  (n+1) * fact n >=
  ```

[3] Αυτό ίσως να μας φαίνεται προφανές και ανάξιο λόγου, για να καταφύγουμε σε μια τυπική απόδειξή του. Δεν είναι όμως όλες οι ιδιότητες που μας ενδιαφέρει να αποδείξουμε σε προγράμματα Haskell τόσο απλές. Στις περιπτώσεις αυτές, η μεθοδολογία της απόδειξης ιδιοτήτων με τυπικό τρόπο, μέσω των ορισμών των εμπλεκόμενων συναρτήσεων, είναι πραγματικά πολύ χρήσιμο εργαλείο.

```
2 * fact n >=
2 * powertwo (n-1) =
2 * (iter (n-1) double 1) =
double (iter (n-1) double 1) =
iter n double 1 =
powertwo n
```

Άρα, αποδείχθηκε η ιδιότητα που μας ενδιαφέρει. □

Μια παραλλαγή της μεθόδου της επαγωγής είναι η **δομική επαγωγή**, η οποία εφαρμόζεται στις περιπτώσεις που θέλουμε να αποδείξουμε μια ιδιότητα για κάθε πιθανή πεπερασμένη λίστα. Ο τρόπος με τον οποίο εφαρμόζουμε τη δομική επαγωγή είναι παρόμοιος με αυτόν της κλασικής επαγωγής επάνω στους φυσικούς αριθμούς. Δηλαδή:

Προκειμένου να αποδειχθεί η ιδιότητα $P(xs)$, για κάθε πεπερασμένη λίστα xs, αρκεί:

- *Να αποδειχθεί ότι ισχύει το $P([\,])$.*
- *Με την υπόθεση ότι ισχύει το $P(xs)$, να αποδειχθεί ότι ισχύει και το $P(x : xs)$.*

Παράδειγμα 11.11
Ας θεωρήσουμε τον ορισμό της συνάρτησης `sum`, από το Παράδειγμα 10.9, που υπολογίζει το άθροισμα των στοιχείων μιας λίστας από ακεραίους, και ας ορίσουμε επίσης μια συνάρτηση `doubleAll`, η οποία να διπλασιάζει τα στοιχεία μιας λίστας από ακεραίους, ως εξής:

```
doubleAll :: [Int] -> [Int]
doubleAll []     = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

Θα θέλαμε να αποδείξουμε, για κάθε λίστα από ακεραίους `xs`, ότι ισχύει:

```
sum (doubleAll xs) = 2 * sum xs
```

Ας εφαρμόσουμε τη μέθοδο της δομικής επαγωγής:

- Για `xs = []`, η ιδιότητα που μας ενδιαφέρει ισχύει, αφού:
  `sum (doubleAll []) = sum [] = 0 = 2 * 0 = 2 * sum []`
- Έστω ότι ισχύει η ιδιότητά μας για την περίπτωση του `xs`, δηλαδή:
  `sum (doubleAll xs) = 2 * sum xs`
  Θα πρέπει να αποδείξουμε ότι ισχύει η ιδιότητα για την περίπτωση του `x:xs`, δηλαδή:
  `sum (doubleAll (x:xs)) = 2 * sum (x:xs)`
  Έχουμε, λοιπόν:

```
sum (doubleAll (x:xs)) =
sum (2*x : doubleAll xs) =
2*x + sum (doubleAll xs) =
2*x + 2 * sum xs =
2 * (x + sum xs) =
2 * sum (x:xs)
```

Άρα, αποδείχθηκε η ιδιότητα που θέλουμε. □

Μερικές φορές, θέλουμε να αποδείξουμε μια ιδιότητα η οποία εμπλέκει δύο λίστες. Για να φέρουμε σε πέρας την απόδειξη, αρκεί να εφαρμόσουμε τη δομική επαγωγή επάνω σε μία από αυτές τις λίστες, αφήνοντας την άλλη ως απλή μεταβλητή. Βέβαια, η επιλογή της λίστας για να δουλέψουμε επάνω της τη δομική επαγωγή είναι δικό μας θέμα, αλλά, αν θέλουμε η απόδειξή μας να έχει αίσιο τέλος, πρέπει να επιλέξουμε την κατάλληλη λίστα, με βάση τους ορισμούς των εμπλεκόμενων συναρτήσεων. Ας δούμε το παράδειγμα που ακολουθεί.

Παράδειγμα 11.12
Ας θεωρήσουμε πάλι τη συνάρτηση `sum`, από το Παράδειγμα 10.9, και ας δώσουμε τον ορισμό της συνάρτησης `++`, για τη συνένωση λιστών, όπως αυτός είναι διατυπωμένος στο πρελούδιο της Haskell:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Υπενθυμίζουμε ότι έχουμε περικλείσει το όνομα της συνάρτησης, στη δήλωση του τύπου της, μέσα σε παρενθέσεις, για να δείξουμε ότι αυτή θα χρησιμοποιηθεί με ενδοθεματική σύνταξη.

Με βάση τους ορισμούς των συναρτήσεων `sum` και `++`, θα θέλαμε να αποδείξουμε ότι, για οποιεσδήποτε λίστες ακεραίων `xs` και `ys`, ισχύει:

```
sum (xs ++ ys) = sum xs + sum ys
```

Για την απόδειξη αυτής της ιδιότητας, επιλέγουμε να εφαρμόσουμε δομική επαγωγή επάνω στην πρώτη λίστα `xs`, γιατί η συνάρτηση της συνένωσης λιστών `++` ορίζεται με αναδρομή επάνω στην πρώτη από τις λίστες προς συνένωση:

- Για `xs = []`, η ιδιότητα που μας ενδιαφέρει ισχύει, αφού:
  `sum ([] ++ ys) = sum ys = 0 + sum ys = sum [] + sum ys`
- Έστω ότι ισχύει η ιδιότητά μας για την περίπτωση του `xs`, δηλαδή:
  `sum (xs ++ ys) = sum xs + sum ys`
  Θα πρέπει να αποδείξουμε ότι ισχύει η ιδιότητα για την περίπτωση του `x:xs`, δηλαδή:
  `sum ((x:xs) ++ ys) = sum (x:xs) + sum ys`
  Έχουμε, λοιπόν:

```
sum ((x:xs) ++ ys) =
sum (x : (xs ++ ys)) =
x + sum (xs ++ ys) =
x + (sum xs + sum ys) =
(x + sum xs) + sum ys =
sum (x:xs) + sum ys
```

Άσκηση 11.11
Θυμηθείτε τον ορισμό της συνάρτησης `powertwo`, που δώσαμε στην Άσκηση 11.4, για τον υπολογισμό του 2^n, για κάθε φυσικό αριθμό `n`. Λαμβάνοντας υπόψη και τους ορισμούς των συναρτήσεων `double` και `iter`, που δόθηκαν επίσης στην Άσκηση 11.4, συμπληρώστε την ημιτελή απόδειξη της ιδιότητας:

```
powertwo (n+1) = 2 * powertwo n
```

που πρέπει να ισχύει για κάθε φυσικό αριθμό `n` και δίνεται στη συνέχεια:

```
powertwo (n+1) =
iter [    ] double [  ] =
[      ] (iter n [          ] ) =
[      ] ( [          ] n) =
[              ]
```

Άσκηση 11.12
Έστω η συνάρτηση `product`, η οποία υπολογίζει το γινόμενο των στοιχείων μιας λίστας από ακεραίους. Η συνάρτηση αυτή μπορεί να οριστεί ως εξής:[4]

```
product :: [Int] -> Int
product []     = 1
product (x:xs) = x * product xs
```

Θεωρήστε επίσης τις συναρτήσεις `doubleAll`, από το Παράδειγμα 11.11, `powertwo`, από την Άσκηση 11.4, και `length`, από το Παράδειγμα 10.11. Αποδείξτε ότι, για οποιαδήποτε λίστα από ακεραίους `xs`, ισχύει:

```
product (doubleAll xs) = product xs * powertwo (length xs)
```

Άσκηση 11.13
Αποδείξτε ότι η συνένωση λιστών, όπως ορίζεται από τη συνάρτηση `++`, που διατυπώσαμε στο Παράδειγμα 11.12, είναι προσεταιριστική. Αποδείξτε, δηλαδή, ότι για οποιεσδήποτε λίστες `xs`, `ys` και `zs` ισχύει:

```
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

11.6 Άλλα χαρακτηριστικά της Haskell

Θα κλείσουμε το κεφάλαιο αυτό με μια επιγραμματική αναφορά σε κάποιες δυνατότητες της Haskell, οι οποίες, παρότι είναι εξαιρετικά χρήσιμες για την ανάπτυξη πολύ ισχυρών προγραμμάτων, η λεπτομερέστερη περιγραφή τους θα ξέφευγε από το σκοπό του παρόντος συγγράμματος. Ειδικότερα:

- Η Haskell προσφέρει τη δυνατότητα των **κλάσεων τύπων**, είτε ως ενσωματωμένων κλάσεων είτε ως κλάσεων οριζόμενων από τον προγραμματιστή. Έτσι, μπορούν να οριστούν συναρτήσεις στις οποίες να εμπλέκονται τύποι από τα στοιχεία μιας κλάσης, πιθανώς με διαφορετικό ορισμό για κάθε τύπο. Αυτό είναι το φαινόμενο της υπερφόρτωσης. Προσφέρεται επίσης και η δυνατότητα της κληρονομικότητας μεταξύ των κλάσεων τύπων, όπως και στον αντικειμενοστραφή προγραμματισμό.

[4] Η `product` είναι ήδη ορισμένη στο πρελούδιο της Haskell, όχι μόνο για ακεραίους, αλλά και για πραγματικούς.

- Πολύ χρήσιμη είναι η δυνατότητα των **αλγεβρικών τύπων** που μπορεί να ορίσει κάποιος. Πιο συνηθισμένοι είναι οι αλγεβρικοί τύποι, των οποίων τα στοιχεία ορίζονται με απαρίθμηση, αλλά και οι αναδρομικοί αλγεβρικοί τύποι, όπως τα δυαδικά δέντρα.
- Η Haskell παρέχει την ευελιξία να οργανωθεί ένα μεγάλο και πολύπλοκο πρόγραμμα, σαν ένα σύνολο από **στοιχεία**. Ένα στοιχείο εγκλωβίζει ένα σύνολο από ορισμούς και, μέσω κατάλληλης διεπαφής, ορίζει ποιους ορισμούς εξάγει σε άλλα στοιχεία και από ποια στοιχεία εισάγει άλλους ορισμούς.
- Οι **αφηρημένοι τύποι δεδομένων** βασίζονται στο μηχανισμό των στοιχείων. Πρόκειται για τύπους, όπως ουρές, στοίβες κτλ., που έχουν κρυφή την υλοποίησή τους, παρέχοντας μόνο τη δυνατότητα να χρησιμοποιηθούν με συγκεκριμένο τρόπο.
- Η Haskell, σε αντίθεση με άλλες συναρτησιακές γλώσσες, έχει ένα μηχανισμό για είσοδο/έξοδο, που δεν αντίκειται στη φιλοσοφία του συναρτησιακού προγραμματισμού, όντας, όμως, εξαιρετικά ισχυρός.

Απαντήσεις ασκήσεων

Απάντηση άσκησης 11.1

Η συνάρτηση `scalProd1` δεν υλοποιεί τον ορισμό του εσωτερικού γινομένου, αφού υπολογίζει το άθροισμα των γινομένων για όλα τα ζευγάρια που μπορούμε να φτιάξουμε με πρώτο συστατικό από την πρώτη λίστα και με δεύτερο από τη δεύτερη, όχι μόνο για τα ζευγάρια με τα ομόλογα στοιχεία. Όντως:

```
*Main> scalProd1 [2.5,3.0,1.8] [1.2,4.3,2.5]
58.4
*Main>
```

Ο σωστός ορισμός είναι αυτός της συνάρτησης `scalProd2`. Στη συνάρτηση αυτή υπολογίζουμε το άθροισμα όλων των γινομένων `x*y`, με τα `x` και `y` να αντιστοιχούν στα συστατικά των ζευγαριών `(x,y)` που είναι στοιχεία της λίστας `zip xs ys`. Θυμηθείτε τη συνάρτηση `zip` του πρελούδιου, η οποία συμπτύσσει τα στοιχεία δύο λιστών σε μία λίστα από ζευγάρια. Τώρα, παίρνουμε το σωστό αποτέλεσμα:

```
*Main> scalProd2 [2.5,3.0,1.8] [1.2,4.3,2.5]
20.4
*Main>
```

Απάντηση άσκησης 11.2

Η συνάρτηση `quicksort` θα μπορούσε να υλοποιηθεί ως εξής:

```
quicksort :: [Int] -> [Int]
quicksort []     = []
quicksort (x:xs) = quicksort [ z | z <- xs, z <= x ] ++
                   [x] ++
                   quicksort [ z | z <- xs, z > x ]
```

Θυμηθείτε τη συνάρτηση `++` από το πρελούδιο, με την οποία μπορούμε να συνενώνουμε λίστες. Έτσι, μπορούμε να χρησιμοποιούμε την `quicksort` ως εξής:

```
*Main> quicksort [4,2,8,3,1,7,5,9,0,6]
[0,1,2,3,4,5,6,7,8,9]
*Main>
```

Απάντηση άσκησης 11.3
Οι ορισμοί θα πρέπει να συμπληρωθούν ως εξής:

```
const1 :: a -> Int
const1 x = 1

hlength :: [a] -> Int
hlength xs = sum (map const1 xs)
```

Η συνάρτηση `const1` είναι η σταθερή συνάρτηση που επιστρέφει `1` ό,τι όρισμα τύπου `a` πάρει στην είσοδο. Η `hlength` κατασκευάζει, μέσω της `map` και της `const1`, μια λίστα που περιέχει το στοιχείο `1` τόσες φορές όσα είναι τα στοιχεία της λίστας εισόδου της `hlength` και, στη συνέχεια, αθροίζει τα στοιχεία αυτής της λίστας με τα `1`. Προφανώς, το αποτέλεσμα ισούται με το μήκος της λίστας εισόδου.

Απάντηση άσκησης 11.4
Στην εκφώνηση της άσκησης, δεν έγινε μνεία στους τύπους του `x` και της συνάρτησης `f`. Αν το `x` είναι τύπου `a` (μεταβλητή τύπου), τότε η `f` πρέπει να είναι τύπου `a -> a`, αφού, όταν εφαρμόζεται επάνω σε κάποιο `x`, πρέπει να δίνει αποτέλεσμα επάνω στο οποίο να μπορούμε να εφαρμόσουμε πάλι την `f`. Ύστερα από αυτές τις παρατηρήσεις, μπορούμε να ορίσουμε τη συνάρτηση `iter` ως εξής:

```
iter :: Int -> (a -> a) -> a -> a
iter n f x
  | n == 0   = x
  | n > 0    = f (iter (n-1) f x)
```

Η συνάρτηση `powertwo` ορίζεται πολύ απλά, μέσω της `iter` και της `double`, αν σκεφτούμε ότι το 2^n προκύπτει αν, αρχίζοντας από το `1`, εφαρμόσουμε σε αυτό `n` φορές την `double`. Δηλαδή:

```
powertwo :: Int -> Int
powertwo n = iter n double 1
```

Τέλος, εύκολα μπορούμε να δούμε ότι η εφαρμογή `n` φορές της συνάρτησης `square` επάνω στο `3`, δηλαδή η έκφραση `iter n square 3`, δίνει το 3^{2^n}. Οπότε:

```
*Main> powertwo 10
1024
*Main> iter 4 square 3
43046721
*Main>
```

Δεν ισχύει $3^{16} = 43046721$;

Απάντηση άσκησης 11.5
Οι απαντήσεις στα ερωτήματα που τέθηκαν είναι οι ακόλουθες:

1. Η έκφραση `(id . f)` είναι μια συνάρτηση, η οποία εφαρμόζει την ταυτοτική συνάρτηση επάνω στο αποτέλεσμα που επιστρέφει η συνάρτηση `f`. Συνεπώς, αυτό το αποτέλεσμα δεν αλλάζει, άρα η έκφραση `(id . f)` είναι ουσιαστικά η ίδια η συνάρτηση `f`.

Η έκφραση `(f . id)` είναι μια συνάρτηση, η οποία εφαρμόζει τη συνάρτηση `f` επάνω στο αποτέλεσμα που επιστρέφει η ταυτοτική συνάρτηση. Άρα, έχει την ίδια λειτουργία με τη συνάρτηση `f`.

Η έκφραση `id f` εφαρμόζει την ταυτοτική συνάρτηση επάνω στη συνάρτηση `f`, οπότε το αποτέλεσμα είναι η ίδια η συνάρτηση `f`.

2. Αν η `f` εφαρμόζεται σε δεδομένα τύπου `Char` και παράγει αποτελέσματα τύπου `Int`, τότε η μεταβλητή τύπου `a`, στον ορισμό της `id`, χρησιμοποιείται με τιμή `Int` στην έκφραση `(id . f)`, με τιμή `Char` στην έκφραση `(f . id)` και με τιμή `Char -> Int` στην έκφραση `id f`.
3. Για να μην υπάρχει λάθος τύπου στην έκφραση `f id`, πρέπει η `f` να εφαρμόζεται σε συναρτήσεις που έχουν τον ίδιο τύπο με την `id`, δηλαδή `a -> a`. Επειδή δεν μας ενδιαφέρει τι τύπου θα είναι το αποτέλεσμα, η `f` θα πρέπει να έχει οριστεί ως εξής:

```
f :: (a -> a) -> b
```

Απάντηση άσκησης 11.6
Ο ορισμός της συνάρτησης `iterf` θα πρέπει να συμπληρωθεί ως εξής:

```
iterf :: Int -> (a -> a) -> (a -> a)
iterf n f
  | n == 0   = id
  | n > 0    = f . iterf (n-1) f
```

Ο ορισμός της `iterf` εκφράζει ότι `0` εφαρμογές της `f` ισοδυναμούν με την εφαρμογή της ταυτοτικής συνάρτησης `id` και `n` εφαρμογές της `f` προκύπτουν από τη σύνθεση της `f` με `n-1` εφαρμογές της `f`.

Απάντηση άσκησης 11.7
Η συνάρτηση `composeList` θα πρέπει να οριστεί ως εξής:

```
composeList :: [a -> a] -> (a -> a)
composeList []     = id
composeList (f:fs) = f . (composeList fs)
```

Οι συναρτήσεις που αποτελούν μέλη της λίστας στην οποία εφαρμόζεται η `composeList` δεν μπορεί παρά να είναι ίδιου τύπου, αφού αυτό επιβάλλεται από τον ορισμό της δομής της λίστας. Επίσης, δεδομένου ότι το αποτέλεσμα κάθε συνάρτησης δίνεται ως είσοδος στην προηγούμενή της συνάρτηση στη λίστα, προκύπτει ότι το πεδίο ορισμού των συναρτήσεων αυτών πρέπει να ταυτίζεται με το πεδίο τιμών τους. Άρα, οι συναρτήσεις αυτές θα πρέπει να είναι τύπου `a -> a`. Ο ορισμός της συνάρτησης `composeList` είναι αναδρομικός, δηλαδή στη γενική περίπτωση πρέπει να συνθέσουμε μια συνάρτηση με το αποτέλεσμα της σύνθεσης των υπολοίπων, αλλά, στην περίπτωση που έχουμε κενή λίστα από συναρτήσεις, το αποτέλεσμα πρέπει να είναι η ταυτοτική συνάρτηση. Να ένα παράδειγμα:

```
*Main> composeList [tail,init,reverse,init,tail] "function"
"itcn"
*Main>
```

Απάντηση άσκησης 11.8
Η σωστή δήλωση είναι η υπ' αριθμόν 2. Καταρχάς, δεν υπάρχει διαφορά στους τύπους των δύο συναρτήσεων. Ο τύπος `Int -> (a -> a) -> a -> a` της `iter` είναι ίδιος με τον τύπο `Int -> (a -> a) -> (a -> a)` της `iterf`, λόγω της δεξιάς προσεταιρεστικότητας του `->`. Όσον αφορά τους ορισμούς αυτούς καθαυτούς, και εδώ δεν υπάρχει ουσιαστική διαφορά. Απλώς, η `iter` είναι ορισμένη με τρία ορίσματα, ενώ η `iterf` με δύο, με το επιπλέον (τρίτο) όρισμα της `iter` να είναι αυτό επάνω στο οποίο πρέπει να εφαρμοστεί η `iterf`, για να δώσουν οι δύο συναρτήσεις τα ίδια αποτελέσματα.

Απάντηση άσκησης 11.9
Δύο είναι οι σωστοί τύποι για τη συνάρτηση `flip`, οι 4 και 7, οι οποίοι εκφράζουν τον ίδιο ακριβώς τύπο συνάρτησης, λόγω της δεξιάς προσεταιριστικότητας του `->`. Τους θυμίζουμε:

4. `flip :: (a -> b -> c) -> (b -> a -> c)`
7. `flip :: (a -> b -> c) -> b -> a -> c`

Η `flip` δέχεται ως είσοδο μια συνάρτηση τύπου `a -> b -> c` (με εισόδους τύπων `a` και `b` και αποτέλεσμα τύπου `c`) και παράγει μια συνάρτηση τύπου `b -> a -> c` (με εισόδους τύπων `b` και `a` και αποτέλεσμα τύπου `c`), όπως περιγράφεται στην εκδοχή 4. Στην εκδοχή 7, η `flip`, εκτός από τη συνάρτηση επάνω στην οποία εφαρμόζεται, δέχεται στην είσοδό της και τα δεδομένα τύπων `b` και `a`, επάνω στα οποία θα εφαρμοστεί η μετασχηματισμένη συνάρτηση, για να παραχθεί το αποτέλεσμα τύπου `c`. Η συνάρτηση `flip` είναι ορισμένη μέσα στο πρελούδιο της Haskell.

Απάντηση άσκησης 11.10
Η συνάρτηση `total` θα μπορούσε να υλοποιηθεί ως εξής:

```
total :: (Int -> Int) -> (Int -> Int)
total f n = sum (map f [0..n])
```

Εναλλακτικά, θα μπορούσε να υλοποιηθεί και έτσι:

```
total :: (Int -> Int) -> (Int -> Int)
total f = sum . ((map f) . (enumFromTo 0))
```

Η συνάρτηση `enumFromTo` είναι ορισμένη μέσα στο πρελούδιο της Haskell, έτσι ώστε, όταν εφαρμοστεί σε δύο ακεραίους, `n` και `m`, να επιστρέφει τη λίστα όλων των ακεραίων, από τον `n` έως τον `m`, συμπεριλαμβανομένων. Οπότε, η έκφραση `enumFromTo 0` επιστρέφει μια συνάρτηση, η οποία, όταν εφαρμοστεί σε έναν ακέραιο `n`, επιστρέφει τη λίστα `[0,1,2,...,n]`.

Απάντηση άσκησης 11.11
Η απόδειξη της ιδιότητας που μας ενδιαφέρει πρέπει να συμπληρωθεί ως εξής:

```
powertwo (n+1) =
iter (n+1) double 1 =
double (iter n double 1) =
double (powertwo n) =
2 * powertwo n
```

Απάντηση άσκησης 11.12
Η ζητούμενη απόδειξη μπορεί να γίνει με τη μέθοδο της δομικής επαγωγής:

- Για `xs = []`, η ιδιότητα που μας ενδιαφέρει ισχύει, αφού:
  `product (doubleAll []) = product [] = 1`
  και:
  ```
  product [] * powertwo (length []) = 1 * powertwo 0 =
       1 * iter 0 double 1 = 1 * 1 = 1
  ```
- Έστω ότι ισχύει η ιδιότητά μας για την περίπτωση του `xs`, δηλαδή:
  `product (doubleAll xs) = product xs * powertwo (length xs)`
  Θα πρέπει να αποδείξουμε ότι ισχύει η ιδιότητα για την περίπτωση του `x:xs`, δηλαδή:
  ```
  product (doubleAll (x:xs)) =
       product (x:xs) * powertwo (length (x:xs))
  ```
  Έχουμε λοιπόν:
  ```
  product (doubleAll (x:xs)) =
  product (2*x : doubleAll xs) =
  (2*x) * product (doubleAll xs) =
  (2*x) * (product xs * powertwo (length xs)) =
  2 * (x * product xs) * powertwo (length xs) =
  2 * product (x:xs) * powertwo (length xs) =
  product (x:xs) * (2 * powertwo (length xs)) =
  product (x:xs) * double (powertwo (length xs)) =
  product (x:xs) * double (iter (length xs) double 1) =
  product (x:xs) * double (iter ((1 + length xs) - 1) double 1) =
  product (x:xs) * iter (1 + length xs) double 1 =
  product (x:xs) * iter length (x:xs) double 1 =
  product (x:xs) * powertwo (length (x:xs))
  ```

Απάντηση άσκησης 11.13
Η ζητούμενη απόδειξη μπορεί να γίνει με δομική επαγωγή επάνω στη λίστα `xs`, ως εξής:

- Για `xs = []`, η ιδιότητα που μας ενδιαφέρει ισχύει, αφού:
  `[] ++ (ys ++ zs) = ys ++ zs`
  και:
  `([] ++ ys) ++ zs = ys ++ zs`
- Έστω ότι ισχύει η ιδιότητά μας για την περίπτωση του `xs`, δηλαδή:
  `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`
  Θα πρέπει να αποδείξουμε ότι ισχύει η ιδιότητα για την περίπτωση του `x:xs`, δηλαδή:
  `(x:xs) ++ (ys ++ zs) = ((x:xs) ++ ys) ++ zs`
  Έχουμε λοιπόν:
  ```
  (x:xs) ++ (ys ++ zs) =
  x : (xs ++ (ys ++ zs)) =
  x : ((xs ++ ys) ++ zs) =
  (x : (xs ++ ys)) ++ zs =
  ((x:xs) ++ ys) ++ zs
  ```

Προβλήματα

Πρόβλημα 11.1
Ορίστε στη Haskell τη συνάρτηση:

```
intersection :: [Int] -> [Int] -> [Int]
```

η οποία να επιστρέφει τη λίστα των κοινών στοιχείων δύο δεδομένων λιστών. Θεωρήστε ότι οι λίστες επάνω στις οποίες εφαρμόζεται η συνάρτηση δεν έχουν επαναλαμβανόμενα στοιχεία.

Πρόβλημα 11.2
Ορίστε στη Haskell τη συνάρτηση:

```
divisors :: Int -> [Int]
```

η οποία, για δεδομένο ακέραιο, να επιστρέφει τη λίστα των διαιρετών του. Για παράδειγμα:

```
*Main> divisors 20
[1,2,4,5,10,20]
*Main> divisors 87
[1,3,29,87]
*Main> divisors 97
[1,97]
*Main>
```

Πρόβλημα 11.3
Ορίστε στη Haskell τη συνάρτηση `isPrime`, η οποία να ελέγχει αν δεδομένος ακέραιος είναι πρώτος. Για παράδειγμα:

```
*Main> isPrime 1
False
*Main> isPrime 2
True
*Main> isPrime 17
True
*Main> isPrime 25
False
*Main>
```

Πρόβλημα 11.4
Ορίστε στη Haskell τη συνάρτηση `coprimes`, η οποία να ελέγχει αν δύο δεδομένοι ακέραιοι είναι πρώτοι μεταξύ τους, δηλαδή αν έχουν μοναδικό κοινό διαιρέτη το 1. Για παράδειγμα:

```
*Main> coprimes 99 300
False
*Main> coprimes 147 500
True
*Main>
```

Πρόβλημα 11.5
Ένας θετικός ακέραιος ονομάζεται τέλειος όταν το άθροισμα των διαιρετών του, εξαιρουμένου του εαυτού του, ισούται με τον ίδιο τον αριθμό. Ορίστε στη Haskell τη συνάρτηση `isPerfect`, η οποία να ελέγχει αν δεδομένος ακέραιος είναι τέλειος. Για παράδειγμα:

```
*Main> isPerfect 6
True
*Main> isPerfect 20
False
*Main> isPerfect 28
True
*Main> isPerfect 496
True
*Main> isPerfect 500
False
*Main>
```

Πρόβλημα 11.6
Γράψτε πρόγραμμα Haskell το οποίο να είναι σε θέση να βρει όλες τις ουσιωδώς διαφορετικές (δηλαδή, χωρίς συμμετρίες) τετράδες θετικών ακεραίων (a, b, c, d), για τις οποίες να ισχύει $a^3 + b^3 = c^3 + d^3 = n$, όπου n κάποιος τετραψήφιος το πολύ ακέραιος αριθμός.

Πληροφοριακά, υπάρχουν μόνο δύο τέτοιες τετράδες, η $(1, 12, 9, 10)$, αφού $1^3 + 12^3 = 9^3 + 10^3 = 1729$, και η $(2, 16, 9, 15)$, αφού $2^3 + 16^3 = 9^3 + 15^3 = 4104$.

Είναι δυνατόν το πρόγραμμά σας να βρει όλες τις λύσεις όταν το n είναι πενταψήφιος το πολύ; Υπάρχουν και άλλες οκτώ λύσεις, εκτός από τις δύο προηγούμενες, οι $(2, 24, 18, 20)$, $(2, 34, 15, 33)$, $(3, 36, 27, 30)$, $(4, 32, 18, 30)$, $(9, 34, 16, 33)$, $(10, 27, 19, 24)$, $(12, 40, 31, 33)$ και $(17, 39, 26, 36)$.

Πρόβλημα 11.7
Ορίστε στη Haskell τη συνάρτηση:

```
combinations :: Int -> [a] -> [[a]]
```

η οποία να επιστρέφει σε μια λίστα όλους τους συνδυασμούς στοιχείων δεδομένης λίστας ανά n, για δεδομένο n. Για παράδειγμα:

```
*Main> combinations 3 "abcde"
["abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde"]
*Main>
```

Πρόβλημα 11.8
Η συνάρτηση `foldr` της Haskell ορίζεται ως εξής:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Συμπληρώστε τον ορισμό της συνάρτησης, δίνοντας και τους τύπους δεδομένων εισόδου και εξόδου, εξηγήστε τη λειτουργία της και δώστε ένα παράδειγμα χρήσης της.

Πρόβλημα 11.9
Αν οι ορισμοί των συναρτήσεων `fork` και `cross` ήταν οι εξής:

```
fork (f, g) x = (f x, g x)
cross (f, g) = fork (f . fst, g . snd)
```

πώς θα έπρεπε να είχαν οριστεί οι τύποι τους;

Πρόβλημα 11.10
Με βάση τον ορισμό της συνάρτησης `++` της Haskell που δόθηκε στο Παράδειγμα 11.12, αποδείξτε, με τη βοήθεια της δομικής επαγωγής, ότι ισχύει:

```
xs ++ [] = xs
```

Πρόβλημα 11.11
Έστω ο εξής ορισμός της συνάρτησης `reverse` της Haskell (δεν είναι απαραίτητο ότι η συνάρτηση αυτή ορίζεται έτσι ακριβώς στο πρελούδιο):

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Αποδείξτε, μέσω δομικής επαγωγής, τις ιδιότητες:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
reverse (reverse xs) = xs
```

Πρόβλημα 11.12
Αν ορισμός της συνάρτησης `length` είναι ο:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

αποδείξτε, με δομική επαγωγή, ότι ισχύει:

```
length (xs ++ ys) = length xs + length ys
```

Βιβλιογραφικές αναφορές

[1] S. Thompson, *Haskell: The Craft of Functional Programming*, Addison Wesley, 2011.

[2] R. Bird, *Thinking Functionally with Haskell*, Cambridge University Press, 2015.

Κεφάλαιο 12
Τα εσωτερικά του συναρτησιακού προγραμματισμού

Σύνοψη

Στο κεφάλαιο αυτό παρουσιάζεται ο λάμδα λογισμός, ένα μαθηματικό σύστημα στο οποίο βασίζονται όλες οι σύγχρονες γλώσσες συναρτησιακού προγραμματισμού. Επίσης, γίνεται σύντομη περιγραφή των συνδυαστών, που είναι ενσωματωμένες συναρτήσεις στον λάμδα λογισμό και μπορούν να διατυπωθούν και ως λάμδα αφαιρέσεις. Ακόμα, εξετάζονται οι δύο καθιερωμένες σειρές αναγωγής στον λάμδα λογισμό, η εφαρμοστική σειρά αναγωγής και η κανονική σειρά αναγωγής, ενώ διατυπώνονται τα δύο θεωρήματα των Church και Rosser. Τέλος, επιδεικνύεται πώς είναι δυνατόν οι λάμδα εκφράσεις στον λάμδα λογισμό να αναπαρασταθούν ως γράφοι και πώς εφαρμόζεται μια διαδικασία αναγωγής στους γράφους αυτούς, με σκοπό την απλοποίησή τους.

Προαπαιτούμενη γνώση

Για την κατανόηση του κεφαλαίου, ο αναγνώστης απαιτείται να έχει μελετήσει τα Κεφάλαια 10 και 11.

12.1 Λάμδα λογισμός

Ο **λάμδα λογισμός** είναι ένας συνεπές μαθηματικό σύστημα, που προτάθηκε από τον Church και αποτελεί το θεωρητικό υπόβαθρο όλων των σύγχρονων γλωσσών συναρτησιακού προγραμματισμού [1, 2, 3, 4]. Ταυτόχρονα, αποτελεί το μέσο με το οποίο επιτυγχάνεται η υλοποίηση των συναρτησιακών γλωσσών. Η χρησιμότητα αυτή του λάμδα λογισμού εντοπίζεται στο γεγονός ότι, όντας ένα σύστημα με πολύ απλή σύνταξη και σημασιολογία, αλλά την ίδια στιγμή εκφραστικά ισχυρότατο, μπορεί να λειτουργήσει ως ενδιάμεση μορφή, στην οποία να μετασχηματίζεται ένα οποιοδήποτε συναρτησιακό πρόγραμμα. Οπότε, αυτό που χρειάζεται πλέον είναι η υλοποίηση ενός πολύ περιορισμένου συνόλου από **κανόνες μετασχηματισμού**, και ειδικότερα από **κανόνες αναγωγής**, για τον υπολογισμό εκφράσεων. Όλα τα προηγούμενα ισχύουν, φυσικά, και για τη Haskell. Τόσο η θεμελίωση, όσο και η σχεδίαση, αλλά και η υλοποίηση, της Haskell είναι βασισμένες στον λάμδα λογισμό.

Οι εκφράσεις στον λάμδα λογισμό ονομάζονται **λάμδα εκφράσεις**, μία κατηγορία από τις οποίες αποτελούν οι **λάμδα αφαιρέσεις**. Μια λάμδα αφαίρεση είναι ο ορισμός μιας ανώνυμης συνάρτησης. Ας δούμε ένα παράδειγμα:

Παράδειγμα 12.1
Η έκφραση:

$$\lambda x. + x\ 5$$

είναι μια λάμδα αφαίρεση, που ορίζει μια συνάρτηση, η οποία προσθέτει το 5 στο όρισμα στο οποίο εφαρμόζεται. Είναι πολύ εύλογο να ισχυριστούμε ότι το σύμβολο «λ», σε μια λάμδα αφαίρεση, είναι η μαθηματική έκφραση της διατύπωσης «*η συνάρτηση του*», ενώ το σύμβολο «.» εκφράζει το «*που επιστρέφει ως αποτέλεσμα το*». Οπότε, το $\lambda x. + x\ 5$ μπορεί να διατυπωθεί ως «*η συνάρτηση του x που επιστρέφει ως αποτέλεσμα το $x + 5$*». □

Το «x» στο Παράδειγμα 12.1 είναι μια **μεταβλητή**, η οποία είναι, στην προκειμένη περίπτωση, η **δεσμευμένη μεταβλητή** της λάμδα αφαίρεσης, άρα και της συνάρτησης που ορίζεται μέσω αυτής. Το «$+x\ 5$» είναι το **σώμα** της λάμδα αφαίρεσης. Παρατηρήστε ότι η συνάρτηση που ορίσαμε με αυτόν τον τρόπο δεν έχει όνομα. Για να αναφερθούμε σε αυτήν, όπως θα δούμε στη συνέχεια, πρέπει να χρησιμοποιήσουμε την πλήρη λάμδα αφαίρεση που την ορίζει. Παρατηρήστε, επίσης, ότι στη λάμδα αφαίρεση που διατυπώσαμε στο Παράδειγμα 12.1, χρησιμοποιείται μια άλλη συνάρτηση, αυτή για την πρόσθεση δύο αριθμών, η οποία έχει, όμως, όνομα, το «+». Η συνάρτηση αυτή μπορεί να θεωρηθεί **ενσωματωμένη συνάρτηση**. Στον λάμδα λογισμό, τόσο οι επώνυμες ενσωματωμένες συναρτήσεις, όσο και οι ανώνυμες, που ορίζονται με λάμδα αφαιρέσεις, συντάσσονται με προθεματικό τρόπο. Επίσης, στον ίδιο ορισμό, χρησιμοποιήσαμε και το σύμβολο 5, το οποίο είναι μια **ενσωματωμένη σταθερά**.[1] Τόσο οι μεταβλητές όσο και οι ενσωματωμένες σταθερές αποτελούν περιπτώσεις λάμδα εκφράσεων.

Πού όμως μπορεί να μας χρησιμεύσει μια λάμδα αφαίρεση, εκτός από την εφαρμογή της συνάρτησης που ορίζει σε ένα όρισμα; Πράγματι, αυτό κάνουμε στο επόμενο παράδειγμα:

Παράδειγμα 12.2
Αν έχουμε τη λάμδα αφαίρεση $\lambda x. + x\ 5$, που, όπως είδαμε στο Παράδειγμα 12.1, ορίζει μια συνάρτηση, μπορούμε να την εφαρμόσουμε σε κάποια λάμδα έκφραση, για παράδειγμα το 2, γράφοντας:

$$(\lambda x. + x\ 5)\ 2$$

Περικλείουμε τη λάμδα αφαίρεση μέσα σε παρενθέσεις για να δείξουμε ότι πρέπει να εφαρμοστεί ως σύνολο επάνω στο 2. Η εφαρμογή αυτή δίνει, αρχικά, αποτέλεσμα αυτό που προκύπτει αν αντικατασταθεί στο σώμα της λάμδα αφαίρεσης η δεσμευμένη μεταβλητή της x με το 2, δηλαδή το $+2\ 5$. Τελικά, αυτή η έκφραση μετασχηματίζεται στο 7, λόγω του εσωτερικού ορισμού της σταθεράς (συνάρτησης) $+$. Συμβολικά:

$$\begin{aligned}(\lambda x. + x\ 5)\ 2 &\longrightarrow_\beta\\ +2\ 5 &\longrightarrow_\delta\\ 7&\end{aligned}$$

Αυτό που κάναμε εδώ ήταν η εφαρμογή κάποιων κανόνων μετασχηματισμού, για την απλοποίηση μιας λάμδα έκφρασης. Γι' αυτόν το λόγο, οι συγκεκριμένοι κανόνες ονομάζονται κανόνες αναγωγής. Μέσω του συμβόλου $\longrightarrow$, δείχνουμε ότι μια **αναγώγιμη έκφραση** ανάγεται σε κάποια άλλη. Τα ειδικότερα σύμβολα $\longrightarrow_\beta$ και $\longrightarrow_\delta$ που χρησιμοποιήσαμε δείχνουν ότι ο πρώτος μετασχηματισμός ήταν μια **β-αναγωγή**, ενώ ο δεύτερος μια **δ-αναγωγή**.

[1] Οι ενσωματωμένες συναρτήσεις είναι, ουσιαστικά, και αυτές ενσωματωμένες σταθερές. Τις ονομάσαμε συναρτήσεις, επειδή γι' αυτές έχουν οριστεί κάποιοι κανόνες αναγωγής οι οποίοι απλοποιούν εκφράσεις που προκύπτουν από την εφαρμογή των συγκεκριμένων συναρτήσεων σε άλλες εκφράσεις.

Η β-αναγωγή αφορά την εφαρμογή μιας λάμδα αφαίρεσης σε μια λάμδα έκφραση. Η δ-αναγωγή αναφέρεται σε έναν εσωτερικό ορισμό, που μπορεί να χρησιμοποιηθεί για να δώσει το αποτέλεσμα της εφαρμογής μιας ενσωματωμένης σταθεράς ως συνάρτησης. □

Θα ήταν χρήσιμο στο σημείο αυτό να συνοψίσουμε τι είδους λάμδα εκφράσεις μπορούμε να έχουμε στον λάμδα λογισμό. Μια λάμδα έκφραση $\langle exp \rangle$ μπορεί να είναι:

- Μια μεταβλητή $\langle var \rangle$.
- Μια ενσωματωμένη σταθερά $\langle con \rangle$.
- Μια λάμδα αφαίρεση $\lambda \langle var \rangle . \langle exp \rangle$, όπου $\langle exp \rangle$ είναι επίσης λάμδα έκφραση.
- Μια εγκλωβισμένη μέσα σε παρενθέσεις λάμδα έκφραση, δηλαδή $(\langle exp \rangle)$.
- Η εφαρμογή μιας λάμδα έκφρασης σε κάποια λάμδα έκφραση, δηλαδή $\langle exp \rangle \ \langle exp \rangle$.

Αυτά διατυπώνονται μέσω του BNF συμβολισμού ως εξής:

$$\langle exp \rangle ::= \langle var \rangle \mid \langle con \rangle \mid \lambda \langle var \rangle . \langle exp \rangle \mid (\langle exp \rangle) \mid \langle exp \rangle \ \langle exp \rangle$$

Η δ-αναγωγή της έκφρασης $+2\ 5$, που κάναμε στο Παράδειγμα 12.2, δείχνει, όμως, να μη συμβαδίζει με τον προηγούμενο ορισμό, ο οποίος δηλώνει συντακτικά τις πιθανές λάμδα εκφράσεις, αφού φαίνεται ότι η ενσωματωμένη σταθερά $+$ πρέπει να εφαρμοστεί σε δύο λάμδα εκφράσεις, το 2 και το 5, ενώ ο ορισμός καλύπτει την περίπτωση εφαρμογής μιας λάμδα έκφρασης σε μία μόνο άλλη. Είναι όμως έτσι; Η απάντηση είναι αρνητική, γιατί στον λάμδα λογισμό, αλλά και σε όλες τις γλώσσες συναρτησιακού προγραμματισμού που βασίζονται σε αυτόν, η εφαρμογή συναρτήσεων είναι μια αριστερά προσεταιριστική πράξη. Δηλαδή, το $+2\ 5$ δεν είναι τίποτε άλλο από το $(+2)\ 5$, το οποίο δείχνει ότι το $+$ εφαρμόζεται μόνο επάνω στο 2 και ότι το αποτέλεσμα αυτής της εφαρμογής είναι η λάμδα έκφραση $(+2)$, που αντιπροσωπεύει μια συνάρτηση η οποία προσθέτει το 2 με το όρισμα που της δίνεται. Οπότε, στη συγκεκριμένη περίπτωση, η συνάρτηση αυτή εφαρμόζεται στο 5, για να πάρουμε το τελικό αποτέλεσμα 7. Τελικά, παρατηρούμε ότι και στον λάμδα λογισμό, όπως και στη Haskell, σύμφωνα με αυτά που είδαμε στην Ενότητα 11.4, οι συναρτήσεις είναι ουσιαστικά συναρτήσεις ενός ορίσματος.

Παράδειγμα 12.3
Η λάμδα αφαίρεση $\lambda x . \lambda y . * \ 3 \ (-x \ y)$ αντιπροσωπεύει μια συνάρτηση, η οποία, για δεδομένα x και y, θα πρέπει να επιστρέφει το $3 * (x - y)$. Ουσιαστικά, πρόκειται για τη λάμδα αφαίρεση $\lambda x . (\lambda y . * \ 3 \ (-x \ y))$, δηλαδή για μια λάμδα αφαίρεση της οποίας το σώμα είναι επίσης λάμδα αφαίρεση. Οι παρενθέσεις δεν προσφέρουν, όμως, κάτι στη σαφήνεια της έκφρασης αυτής, οπότε δεν υπάρχει κανένα πρόβλημα αν παραλειφθούν, αφού η εφαρμογή συναρτήσεων είναι αριστερά προσεταιριστική πράξη. Η μεταβλητή y είναι δεσμευμένη μεταβλητή στη λάμδα έκφραση $\lambda y . * \ 3 \ (-x \ y)$, αλλά η x, εντός αυτής της έκφρασης, είναι **ελεύθερη μεταβλητή**. Αυτό σημαίνει ότι θα πρέπει η x να αποκτήσει πρώτα συγκεκριμένη αριθμητική τιμή, αν θέλουμε να εφαρμόσουμε αυτήν τη λάμδα έκφραση επάνω σε έναν αριθμό και να πάρουμε το τελικό αποτέλεσμα, ύστερα από τις κατάλληλες αναγωγές. Αντίθετα, στη λάμδα αφαίρεση $\lambda x . \lambda y . * \ 3 \ (-x \ y)$, επειδή η μεταβλητή x είναι δεσμευμένη, αν εφαρμόσουμε την έκφραση αυτή επάνω σε έναν αριθμό, για παράδειγμα στο 8, τότε θα πάρουμε, μέσω β-αναγωγής, τη λάμδα αφαίρεση $\lambda y . * \ 3 \ (-8 \ y)$, η οποία, αν με τη σειρά της εφαρμοστεί επίσης σε έναν αριθμό, έστω στο 2, θα μας δώσει, πάλι μέσω β-αναγωγής,

την έκφραση $*3\ (-8\ 2)$, η οποία με δύο δ-αναγωγές θα μετασχηματιστεί στο 18. Αυτά γράφονται, συμβολικά, ως εξής:

$$\begin{aligned}
&(\lambda x.\lambda y.*3\ (-x\ y))\ 8\ 2 \longrightarrow_\beta \\
&\quad (\lambda y.*3\ (-8\ y))\ 2 \longrightarrow_\beta \\
&\quad *3\ (-8\ 2) \longrightarrow_\delta \\
&\quad *3\ 6 \longrightarrow_\delta \\
&\quad 18
\end{aligned}$$

Σημειώστε ότι το $(\lambda x.\lambda y.*3\ (-x\ y))\ 8\ 2$ είναι ουσιαστικά το $((\lambda x.\lambda y.*3\ (-x\ y))\ 8)\ 2$, αλλά, λόγω της αριστερής προσεταιριστικότητας της εφαρμογής συναρτήσεων (λάμδα εκφράσεων), οι παρενθέσεις δεν είναι απαραίτητες. □

Ένα πρόβλημα που πρέπει να αντιμετωπιστεί στον λάμδα λογισμό είναι η περίπτωση κατά την οποία μια λάμδα έκφραση περιέχει μια ελεύθερη μεταβλητή, αλλά και μια δεσμευμένη μεταβλητή που έχει το ίδιο όνομα με την ελεύθερη. Ουσιαστικά, πρόκειται για διαφορετικές μεταβλητές, παρότι έχουν το ίδιο όνομα. Αν δεν δοθεί προσοχή σε αυτό το σημείο, δεν θα είναι δυνατόν να γίνουν οι σωστές αναγωγές, όπου χρειάζεται. Δείτε το παράδειγμα που ακολουθεί.

Παράδειγμα 12.4
Έστω ότι θέλουμε να κάνουμε ό,τι αναγωγές μπορούν να γίνουν στη λάμδα έκφραση:

$$(\lambda f.\lambda x.f\ x\ 5)\ (\lambda y.\lambda x.*\ y\ x)\ 7$$

έτσι ώστε να την απλοποιήσουμε κατά το δυνατόν περισσότερο.

Καταρχάς, ένα σημείο ιδιαίτερα ενδιαφέρον σε αυτήν τη λάμδα έκφραση είναι ότι η λάμδα αφαίρεση $(\lambda f.\lambda x.f\ x\ 5)$ πρέπει πρώτα να εφαρμοστεί στη λάμδα έκφραση, που είναι επίσης λάμδα αφαίρεση, $(\lambda y.\lambda x.*\ y\ x)$. Αυτό μπορεί να γίνει αν αντικαταστήσουμε τη δεύτερη λάμδα αφαίρεση στη θέση της μεταβλητής f στο σώμα της πρώτης λάμδα αφαίρεσης.[2] Ουσιαστικά, η πρώτη λάμδα αφαίρεση αντιπροσωπεύει μια συνάρτηση ανώτερης τάξης, σαν αυτές που είδαμε για τη Haskell στην Ενότητα 11.2, αφού δέχεται ως όρισμα μια συνάρτηση. Κάνοντας, λοιπόν, την απαιτούμενη β-αναγωγή, έχουμε:

$$\begin{aligned}
&(\lambda f.\lambda x.f\ x\ 5)\ (\lambda y.\lambda x.*\ y\ x)\ 7 \longrightarrow_\beta \\
&\quad (\lambda x.((\lambda y.\lambda x.*\ y\ x)\ x\ 5))\ 7
\end{aligned}$$

Τώρα, μπορούμε να κάνουμε και μια δεύτερη β-αναγωγή, αντικαθιστώντας το 7 στο σώμα της λάμδα αφαίρεσης $(\lambda x.((\lambda y.\lambda x.*\ y\ x)\ x\ 5))$. Εδώ, όμως, χρειάζεται προσοχή. Με το 7 πρέπει να αντικαταστήσουμε τη δεσμευμένη μεταβλητή x της λάμδα αφαίρεσης, αλλά μόνο εκείνη την εμφάνισή της που είναι ελεύθερη μέσα στο σώμα της λάμδα αφαίρεσης, και όχι την εμφάνισή της στην εσωτερική λάμδα αφαίρεση $(\lambda y.\lambda x.*\ y\ x)$, όπου εκεί είναι δεσμευμένη. Όπως εξηγήσαμε προηγουμένως, πρόκειται για δύο διαφορετικές μεταβλητές, που απλώς τυχαίνει να έχουν το ίδιο όνομα. Οπότε, έχουμε:

[2] Αφού $(\lambda f.\lambda x.f\ x\ 5)\ (\lambda y.\lambda x.*\ y\ x)\ 7 = ((\lambda f.\lambda x.f\ x\ 5)\ (\lambda y.\lambda x.*\ y\ x))\ 7$.

$$
\begin{aligned}
&(\lambda x.((\lambda y.\lambda x. * \ y \ x) \ x \ 5)) \ 7 \longrightarrow_\beta \\
&\quad (\lambda y.\lambda x. * \ y \ x) \ 7 \ 5 \longrightarrow_\beta \\
&\quad (\lambda x. * \ 7 \ x) \ 5 \longrightarrow_\beta \\
&\quad *7 \ 5 \longrightarrow_\delta \\
&\quad 35
\end{aligned}
$$

Στην ακολουθία των αναγωγών που κάναμε προηγουμένως, αντί να αρχίσουμε με την αντικατάσταση του 7 στο σώμα της λάμδα αφαίρεσης $(\lambda x.((\lambda y.\lambda x. * \ y \ x) \ x \ 5))$, θα μπορούσαμε, εάν θέλαμε, να εφαρμόσουμε αρχικά την εσωτερική β-αναγωγή που είναι πιθανή, δηλαδή την εφαρμογή της λάμδα έκφρασης $(\lambda y.\lambda x. * \ y \ x)$ στο x. Εδώ, ίσως να μπαίναμε στον πειρασμό, για να κάνουμε αυτήν την αναγωγή, να αντικαταστήσουμε το x στη θέση της μεταβλητής y στο σώμα της λάμδα αφαίρεσης $(\lambda y.\lambda x. * \ y \ x)$, οπότε να πάρουμε το $(\lambda x. * \ x \ x)$. Κάτι τέτοιο θα ήταν λάθος όμως, γιατί το σώμα αυτό περιέχει ήδη μια λάμδα αφαίρεση με δεσμευμένη μεταβλητή, η οποία έχει το ίδιο όνομα με τη μεταβλητή που πρόκειται να εισαγάγουμε με τη β-αναγωγή. Η λύση στο πρόβλημα αυτό είναι να κάνουμε ένα μετασχηματισμό στην εσωτερική λάμδα αφαίρεση, που ονομάζεται **α-μετατροπή**, με αποκλειστικό σκοπό να μετονομαστεί η μεταβλητή εκείνη που είναι συνυπεύθυνη για την ανεπιθύμητη σύγκρουση των ονομάτων. Όταν κάνουμε α-μετατροπή, χρησιμοποιούμε το σύμβολο $\longleftrightarrow_\alpha$. Έτσι, θα μπορούσαμε να προχωρήσουμε στην εξής ακολουθία από αναγωγές και μετατροπές:

$$
\begin{aligned}
&(\lambda x.((\lambda y.\lambda x. * \ y \ x) \ x \ 5)) \ 7 \longleftrightarrow_\alpha \\
&\quad (\lambda x.((\lambda y.\lambda x'. * \ y \ x') \ x \ 5)) \ 7 \longrightarrow_\beta \\
&\quad (\lambda x.((\lambda x'. * \ x \ x') \ 5)) \ 7 \longrightarrow_\beta \\
&\quad (\lambda x. * \ x \ 5) \ 7 \longrightarrow_\beta \\
&\quad *7 \ 5 \longrightarrow_\delta \\
&\quad 35
\end{aligned}
$$

Παρατηρήστε ότι και με τις δύο διαφορετικές ακολουθίες μετασχηματισμών που εφαρμόσαμε καταλήξαμε στο ίδιο αποτέλεσμα. □

Εκτός από τη β-αναγωγή, τη δ-αναγωγή και την α-μετατροπή, στον λάμδα λογισμό υπάρχει και ένας τέταρτος κανόνας μετασχηματισμού, η **η-μετατροπή**.

Παράδειγμα 12.5
Τι διαφορά έχουν οι λάμδα εκφράσεις $\lambda x. + \ 4 \ x$ και $+4$ στον λάμδα λογισμό; Δεν έχουν καμία διαφορά, γιατί και οι δύο, όταν εφαρμοστούν επάνω σε μια λάμδα έκφραση, όπως στο 2, δίνουν το ίδιο αποτέλεσμα, αφού $(\lambda x. + \ 4 \ x) \ 2 \longrightarrow_\beta (+4) \ 2$. Έτσι, μπορούμε κάλλιστα να γράψουμε:

$$\lambda x. + \ 4 \ x \longleftrightarrow_\eta +4$$

Γενικά, μπορούμε να γράψουμε:

$$\lambda x.E \ x \longleftrightarrow_\eta E$$

υπό την προϋπόθεση ότι η έκφραση E δεν περιέχει ως ελεύθερη μεταβλητή τη x. Αυτό είναι μια η-μετατροπή. □

Θα κλείσουμε αυτήν την ενότητα με ένα σχόλιο που αφορά την αναγκαιότητα των ενσωματωμένων σταθερών (συναρτήσεων) στον λάμδα λογισμό. Ήδη, στα παραδείγματα που προηγήθηκαν, θεωρήσαμε δεδομένες κάποιες ενσωματωμένες σταθερές (2, 5, $+$, $-$, $*$, ...), που αντιπροσώπευαν αριθμούς και αριθμητικές συναρτήσεις. Μάλιστα, θεωρήσαμε δεδομένο και ένα σύνολο από προκαθορισμένες δ-αναγωγές, έτσι ώστε να είμαστε σε θέση να πάρουμε συγκεκριμένα αποτελέσματα από την εφαρμογή των αριθμητικών συναρτήσεων επάνω στους αριθμούς. Συνήθως, στον λάμδα λογισμό, εκτός από αριθμούς και αριθμητικές συναρτήσεις, θεωρούμε δεδομένο και ένα πλήθος από άλλες ενσωματωμένες σταθερές, όπως τις λογικές συναρτήσεις AND, OR και NOT, τις λογικές σταθερές $TRUE$ και $FALSE$, τους τελεστές σύγκρισης $=$, $>$, $\leq$ κτλ., τη συνάρτηση συνθήκης $COND$, τη συνάρτηση $CONS$, για τη δόμηση μιας λίστας από την κεφαλή και την ουρά της, τη σταθερά NIL, για την κενή λίστα, τις συναρτήσεις $HEAD$ και $TAIL$, για την απομόνωση της κεφαλής και της ουράς μιας λίστας, αντίστοιχα, κ.ο.κ. Φυσικά, όπως θεωρούσαμε δεδομένες τις δ-αναγωγές για τις αριθμητικές συναρτήσεις, για παράδειγμα την:

$$+5\ 7 \longrightarrow_\delta 12$$

έτσι έχουμε, και μπορούμε να χρησιμοποιούμε, τις απαιτούμενες δ-αναγωγές, για τις άλλες ενσωματωμένες σταθερές, όπως:

$$\begin{array}{ll} AND\ TRUE\ FALSE & \longrightarrow_\delta FALSE \\ > 8\ 5 & \longrightarrow_\delta TRUE \\ COND\ TRUE\ e_t\ e_f & \longrightarrow_\delta e_t \\ COND\ FALSE\ e_t\ e_f & \longrightarrow_\delta e_f \\ HEAD\ (CONS\ p\ q) & \longrightarrow_\delta p \\ TAIL\ (CONS\ p\ q) & \longrightarrow_\delta q \end{array}$$

Ωστόσο, είναι οι ενσωματωμένες σταθερές απαραίτητες για την πληρότητα του λάμδα λογισμού; Η απάντηση στο ερώτημα αυτό είναι αρνητική. Θα μπορούσαμε πολύ άνετα να δουλέψουμε με τον λάμδα λογισμό, στην πλήρη δύναμή του, χωρίς να έχουμε στη διάθεσή μας έστω και μία ενσωματωμένη σταθερά. Τελικά, είναι δυνατόν όλες οι ενσωματωμένες σταθερές να αντιπροσωπεύονται από κατάλληλες λάμδα αφαιρέσεις, οπότε δεν χρειάζεται να έχουμε δεδομένες τις σχετικές δ-αναγωγές. Όποια πληροφορία δίνει μια δ-αναγωγή μπορεί να προκύψει από μια κατάλληλη ακολουθία β-αναγωγών, α-μετατροπών και η-μετατροπών. Μερικές από τις ενσωματωμένες σταθερές που αναφέραμε ήδη μπορούν να οριστούν ως εξής:

$$\begin{array}{rcl} COND & = & (\lambda x.\lambda y.\lambda z.x\ y\ z) \\ TRUE & = & (\lambda x.\lambda y.x) \\ FALSE & = & (\lambda x.\lambda y.y) \\ CONS & = & (\lambda h.\lambda t.\lambda s.s\ h\ t) \\ HEAD & = & (\lambda c.c\ (\lambda a.\lambda b.a)) \\ TAIL & = & (\lambda c.c\ (\lambda a.\lambda b.b)) \end{array}$$

Για να πάρετε μια ιδέα του τρόπου με τον οποίο μπορείτε να επαληθεύσετε αν κάποιοι από τους ορισμούς αυτούς είναι συμβατοί με τις δ-αναγωγές που προαναφέραμε, δοκιμάστε να αντιμετωπίσετε την Άσκηση 12.2.

Άσκηση 12.1
Συμπληρώστε κατάλληλα την ημιτελή ακολουθία από αναγωγές και μετατροπές που δίνεται στη συνέχεια:

$$
\begin{array}{l}
(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.*\ 3\ x)\ 5\ \boxed{} \\
\quad (\lambda f.\lambda x'.f\ \boxed{}\ )\ (\lambda x.*\ 3\ x)\ 5 \longrightarrow_\beta \\
\quad (\lambda x'.\ \boxed{}\ ((\lambda x.*\ 3\ x)\ x'))\ 5\ \boxed{} \\
\quad (\lambda x.*\ 3\ x)\ ((\lambda x.\ \boxed{}\ )\ 5) \longrightarrow_\beta \\
\quad (\lambda x.*\ 3\ x)\ (*3\ 5)\ \boxed{} \\
\quad (\ \boxed{}\ *\ 3\ x)\ 15 \longrightarrow_\beta \\
\quad \boxed{} \longrightarrow_\delta \\
\quad \boxed{}
\end{array}
$$

Μπορείτε να υποθέσετε ποιος είναι ο ρόλος της λάμδα αφαίρεσης $(\lambda f.\lambda x.f\ (f\ x))$;

Άσκηση 12.2
Σύμφωνα με τους ορισμούς των ενσωματωμένων σταθερών (συναρτήσεων) $CONS$ και $HEAD$, δηλαδή τους:

$$
\begin{array}{rcl}
CONS & = & (\lambda h.\lambda t.\lambda s.s\ h\ t) \\
HEAD & = & (\lambda c.c\ (\lambda a.\lambda b.a))
\end{array}
$$

επαληθεύστε ότι η λάμδα έκφραση $HEAD\ (CONS\ p\ q)$ ανάγεται τελικά στο p (χωρίς, φυσικά, να χρησιμοποιήσετε την εκ των προτέρων δεδομένη δ-αναγωγή).

12.2 Συνδυαστές

Στην Ενότητα 12.1 είδαμε κάποια βασικά χαρακτηριστικά του λάμδα λογισμού. Συγκεκριμένα, αναφερθήκαμε στις λάμδα εκφράσεις και δώσαμε ιδιαίτερη έμφαση σε μια κατηγορία λάμδα εκφράσεων, τις λάμδα αφαιρέσεις. Όπως θα θυμάστε, οι λάμδα αφαιρέσεις είναι το μέσον με το οποίο ορίζουμε συναρτήσεις στον λάμδα λογισμό. Οι συναρτήσεις αυτές, όμως, είναι ανώνυμες. Για να αναφερθούμε σε μια τέτοια συνάρτηση, πρέπει να χρησιμοποιήσουμε την πλήρη λάμδα αφαίρεση που την ορίζει. Τι γίνεται, όμως, αν θέλουμε να ορίσουμε μια αναδρομική συνάρτηση; Γνωρίζουμε πολύ καλά ότι μια αναδρομική συνάρτηση ορίζεται μέσω του εαυτού της. Άρα, θα πρέπει, όταν διατυπώσουμε τον ορισμό της μέσω μιας λάμδα αφαίρεσης, να χρησιμοποιήσουμε την ίδια τη συνάρτηση στο σώμα της λάμδα αφαίρεσης. Αλλά, πώς θα γίνει αυτό, αφού η συνάρτηση δεν έχει όνομα; Υπάρχει περίπτωση να μην μπορούμε να ορίσουμε αναδρομικές συναρτήσεις στον λάμδα λογισμό; Φυσικά όχι, γιατί, αν δεν μπορούσαμε, ο λάμδα λογισμός θα ήταν ένα εντελώς ανίσχυρο εργαλείο.

Στην ενότητα αυτή δίνεται λύση στο πρόβλημα του ορισμού αναδρομικών συναρτήσεων στον λάμδα λογισμό (και όχι μόνο!). Αυτό γίνεται με τη βοήθεια των **συνδυαστών**. Ένας συνδυαστής είναι μια οποιαδήποτε λάμδα αφαίρεση που δεν περιέχει ελεύθερες μεταβλητές. Για τον ορισμό αναδρομικών συναρτήσεων, χρειαζόμαστε έναν συγκεκριμένο συνδυαστή, τον **Y-συνδυαστή**, όπως θα δούμε στη συνέχεια. Καλύτερα, όμως, να διαπραγματευθούμε το θέμα με τη βοήθεια ενός παραδείγματος.

Παράδειγμα 12.6
Έστω ότι μας ενδιαφέρει να ορίσουμε τη συνάρτηση που υπολογίζει το παραγοντικό ενός φυσικού αριθμού. Αν είχαμε τη δυνατότητα να ορίσουμε στον λάμδα λογισμό συναρτήσεις με όνομα, θα γράφαμε για τη συνάρτηση $FACT$ του παραγοντικού:

$$FACT = (\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (FACT\ (-n\ 1))))$$

Αυτό θα μπορούσε να γραφεί και έτσι:[3]

$$FACT = (\lambda f.(\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (f\ (-n\ 1)))))\ FACT$$

Δηλαδή, ζητάμε να βρούμε μια συνάρτηση $FACT$ τέτοια ώστε να ισχύει:

$$FACT = H\ FACT \tag{12.1}$$

όπου:

$$H = (\lambda f.(\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (f\ (-n\ 1))))) \tag{12.2}$$

Με άλλα λόγια, θέλουμε να βρούμε ένα **σταθερό σημείο**[4] $FACT$ της συνάρτησης H, αφού, εφαρμόζοντας την H επάνω στο $FACT$, παίρνουμε αποτέλεσμα πάλι το $FACT$. □

Έστω, λοιπόν, ότι υπάρχει μια συνάρτηση Y, η οποία, όταν εφαρμοστεί επάνω σε μια συνάρτηση, μας επιστρέφει ένα σταθερό σημείο της συνάρτησης αυτής. Για την περίπτωση του Παραδείγματος 12.6, θα πρέπει να ισχύει:

$$FACT = Y\ H \tag{12.3}$$

Συνδυάζοντας τώρα τις σχέσεις (12.1) και (12.3), προκύπτει ότι:

$$Y\ H = H\ (Y\ H) \tag{12.4}$$

[3] Ο μετασχηματισμός αυτός λέγεται **β-αφαίρεση**, επειδή είναι ο αντίστροφος από τη β-αναγωγή. Συνήθως, χρησιμοποιούμε το σύμβολο $\longleftrightarrow_\beta$ μεταξύ δύο λάμδα εκφράσεων, για να δείξουμε ότι η δεύτερη προκύπτει από την πρώτη με β-αναγωγή, και η πρώτη από τη δεύτερη με β-αφαίρεση, ή αντίστροφα. Ομοίως, υπάρχει και η **δ-αφαίρεση**, που είναι η αντίστροφη της δ-αναγωγής, με την έννοια ότι μπορούμε να μετασχηματίσουμε μια ενσωματωμένη σταθερά σε μια λάμδα έκφραση που περιγράφει την εφαρμογή μιας ενσωματωμένης σταθεράς (συνάρτησης) σε άλλη ή άλλες εκφράσεις. Φυσικά, τότε χρησιμοποιούμε το σύμβολο $\longleftrightarrow_\delta$. Για παράδειγμα, μπορούμε να γράψουμε $5 \longleftrightarrow_\delta +3\ 2$.

[4] Σταθερό σημείο μιας συνάρτησης f είναι κάποιο x, για το οποίο ισχύει $f(x) = x$.

Η συνάρτηση Y είναι ο Y-συνδυαστής, που αναφέραμε προηγουμένως, ή, αλλιώς, ο **συνδυαστής σταθερού σημείου**. Αποδεικνύεται ότι ο Y-συνδυαστής μπορεί να γραφεί ως μια λάμδα αφαίρεση:

$$Y = (\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x))) \tag{12.5}$$

Ας δούμε γιατί αυτός ο ορισμός του Y-συνδυαστή επαληθεύει τη βασική ιδιότητα (12.4) που πρέπει να έχει, δηλαδή τη λειτουργία του ως δημιουργού ενός σταθερού σημείου μιας συνάρτησης:

$$\begin{aligned}
&Y\ H = \\
&\quad (\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x)))\ H \longleftrightarrow_\beta \\
&\quad (\lambda x.H\ (x\ x))\ (\lambda x.H\ (x\ x)) \longleftrightarrow_\beta \\
&\quad H\ ((\lambda x.H\ (x\ x))\ (\lambda x.H\ (x\ x))) \longleftrightarrow_\beta \\
&\quad H\ ((\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x)))\ H) \longleftrightarrow_\beta \\
&\quad H\ (Y\ H)
\end{aligned}$$

Τελικά, από τη σχέση (12.3), έχουμε τον ορισμό της συνάρτησης $FACT$, ο οποίος είναι μη αναδρομικός. Δηλαδή:

$$\begin{aligned}
FACT\ \ =\ \ & (\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x))) \\
& (\lambda f.(\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (f\ (-n\ 1)))))
\end{aligned} \tag{12.6}$$

Εκτός από τον Y-συνδυαστή, μια άλλη ομάδα συνδυαστών που έχουν ιδιαίτερη σημασία για τον λάμδα λογισμό είναι ο **I-συνδυαστής** (ή **ταυτοτικός συνδυαστής**), ο **K-συνδυαστής** (ή **συνδυαστής ακύρωσης**) και ο **S-συνδυαστής** (ή **συνδυαστής διανομής**). Οι συνδυαστές αυτοί ορίζονται ως λάμδα αφαιρέσεις:

$$\begin{aligned}
I &= (\lambda x.x) \\
K &= (\lambda x.\lambda y.x) \\
S &= (\lambda f.\lambda g.\lambda x.f\ x\ (g\ x))
\end{aligned}$$

Ο ρόλος των I-, K- και S- συνδυαστών στον λάμδα λογισμό είναι πολύ σημαντικός, γιατί αποδεικνύεται ότι οποιαδήποτε λάμδα έκφραση μπορεί να γραφεί με ισοδύναμο τρόπο, με τη βοήθεια των συνδυαστών αυτών, έτσι ώστε να μην περιλαμβάνει λάμδα αφαιρέσεις. Έτσι, αν εντάξουμε τους συνδυαστές αυτούς στις ενσωματωμένες σταθερές του λάμδα λογισμού και ορίσουμε τις κατάλληλες δ-αναγωγές, μπορούμε να εκφράσουμε οποιαδήποτε συνάρτηση χωρίς να χρησιμοποιήσουμε λάμδα αφαιρέσεις. Αυτό είναι πολύ σημαντικό από τη σκοπιά της υλοποίησης των συναρτησιακών γλωσσών, γιατί, με τους I-, K- και S- συνδυαστές, παρέχεται μια γλώσσα χαμηλού επιπέδου, τόσο ισχυρή όσο και ο λάμδα λογισμός, στην οποία μπορεί να μεταγλωττιστεί οποιοδήποτε συναρτησιακό πρόγραμμα. Για να υλοποιήσουμε πλήρως μια συναρτησιακή γλώσσα, πρέπει, εκτός από αυτήν τη μεταγλώττιση, να πραγματοποιήσουμε με κάποιον τρόπο (για παράδειγμα, με **αναγωγή γράφων**, όπως θα δούμε στην Ενότητα 12.4) τις διάφορες περιπτώσεις αναγωγών που απαιτούνται. Ωστόσο,

λόγω της εξαιρετικής απλότητας της γλώσσας χαμηλού επιπέδου στην οποία μεταγλωττίζεται ένα συναρτησιακό πρόγραμμα, αυτές οι αναγωγές είναι πολύ λίγες, με αποτέλεσμα η διαδικασία υλοποίησής τους να είναι σχετικά απλή. Για παράδειγμα, για τους I-, K- και S-συνδυαστές, οι μόνες αναγωγές που απαιτούνται είναι οι εξής:

$$\begin{aligned} I\ e &\longrightarrow_\delta e \\ K\ e\ f &\longrightarrow_\delta e \\ S\ e\ f\ g &\longrightarrow_\delta e\ g\ (f\ g) \end{aligned}$$

Ο I-συνδυαστής ονομάζεται ταυτοτικός, αφού αφήνει αναλλοίωτο το e, ο K-συνδυαστής ονομάζεται συνδυαστής ακύρωσης, αφού «εξαφανίζει» το f, και ο S-συνδυαστής ονομάζεται συνδυαστής διανομής, αφού «προσφέρει» το g τόσο στο e όσο και στο f.

Παράδειγμα 12.7
Όσο κι αν σας φανεί απίστευτο, η λάμδα αφαίρεση:

$$(\lambda x.\lambda y.+\ x\ y)$$

η οποία, όπως καταλαβαίνετε, όταν εφαρμοστεί σε δύο αριθμούς, επιστρέφει το άθροισμά τους, μπορεί να μεταγλωττιστεί σε μια έκφραση που περιλαμβάνει μόνο ενσωματωμένες σταθερές, ως εξής:

$$S\ (S\ (K\ S)\ (S\ (S\ (K\ S)\ (S\ (K\ K)\ (K\ +)))\ (S\ (K\ K)\ I)))\ (K\ I)$$

Μη σας τρομάζει η «φαινομενική» πολυπλοκότητα αυτής της έκφρασης. Αν δοκιμάσετε να την εφαρμόσετε σε δύο αριθμούς, κάνοντας δ-αναγωγές, τη μία μετά την άλλη, θα πάρετε αποτέλεσμα το άθροισμα των δύο αριθμών.

Άσκηση 12.3
Επιλέξτε ως σωστή μία από τις εξής δύο δηλώσεις:

1. *Οι I-, K- και S- συνδυαστές είναι απολύτως απαραίτητοι, για να διατυπώσουμε μια οποιαδήποτε λάμδα έκφραση στον λάμδα λογισμό συναρτήσει αυτών.*
2. *Ο I-συνδυαστής δεν είναι απολύτως απαραίτητος, επειδή μπορεί να γραφεί συναρτήσει των K- και S- συνδυαστών, σύμφωνα με τη σχέση $I = S\ K\ K$.*

Άσκηση 12.4
Με βάση τη σχέση (12.3), υπολογίστε το παραγοντικό του 1, δηλαδή το $FACT\ 1$.

12.3 Σειρές αναγωγής

Στις Ενότητες 12.1 και 12.2 είδαμε διάφορα παραδείγματα αναγωγής λάμδα εκφράσεων. Λέμε ότι μια έκφραση είναι σε **κανονική μορφή**, αν δεν είναι δυνατόν να αναχθεί περισσότερο. Είναι, όμως, πιθανό, έχοντας μια λάμδα έκφραση, αυτή να μπορεί να αναχθεί με περισσότερους του ενός τρόπους, επειδή περιλαμβάνει περισσότερες της μιας αναγώγιμες εκφράσεις. Κάτι τέτοιο συνέβη στο Παράδειγμα 12.4, στο οποίο είδαμε ότι η έκφραση

$(\lambda x.((\lambda y.\lambda x.*\ y\ x)\ x\ 5))\ 7$ μπορούσε να αναχθεί είτε εφαρμόζοντας την εξωτερική λάμδα αφαίρεση στο 7 είτε εφαρμόζοντας την εσωτερική λάμδα αφαίρεση $(\lambda y.\lambda x.*\ y\ x)$ στο x (φυσικά, μετά την απαιτούμενη α-μετατροπή). Το ευτύχημα ήταν ότι, και με τις δύο εναλλακτικές ακολουθίες αναγωγών, καταλήξαμε στην ίδια κανονική μορφή, το 35. Το ερώτημα, συνεπώς, που τίθεται είναι: Υπήρχε περίπτωση να καταλήξουμε σε δύο διαφορετικές κανονικές μορφές; Στο συγκεκριμένο ερώτημα θα προσπαθήσουμε να απαντήσουμε στην ενότητα αυτή.

Ας δούμε ένα παράδειγμα, στο οποίο η κατάσταση δεν είναι τόσο ιδανική όπως στο Παράδειγμα 12.4.

Παράδειγμα 12.8
Έστω ότι θέλουμε να αναγάγουμε σε κανονική μορφή τη λάμδα έκφραση:

$$(\lambda x.\lambda y.x)\ 5\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

Έχουμε δύο δυνατές επιλογές. Η πρώτη είναι να κάνουμε μια β-αναγωγή στην εξωτερική λάμδα αφαίρεση, δηλαδή να εφαρμόσουμε τη λάμδα έκφραση $(\lambda x.\lambda y.x)$ στο 5 και να πάρουμε το τελικό αποτέλεσμα, και μάλιστα σε δύο μόνο βήματα:

$$\begin{aligned}&(\lambda x.\lambda y.x)\ 5\ ((\lambda z.z\ z)\ (\lambda z.z\ z)) \longrightarrow_\beta\\ &\quad (\lambda y.5)\ ((\lambda z.z\ z)\ (\lambda z.z\ z)) \longrightarrow_\beta\\ &\quad 5\end{aligned}$$

Η δεύτερη επιλογή είναι να κάνουμε β-αναγωγή στην εσωτερική λάμδα αφαίρεση, δηλαδή να εφαρμόσουμε τη λάμδα έκφραση $(\lambda z.z\ z)$ στη λάμδα έκφραση $(\lambda z.z\ z)$. Αν το κάνουμε αυτό, θα πάρουμε αποτέλεσμα, όπως εύκολα μπορούμε να δούμε, πάλι τη λάμδα έκφραση $(\lambda z.z\ z)\ (\lambda z.z\ z)$. Και αν συνεχίσουμε να επιλέγουμε για αναγωγή την εσωτερική λάμδα έκφραση, δεν πρόκειται ποτέ να καταλήξουμε σε κάποια τελική κανονική μορφή. Δηλαδή, οι διαδοχικές αναγωγές που θα κάναμε θα ήταν:

$$\begin{aligned}&(\lambda x.\lambda y.x)\ 5\ ((\lambda z.z\ z)\ (\lambda z.z\ z)) \longrightarrow_\beta\\ &\quad (\lambda x.\lambda y.x)\ 5\ ((\lambda z.z\ z)\ (\lambda z.z\ z)) \longrightarrow_\beta\\ &\quad (\lambda x.\lambda y.x)\ 5\ ((\lambda z.z\ z)\ (\lambda z.z\ z)) \longrightarrow_\beta\\ &\quad (\lambda x.\lambda y.x)\ 5\ ((\lambda z.z\ z)\ (\lambda z.z\ z)) \longrightarrow_\beta\\ &\quad \ldots\end{aligned}$$

Οπότε, πρέπει κάποιος να μπορεί να απαντήσει στο ερώτημα: Υπήρχε περίπτωση να γνωρίζουμε εκ των προτέρων ποια είναι η «καλή» και ποια η «κακή» αναγωγή; □

Έστω ότι έχουμε μια λάμδα έκφραση, η οποία περιλαμβάνει περισσότερες της μιας αναγώγιμες εκφράσεις. Για λόγους που θα γίνουν σαφείς πολύ σύντομα, μας ενδιαφέρει να αναφερόμαστε στην **πιο αριστερή**, στην **πιο εξωτερική** και στην **πιο εσωτερική** αναγώγιμη έκφραση της συνολικής λάμδα έκφρασης. Συγκεκριμένα:

Σε μια λάμδα έκφραση, πιο αριστερή αναγώγιμη έκφραση είναι αυτή που βρίσκεται αριστερότερα από όλες τις άλλες αναγώγιμες εκφράσεις. Πιο εξωτερική αναγώγιμη έκφραση είναι μια αναγώγιμη έκφραση που δεν περιέχεται σε κάποια άλλη αναγώγιμη έκφραση. Πιο εσωτερική αναγώγιμη έκφραση είναι μια αναγώγιμη έκφραση που δεν περιέχει καμία άλλη αναγώγιμη έκφραση.

Όταν, λοιπόν, πρέπει να απλοποιήσουμε μια λάμδα έκφραση, η οποία περιλαμβάνει περισσότερες της μιας αναγώγιμες εκφράσεις, πρέπει να αποφασίσουμε με ποια σειρά θα κάνουμε τις δυνατές αναγωγές. Στον λάμδα λογισμό, έχουν καθιερωθεί δύο βασικές **σειρές αναγωγής**, η **εφαρμοστική σειρά αναγωγής** και η **κανονική σειρά αναγωγής**, οι οποίες ορίζονται ως εξής:

Εφαρμοστική σειρά αναγωγής

Να αναχθεί πρώτα η πιο αριστερή, πιο εσωτερική αναγώγιμη έκφραση.

Κανονική σειρά αναγωγής

Να αναχθεί πρώτα η πιο αριστερή, πιο εξωτερική αναγώγιμη έκφραση.

Στο Παράδειγμα 12.8, η πρώτη επιλογή που ακολουθήσαμε, και στην οποία πήραμε αποτέλεσμα το 5, ήταν η κανονική σειρά αναγωγής της λάμδα έκφρασης. Η δεύτερη επιλογή, αυτή που δεν ήταν δυνατόν να τερματίσει δίνοντας τελικό αποτέλεσμα, ήταν η εφαρμοστική σειρά αναγωγής.

Στην ενότητα αυτή έχουμε θέσει, αν θυμάστε, δύο ερωτήματα: α) Υπάρχει περίπτωση, αν εφαρμόσουμε διαφορετικές σειρές αναγωγής σε μια λάμδα έκφραση, να πάρουμε διαφορετικά τελικά αποτελέσματα; και β) Μπορούμε να ξέρουμε εκ των προτέρων αν κάποια σειρά αναγωγής είναι καταδικασμένη να μην τερματίσει; Τις απαντήσεις σε αυτά τα ερωτήματα μας τις δίνουν δύο ισχυρότατα θεωρήματα στον λάμδα λογισμό, το πρώτο θεώρημα και το δεύτερο θεώρημα των Church-Rosser. Ας δώσουμε τις διατυπώσεις αυτών των θεωρημάτων και ας τα συζητήσουμε.[5]

Πρώτο θεώρημα Church-Rosser

Αν για δύο λάμδα εκφράσεις E_1 και E_2 ισχύει $E_1 \longleftrightarrow E_2$, τότε υπάρχει λάμδα έκφραση E τέτοια ώστε $E_1 \longrightarrow E$ και $E_2 \longrightarrow E$.

Άμεση συνέπεια του πρώτου θεωρήματος των Church-Rosser είναι ότι δεν μπορεί μια λάμδα έκφραση να αναχθεί σε δύο διαφορετικές κανονικές μορφές (εκτός αν η μία προκύπτει από την άλλη με α-μετατροπή, δηλαδή με μετονομασία μεταβλητών). Αυτό απαντά στο πρώτο από τα ερωτήματα που θέσαμε προηγουμένως. Δηλαδή, έχουμε πλέον τη βεβαιότητα ότι δεν υπάρχει περίπτωση να πάρουμε διαφορετικά τελικά αποτελέσματα με διαφορετικές

[5]Είναι απαραίτητο εδώ να δώσουμε κάποιες διευκρινίσεις για τους συμβολισμούς που θα χρησιμοποιήσουμε στα θεωρήματα των Church-Rosser. Στον λάμδα λογισμό, όταν γράφουμε $E_1 \longrightarrow E_2$, για δύο λάμδα εκφράσεις, εννοούμε ότι έπειτα από κάποια ακολουθία μηδέν ή περισσότερων αναγωγών και μετατροπών, η έκφραση E_1 μετασχηματίζεται στην έκφραση E_2. Φυσικά, ακολουθία μηδενικού μήκους αντιστοιχεί στην περίπτωση $E_1 = E_2$. Επίσης, χρησιμοποιούμε και τον συμβολισμό $E_1 \longleftrightarrow E_2$, για να καλύψουμε την περίπτωση που η λάμδα έκφραση E_1 μετασχηματίζεται έπειτα από κάποια ακολουθία μηδέν ή περισσότερων αναγωγών, αφαιρέσεων και μετατροπών στη λάμδα έκφραση E_2, και αντίστροφα.

σειρές αναγωγής σε μια λάμδα έκφραση. Όσον αφορά το δεύτερο ερώτημα, δηλαδή αν μπορούμε να αποφύγουμε να εγκλωβιστούμε σε μια ατέρμονη ακολουθία από αναγωγές, την απάντηση μας τη δίνει το δεύτερο θεώρημα των Church-Rosser.

Δεύτερο θεώρημα Church-Rosser

Αν για δύο λάμδα εκφράσεις E_1 και E_2 ισχύει $E_1 \longrightarrow E_2$ και η E_2 είναι σε κανονική μορφή, τότε η E_2 προκύπτει από την E_1 με την κανονική σειρά αναγωγής.

Συνεπώς, το δεύτερο θεώρημα των Church-Rosser εγγυάται ότι η κανονική σειρά αναγωγής είναι μια ασφαλής σειρά, από την άποψη ότι, αν μια λάμδα έκφραση μπορεί να αναχθεί σε κανονική μορφή, η κανονική σειρά αναγωγής θα καταλήξει σε αυτήν την κανονική μορφή. Δηλαδή, μια λάμδα έκφραση ενδέχεται να μην μπορεί να αναχθεί σε κανονική μορφή; Φυσικά! Θυμηθείτε τη λάμδα έκφραση $(\lambda z.z\ z)\ (\lambda z.z\ z)$ στο Παράδειγμα 12.8. Είναι αναγώγιμη έκφραση, μπορούμε να κάνουμε μια β-αναγωγή, αλλά παίρνουμε αποτέλεσμα τον ίδιο της τον εαυτό. Όσο και να συνεχίσουμε τις αναγωγές, φυσικά, δεν πρόκειται να καταλήξουμε πουθενά.

Η κανονική σειρά αναγωγής στον λάμδα λογισμό αντιστοιχεί σε αυτό που χαρακτηρίζουμε στις συναρτησιακές γλώσσες προγραμματισμού **οκνηρό υπολογισμό**. Ο οκνηρός υπολογισμός προτείνει, ουσιαστικά, να υπολογίζεται μια έκφραση μόνο όταν χρειάζεται. Η εφαρμοστική σειρά αναγωγής αντιστοιχεί στο μοντέλο του **ενθουσιώδους υπολογισμού**, στο οποίο γίνονται αναγωγές εκφράσεων πριν η κανονική μορφή τους απαιτηθεί, αν απαιτηθεί ποτέ, από άλλες εκφράσεις. Μερικές συναρτησιακές γλώσσες, όπως η Standard ML και η Hope, εφαρμόζουν τον ενθουσιώδη υπολογισμό, ενώ άλλες, όπως η SASL, η Miranda και η Haskell, ακολουθούν το μοντέλο του οκνηρού υπολογισμού. Ο ενθουσιώδης υπολογισμός οδηγεί, σε σχέση με τον οκνηρό υπολογισμό, σε πιο αποδοτικές υλοποιήσεις των συναρτησιακών γλωσσών που βασίζονται επάνω σε αυτόν. Όμως, ο οκνηρός υπολογισμός υπερτερεί του ενθουσιώδους σε κάποια σημεία, όπως στη δυνατότητα διαχείρισης μη πεπερασμένων δομών δεδομένων.

Άσκηση 12.5
Έστω η λάμδα έκφραση $(\lambda x. * x\ x)\ (-4\ 1)$. Μια ακολουθία από αναγωγές αυτής της έκφρασης είναι η εξής:

$$
\begin{aligned}
&(\lambda x. * x\ x)\ (-4\ 1) \longrightarrow_\beta \\
&\quad *(-4\ 1)\ (-4\ 1) \longrightarrow_\delta \\
&\quad *3\ (-4\ 1) \longrightarrow_\delta \\
&\quad *3\ 3 \longrightarrow_\delta \\
&\quad 9
\end{aligned}
$$

Μια εναλλακτική ακολουθία είναι όμως και η εξής:

$$
\begin{aligned}
&(\lambda x. * x\ x)\ (-4\ 1) \longrightarrow_\beta \\
&\quad *(-4\ 1)\ (-4\ 1) \longrightarrow_\delta \\
&\quad *(-4\ 1)\ 3 \longrightarrow_\delta \\
&\quad *3\ 3 \longrightarrow_\delta \\
&\quad 9
\end{aligned}
$$

Μια άλλη ακολουθία είναι επίσης η εξής:

$$
\begin{aligned}
&(\lambda x.\ast\, x\ x)\ (-4\ 1) \longrightarrow_{\delta}\\
&\quad (\lambda x.\ast\, x\ x)\ 3 \longrightarrow_{\beta}\\
&\quad \ast 3\ 3 \longrightarrow_{\delta}\\
&\quad 9
\end{aligned}
$$

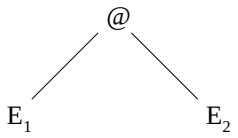
Αντιστοιχεί κάποια από αυτές τις ακολουθίες στην εφαρμοστική σειρά αναγωγής και κάποια στην κανονική σειρά αναγωγής της αρχικής έκφρασης, και ποια;

Άσκηση 12.6
Στην Άσκηση 12.1, σας ζητήθηκε να συμπληρώσετε μια ημιτελή ακολουθία από αναγωγές και μετατροπές. Είναι συμβατή η απάντηση που δώσατε με κάποια από τις καθιερωμένες σειρές αναγωγής στον λάμδα λογισμό; Αν όχι, διατυπώστε πάλι την ακολουθία και με τις δύο αυτές σειρές.

12.4 Αναγωγή γράφων

Η **αναγωγή γράφων** είναι μια πολύ βασική, η βασικότερη ίσως, τεχνική υλοποίησης συναρτησιακών γλωσσών που ακολουθούν το μοντέλο του οκνηρού υπολογισμού. Αυτό σημαίνει ότι, για την αναγωγή εκφράσεων, οι γλώσσες αυτές εφαρμόζουν την κανονική σειρά αναγωγής. Η ιδέα της αναγωγής γράφων έγκειται στην αναπαράσταση μιας λάμδα έκφρασης με τη μορφή ενός γράφου και στην εφαρμογή ενός συνόλου από μετασχηματισμούς (αναγωγές) στον γράφο αυτό, με σκοπό την απλοποίησή του, που αντιστοιχεί στην απλοποίηση της λάμδα έκφρασης.

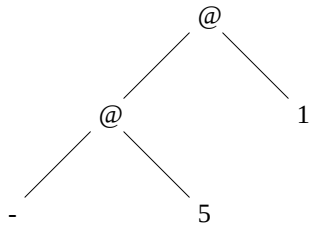
Το συχνότερα εμφανιζόμενο συστατικό στους γράφους που αναπαριστούν λάμδα εκφράσεις είναι ο **κόμβος εφαρμογής** ή **@-κόμβος**. Μια λάμδα έκφραση $E_1\ E_2$, όπου E_1 και E_2 είναι επίσης λάμδα εκφράσεις, παριστάνεται από τον γράφο του Σχήματος 12.1. Η ρίζα του γράφου[6] είναι κόμβος εφαρμογής.



Σχήμα 12.1: Γράφος με κόμβο εφαρμογής.

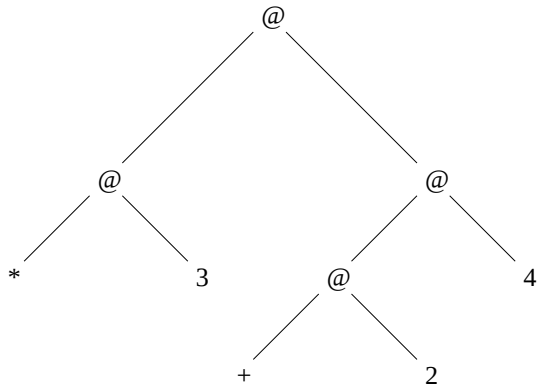
Παράδειγμα 12.9
Έστω ότι θέλουμε να παραστήσουμε τη λάμδα έκφραση $-5\ 1$ σαν έναν γράφο. Θυμηθείτε ότι, λόγω της αριστερής προσεταιριστικότητας της εφαρμογής συναρτήσεων, αυτή η έκφραση είναι ουσιαστικά η $(-5)\ 1$. Οπότε, ο γράφος που αντιστοιχεί στη συγκεκριμένη έκφραση είναι αυτός που φαίνεται στο Σχήμα 12.2.

[6] Αναφερόμαστε σε γράφους, παρότι ίσως να νομίζετε ότι θα μας αρκούσε να είχαμε απλώς δέντρα. Θα δούμε στη συνέχεια γιατί πραγματικά χρειαζόμαστε γράφους.



Σχήμα 12.2: Γράφος για την έκφραση $-5\ 1$.

Παράδειγμα 12.10
Η πιο σύνθετη λάμδα έκφραση $*3\ (+2\ 4)$ μπορεί να αναπαρασταθεί με τον γράφο του Σχήματος 12.3. □



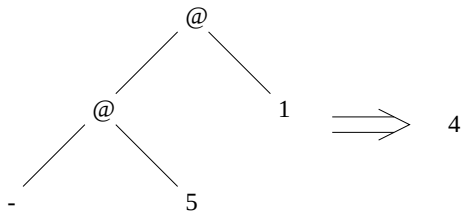
Σχήμα 12.3: Γράφος για την έκφραση $*3\ (+2\ 4)$.

Η διαδικασία της αναγωγής ενός γράφου που περιγράφει την εφαρμογή μιας λάμδα έκφρασης επάνω σε μια λάμδα έκφραση, δηλαδή ενός γράφου που έχει ρίζα έναν @-κόμβο, αρχίζει με τη διάσχιση του γράφου κατά βάθος και προς τα αριστερά, περνώντας επάνω από όποιους @-κόμβους συναντά, μέχρι να βρεθεί κάποιος που δεν είναι @-κόμβος. Υπάρχει περίπτωση αυτός ο κόμβος να είναι το σύμβολο μιας ενσωματωμένης σταθεράς (συνάρτησης). Τότε, εφόσον υπάρχει ο απαιτούμενος αριθμός εκφράσεων για να εφαρμοστεί επάνω τους η ενσωματωμένη συνάρτηση που βρέθηκε, μπορεί να γίνει η κατάλληλη αναγωγή και να απλοποιηθεί ο γράφος. Συγκεκριμένα, αν η συνάρτηση που βρέθηκε χρειάζεται N εκφράσεις ως ορίσματα, για να είναι δυνατόν να αναχθεί ο γράφος, θα πρέπει, πριν βρούμε τη συνάρτηση αυτή στο πιο αριστερό φύλλο του γράφου, να έχουμε βρει τουλάχιστον N @-κόμβους. Τα δεξιά παιδιά των N τελευταίων από αυτούς παριστάνουν τα απαιτούμενα N ορίσματα της συνάρτησης. Αν αυτά τα ορίσματα είναι ανηγμένα, όπως απαιτεί η συνάρτηση, για να γίνει η κατάλληλη δ-αναγωγή, έχει καλώς. Αν όχι, γίνεται η ίδια διαδικασία (αναδρομικά), για να αναχθεί ο γράφος καθενός από αυτά. Αφού απλοποιηθούν τα ορίσματα, γίνεται η δ-αναγωγή και αντικαθίσταται ο πρώτος από τους N @-κόμβους, μαζί με όλο τον γράφο κάτω από αυτόν, με το αποτέλεσμα της αναγωγής.

Η διαδικασία αναγωγής του γράφου, όπως την περιγράψαμε, αντιστοιχεί στην κανονική σειρά αναγωγής. Ας δούμε, στη συνέχεια, τι αποτελέσματα δίνει σε συγκεκριμένα παραδείγματα.

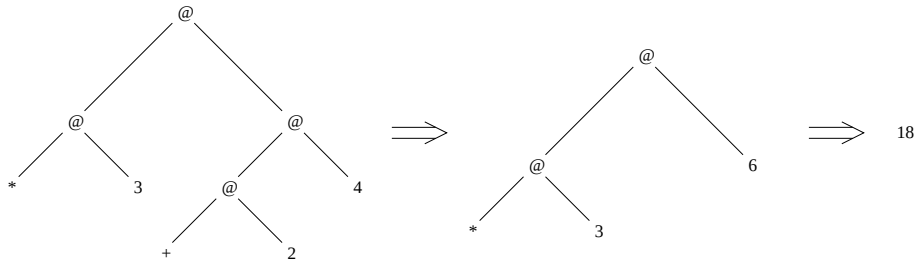
Παράδειγμα 12.11

Αν εφαρμόσουμε τη διαδικασία αναγωγής στον γράφο του Σχήματος 12.2, διατρέχοντας το αριστερό μονοπάτι, θα καταλήξουμε στον κόμβο «$-$», έχοντας περάσει επάνω από δύο @-κόμβους, όσα ακριβώς είναι και τα ορίσματα που χρειάζεται η ενσωματωμένη συνάρτηση της αφαίρεσης για να εφαρμοστεί μια δ-αναγωγή. Τα ορίσματα αυτά είναι ήδη ανηγμένα, αφού είναι οι αριθμοί «5» και «1». Οπότε, ο γράφος ανάγεται και αντικαθίσταται ο @-κόμβος στη ρίζα, μαζί με όλο το υποδέντρο του, από το αποτέλεσμα της δ-αναγωγής, δηλαδή τον αριθμό «4», όπως φαίνεται στο Σχήμα 12.4.



Σχήμα 12.4: Αναγωγή γράφου για την έκφραση $-5\ 1$.

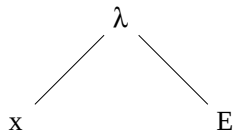
Παράδειγμα 12.12
Ο γράφος του Σχήματος 12.3 είναι λίγο πιο πολύπλοκος. Εδώ, διατρέχοντας το αριστερό μονοπάτι, θα συναντήσουμε την ενσωματωμένη συνάρτηση «$*$», για τον πολλαπλασιασμό, έπειτα από δύο @-κόμβους. Όμως, εδώ έχουμε το πρόβλημα ότι, ενώ το πρώτο όρισμα του «$*$» είναι ανηγμένο, ο αριθμός «3», το δεύτερο όρισμα είναι ένας γράφος με @-κόμβο στη ρίζα του. Για να αναχθεί λοιπόν ο αρχικός γράφος, πρέπει πρώτα να αναχθεί ο γράφος του ορίσματος και να αντικατασταθεί αυτός ο γράφος με το αποτέλεσμα. Έτσι, διατρέχουμε τον νέο γράφο, μέχρι που βρίσκουμε το «$+$», τα ορίσματά του είναι έτοιμα, το «2» και το «4», γίνεται η δ-αναγωγή και αντικαθίσταται ο γράφος με το αποτέλεσμα, το «6». Τώρα, μπορεί να γίνει και η δ-αναγωγή του αρχικού γράφου, δίνοντας αποτέλεσμα το «18». Αυτά φαίνονται στο Σχήμα 12.5. □



Σχήμα 12.5: Αναγωγή γράφου για την έκφραση $*3\ (+2\ 4)$.

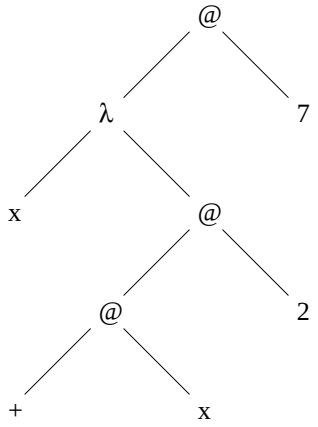
Εκτός από τους @-κόμβους, υπάρχει και μια δεύτερη κατηγορία κόμβων, οι οποίοι συμμετέχουν στην κατασκευή γράφων για την αναπαράσταση λάμδα εκφράσεων. Πρόκειται για τους **λ-κόμβους**. Ένας λ-κόμβος χρησιμοποιείται για την αναπαράσταση μιας λάμδα αφαίρεσης. Η λάμδα αφαίρεση $(\lambda x.E)$ παριστάνεται από τον γράφο του Σχήματος 12.6.

Παράδειγμα 12.13
Η λάμδα έκφραση $(\lambda x.\ +\ x\ 2)\ 7$, που περιγράφει την εφαρμογή μιας λάμδα αφαίρεσης



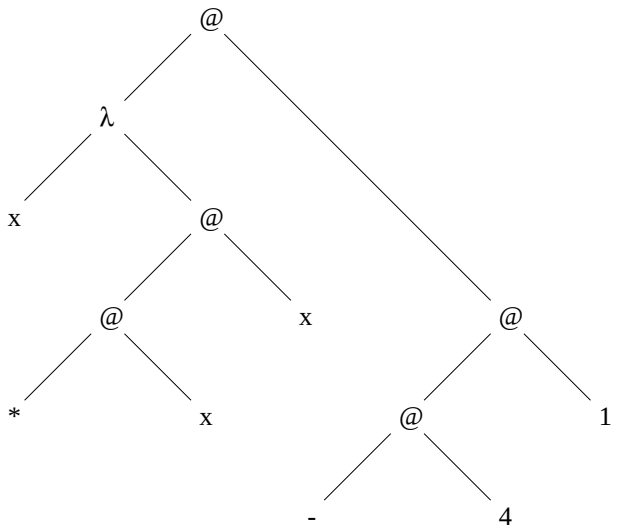
Σχήμα 12.6: Γράφος με λ-κόμβο.

επάνω σε μια άλλη λάμδα έκφραση, μπορεί να παρασταθεί από έναν γράφο, όπως φαίνεται στο Σχήμα 12.7.



Σχήμα 12.7: Γράφος για την έκφραση $(\lambda x.\ +\ x\ 2)\ 7$.

Παράδειγμα 12.14
Ο γράφος για την έκφραση $(\lambda x.\ *\ x\ x)\ (-4\ 1)$ φαίνεται στο Σχήμα 12.8. □

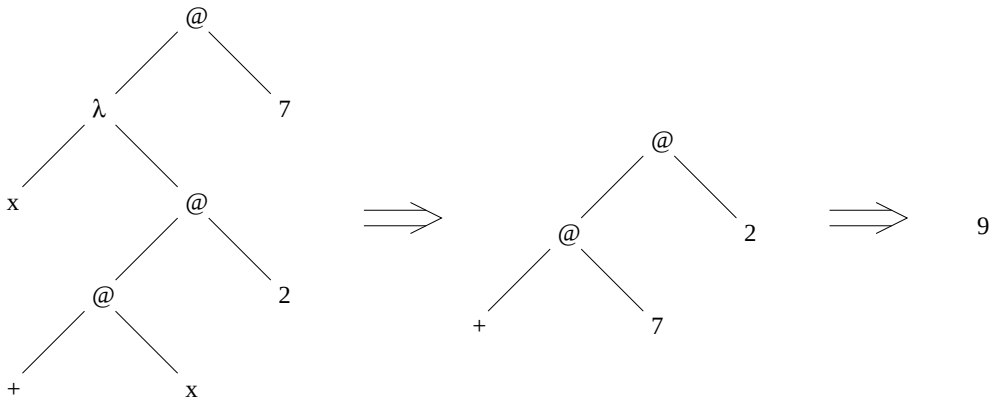


Σχήμα 12.8: Γράφος για την έκφραση $(\lambda x.\ *\ x\ x)\ (-4\ 1)$.

Όταν προσπαθούμε να κάνουμε αναγωγή σε έναν γράφο, διατρέχοντάς τον κατά βάθος και προς τα αριστερά, έχοντας ξεκινήσει από τη ρίζα του και περνώντας επάνω από @-κόμβους, υπάρχει ενδεχόμενο, αντί να συναντήσουμε κόμβο που αντιστοιχεί σε ενσωματωμένη σταθερά (συνάρτηση), να βρούμε έναν λ-κόμβο. Πάλι μπορούμε να κάνουμε

αναγωγή στον γράφο, με την εξής διαδικασία όμως τώρα: Το δεξί παιδί του @-κόμβου που συναντήσαμε πριν βρούμε τον λ-κόμβο, κατεβαίνοντας το αριστερό μονοπάτι, παριστάνει την έκφραση επάνω στην οποία πρέπει να εφαρμοστεί η λάμδα αφαίρεση που αντιστοιχεί στον λ-κόμβο. Η αναγωγή (πρόκειται περί β-αναγωγής) αφορά την αντικατάσταση του @-κόμβου, που είναι γονέας του λ-κόμβου, μαζί με όλο τον γράφο κάτω από αυτόν, με το δεξί παιδί του λ-κόμβου (σώμα της λάμδα αφαίρεσης), έχοντας όμως αντικαταστήσει τις εμφανίσεις της δεσμευμένης μεταβλητής της λάμδα αφαίρεσης (αριστερό παιδί του λ-κόμβου) στο σώμα με το δεξί παιδί του @-κόμβου. Ίσως αυτά να σας φαίνονται λίγο μπερδεμένα, αλλά θα ξεκαθαρίσουν στα παραδείγματα που ακολουθούν.

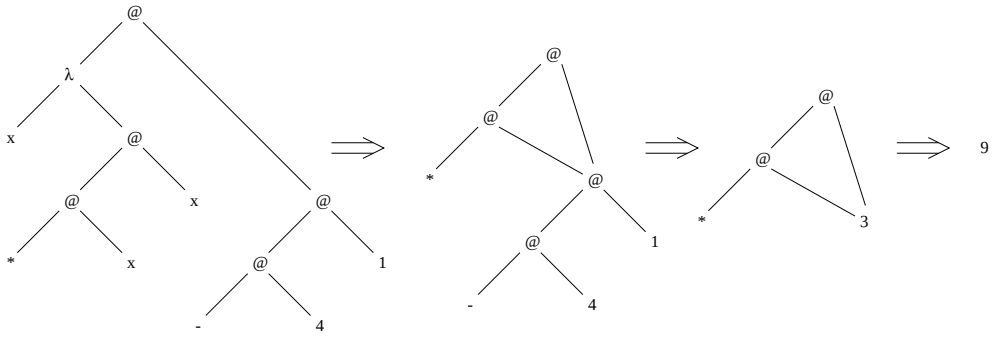
Παράδειγμα 12.15
Ο γράφος του Σχήματος 12.7 μπορεί να αναχθεί ως εξής: Κατεβαίνοντας το αριστερό μονοπάτι, μετά τον @-κόμβο στη ρίζα, συναντάμε έναν λ-κόμβο. Γίνεται μια πρώτη αναγωγή, αντικαθιστώντας τον αρχικό γράφο με το δεξί παιδί του λ-κόμβου, έχοντας αντικαταστήσει στη θέση του «x» το «7». Έτσι, παίρνουμε έναν γράφο που ανάγεται πολύ απλά, κάνοντας μια πρόσθεση που δίνει αποτέλεσμα το «9». Η διαδικασία της αναγωγής του αρχικού γράφου φαίνεται στο Σχήμα 12.9.



Σχήμα 12.9: Αναγωγή γράφου για την έκφραση $(\lambda x.\ +\ x\ 2)\ 7$.

Παράδειγμα 12.16
Ας δούμε πώς μπορεί να γίνει η αναγωγή του γράφου στο Σχήμα 12.8. Εδώ υπάρχει μια ιδιαιτερότητα, που πρέπει να της δώσετε μεγάλη προσοχή. Συναντώντας τον λ-κόμβο στο αριστερό μονοπάτι, για την απαιτούμενη αναγωγή, θα πρέπει οι εμφανίσεις της μεταβλητής «x» στο δεξί παιδί του λ-κόμβου να αντικατασταθούν με τον γράφο που είναι δεξί παιδί του @-κόμβου στη ρίζα. Ο γράφος όμως αυτός δεν είναι ανηγμένος και δεν πρέπει να αναχθεί αυτήν τη στιγμή, αφού κάτι τέτοιο δεν επιτρέπεται από την κανονική σειρά αναγωγής. Αν όμως τον αντικαθιστούσαμε στη θέση της μεταβλητής «x», τότε θα είχαμε δύο αντίγραφα αυτού του γράφου στον νέο γράφο που θα προέκυπτε, με αποτέλεσμα στις αναγωγές που θα ακολουθούσαν να έπρεπε να γίνει δύο φορές η αναγωγή του ίδιου γράφου. Αυτό όμως δεν είναι ιδιαίτερα αποδοτικό, γιατί μπορεί στη συγκεκριμένη περίπτωση ο γράφος για τον οποίο συζητάμε να παριστάνει μια απλή αφαίρεση (την έκφραση $(-4\ 1)$), αλλά, εν γένει, μπορεί να είναι πολύ πολύπλοκος και, συνεπώς, να κοστίζει πολύ η αναγωγή του. Η τεχνική λοιπόν εδώ επιβάλλει να μην αντικαταστήσουμε τις εμφανίσεις της μεταβλητής «x» με δύο αντίγραφα του γράφου, αλλά με δείκτες στο ίδιο αντίγραφο. Έτσι, στη συνέχεια ο

γράφος αυτός θα αναχθεί μία μόνο φορά. Όλη η ακολουθία των αναγωγών φαίνεται στο Σχήμα 12.10. □



Σχήμα 12.10: Αναγωγή γράφου για την έκφραση $(\lambda x. * \; x \; x) \; (-4 \; 1)$.

Στο Σχήμα 12.10 φαίνεται καθαρά γιατί μιλάμε για αναγωγή γράφων και όχι για αναγωγή δέντρων.

Όπως αναφέραμε και στην αρχή αυτής της ενότητας, η αναγωγή γράφων είναι μια βασική τεχνική υλοποίησης οκνηρών συναρτησιακών γλωσσών. Ακόμα και στην περίπτωση που η υλοποίηση συγκεκριμένης συναρτησιακής γλώσσας είναι βασισμένη σε συνδυαστές (I-, K- και S-), η αναγωγή γράφων είναι πάλι εφαρμόσιμη. Και εδώ δουλεύουμε όπως στα παραδείγματα αυτής της ενότητας, απλώς, αφού δεν έχουμε λάμδα αφαιρέσεις, δεν υπάρχουν λ-κόμβοι στο γράφο. Χρειαζόμαστε όμως τους κανόνες αναγωγής για τους συνδυαστές που χρησιμοποιούμε.

Έχουν γίνει πολλές υλοποιήσεις συναρτησιακών γλωσσών που είναι βασισμένες σε αναγωγή γράφων. Μια υλοποίηση από αυτές, εξαιρετικά διαδεδομένη, είναι η **μηχανή-G**. Τέλος, πρέπει να επισημανθεί ότι, λόγω της έμφυτης παραλληλίας που υπάρχει στην αναγωγή γράφων, έχει γίνει πολλή έρευνα στην περιοχή της **παράλληλης αναγωγής γράφων**, με σημαντικά αποτελέσματα.

Άσκηση 12.7

Έστω η λάμδα έκφραση $(\lambda x. \lambda y. * \; x \; y) \; 7 \; 5$. Ποιος από τους γράφους του Σχήματος 12.11 την παριστάνει με απόλυτη ακρίβεια και σε ποιες εκφράσεις αντιστοιχούν οι άλλοι;

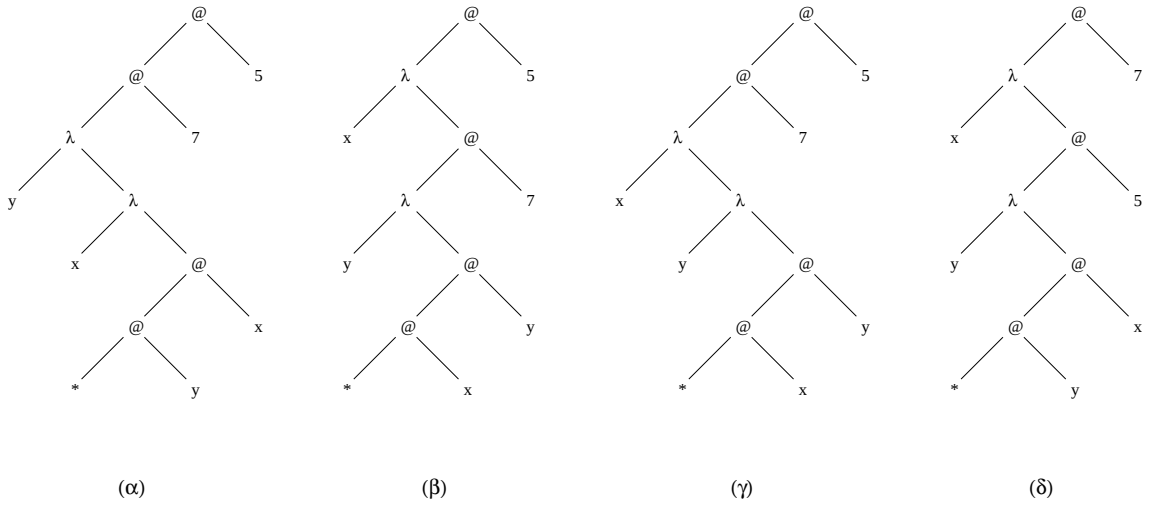
Άσκηση 12.8

Στην Άσκηση 12.6 σας ζητήθηκε να δώσετε την εφαρμοστική και την κανονική σειρά αναγωγής για τη λάμδα έκφραση $(\lambda f. \lambda x. f \; (f \; x)) \; (\lambda x. * 3 \; x) \; 5$, η οποία είχε αναχθεί με έναν κάπως αυθαίρετο τρόπο στην Άσκηση 12.1. Αναπαραστήστε την έκφραση αυτή με έναν γράφο και κάντε τις κατάλληλες αναγωγές σε αυτόν, μέχρι να φτάσετε στην κανονική μορφή του. Μπορείτε να συμβουλευτείτε, φυσικά, την κανονική σειρά αναγωγής της έκφρασης, όπως δίνεται στην απάντηση της Άσκησης 12.6.

Απαντήσεις ασκήσεων

Απάντηση άσκησης 12.1

Η ακολουθία αναγωγών και μετατροπών που δόθηκε πρέπει να συμπληρωθεί ως εξής:



Σχήμα 12.11: Υποψήφιοι γράφοι για την έκφραση $(\lambda x.\lambda y. * \; x \; y) \; 7 \; 5$.

$$
\begin{aligned}
&(\lambda f.\lambda x.f \; (f \; x)) \; (\lambda x. * \; 3 \; x) \; 5 \longleftrightarrow_{\alpha} \\
&\quad (\lambda f.\lambda x'.f \; (f \; x')) \; (\lambda x. * \; 3 \; x) \; 5 \longrightarrow_{\beta} \\
&\quad (\lambda x'.(\lambda x. * \; 3 \; x) \; ((\lambda x. * \; 3 \; x) \; x')) \; 5 \longrightarrow_{\beta} \\
&\quad (\lambda x. * \; 3 \; x) \; ((\lambda x. * \; 3 \; x) \; 5) \longrightarrow_{\beta} \\
&\quad (\lambda x. * \; 3 \; x) \; (*3 \; 5) \longrightarrow_{\delta} \\
&\quad (\lambda x. * \; 3 \; x) \; 15 \longrightarrow_{\beta} \\
&\quad *3 \; 15 \longrightarrow_{\delta} \\
&\quad 45
\end{aligned}
$$

Η λάμδα αφαίρεση $(\lambda f.\lambda x.f \; (f \; x))$ παριστάνει ουσιαστικά τη συνάρτηση `twice`, που ορίσαμε για τη Haskell στο Παράδειγμα 11.6. Στη συγκεκριμένη λάμδα έκφραση, από την οποία αρχίζει η ακολουθία που παρουσιάσαμε, η λάμδα αφαίρεση για τη συνάρτηση `twice` εφαρμόζεται επάνω στη λάμδα αφαίρεση $(\lambda x. * \; 3 \; x)$, η οποία αντιστοιχεί στη συνάρτηση που επιστρέφει για κάθε x το $3 * x$. Οπότε, το αποτέλεσμα είναι η συνάρτηση που επιστρέφει για κάθε x το $3 * (3 * x) = 9 * x$. Συνεπώς, με εφαρμογή επάνω στο 5, σωστά προκύπτει το αποτέλεσμα 45.

Απάντηση άσκησης 12.2
Το ζητούμενο μπορεί να αποδειχθεί ως εξής:

$$
\begin{aligned}
&HEAD \; (CONS \; p \; q) = \\
&\quad (\lambda c.c \; (\lambda a.\lambda b.a)) \; ((\lambda h.\lambda t.\lambda s.s \; h \; t) \; p \; q) \longrightarrow_{\beta} \\
&\quad (\lambda h.\lambda t.\lambda s.s \; h \; t) \; p \; q \; (\lambda a.\lambda b.a) \longrightarrow_{\beta} \\
&\quad (\lambda t.\lambda s.s \; p \; t) \; q \; (\lambda a.\lambda b.a) \longrightarrow_{\beta} \\
&\quad (\lambda s.s \; p \; q) \; (\lambda a.\lambda b.a) \longrightarrow_{\beta} \\
&\quad (\lambda a.\lambda b.a) \; p \; q \longrightarrow_{\beta} \\
&\quad (\lambda b.p) \; q \longrightarrow_{\beta}
\end{aligned}
$$

p

Απάντηση άσκησης 12.3
Η σωστή δήλωση είναι η υπ' αριθμόν 2. Παρότι στην αναφορά μας στους I-, K- και S- συνδυαστές, στην ενότητα αυτή, δεν δηλώσαμε ρητά ότι ο I-συνδυαστής μπορεί να εκφραστεί συναρτήσει των άλλων δύο, είναι εύκολο να διαπιστώσουμε, πράγματι, ότι $I = S\ K\ K$. Ας δοκιμάσουμε να εφαρμόσουμε το $S\ K\ K$ σε κάποιο x. Έχουμε, λοιπόν, σύμφωνα και με τις δ-αναγωγές που ισχύουν για τους K- και S- συνδυαστές:

$$\begin{aligned}&(S\ K\ K)\ x \longrightarrow_\delta\\ &\quad K\ x\ (K\ x) \longrightarrow_\delta\\ &\quad x\end{aligned}$$

Αλλά, ως γνωστόν, έχουμε και:

$$I\ x \longrightarrow_\delta x$$

Άρα, $I = S\ K\ K$.

Απάντηση άσκησης 12.4
Για να υπολογίσουμε το $FACT\ 1$, εργαζόμαστε ως εξής:

$$\begin{aligned}&FACT\ 1 =\\ &\quad Y\ H\ 1 =\\ &\quad H\ (Y\ H)\ 1 =\\ &\quad (\lambda f.(\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (f\ (-n\ 1))))))\ (Y\ H)\ 1 \longrightarrow_\beta\\ &\quad (\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (Y\ H\ (-n\ 1))))\ 1 \longrightarrow_\beta\\ &\quad COND\ (=\ 1\ 0)\ 1\ (*1\ (Y\ H\ (-1\ 1))) \longrightarrow_\delta\\ &\quad *1\ (Y\ H\ (-1\ 1)) \longrightarrow_\delta\\ &\quad *1\ (Y\ H\ 0) =\\ &\quad *1\ (H\ (Y\ H)\ 0) =\\ &\quad *1\ ((\lambda f.(\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (f\ (-n\ 1)))))\ (Y\ H)\ 0) \longrightarrow_\beta\\ &\quad *1\ ((\lambda n.COND\ (=\ n\ 0)\ 1\ (*n\ (Y\ H\ (-n\ 1))))\ 0) \longrightarrow_\beta\\ &\quad *1\ (COND\ (=\ 0\ 0)\ 1\ (*0\ (Y\ H\ (-0\ 1)))) \longrightarrow_\delta\\ &\quad *1\ 1 \longrightarrow_\delta\\ &\quad 1\end{aligned}$$

Απάντηση άσκησης 12.5
Η τρίτη ακολουθία υπακούει στην εφαρμοστική σειρά αναγωγής, αφού στο πρώτο βήμα, που είχαμε δύο επιλογές, αρχίσαμε με την αναγωγή της πιο αριστερής, πιο εσωτερικής αναγώγιμης έκφρασης $(-4\ 1)$. Η πρώτη ακολουθία προκύπτει από την κανονική σειρά αναγωγής, αφού στο πρώτο βήμα κάναμε αναγωγή στην πιο αριστερή, πιο εξωτερική αναγώγιμη

έκφραση, που ήταν η ίδια η αρχική έκφραση, και στο δεύτερο βήμα, στο οποίο είχαμε να επιλέξουμε ανάμεσα στις δύο αναγώγιμες εκφράσεις $(-4\ 1)$, επιλέξαμε την πιο αριστερή από αυτές. Στο σημείο αυτό, στη δεύτερη ακολουθία, επιλέξαμε την πιο δεξιά αναγώγιμη έκφραση για αναγωγή, όμως αυτό δεν αντιστοιχεί σε καμία από τις καθιερωμένες σειρές αναγωγής.

Απάντηση άσκησης 12.6
Στην απάντηση της Άσκησης 12.1 βρίσκεται η εξής ακολουθία αναγωγών:

$$
\begin{aligned}
&(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.*3\ x)\ 5 \longleftrightarrow_\alpha \\
&\quad (\lambda f.\lambda x'.f\ (f\ x'))\ (\lambda x.*3\ x)\ 5 \longrightarrow_\beta \\
&\quad (\lambda x'.(\lambda x.*3\ x)\ ((\lambda x.*3\ x)\ x'))\ 5 \longrightarrow_\beta && (12.7)\\
&\quad (\lambda x.*3\ x)\ ((\lambda x.*3\ x)\ 5) \longrightarrow_\beta && (12.8)\\
&\quad (\lambda x.*3\ x)\ (*3\ 5) \longrightarrow_\delta && (12.9)\\
&\quad (\lambda x.*3\ x)\ 15 \longrightarrow_\beta \\
&\quad *3\ 15 \longrightarrow_\delta \\
&\quad 45
\end{aligned}
$$

Αυτή η ακολουθία δεν είναι σύμφωνη ούτε με την εφαρμοστική σειρά αναγωγής, αφού στο βήμα (12.7) κάναμε αναγωγή της πιο εξωτερικής αναγώγιμης έκφρασης, ούτε με την κανονική σειρά αναγωγής, αφού στα βήματα (12.8) και (12.9) κάναμε αναγωγές των πιο εσωτερικών αναγώγιμων εκφράσεων.

Η εφαρμοστική σειρά αναγωγής είναι η εξής:

$$
\begin{aligned}
&(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.*3\ x)\ 5 \longleftrightarrow_\alpha \\
&\quad (\lambda f.\lambda x'.f\ (f\ x'))\ (\lambda x.*3\ x)\ 5 \longrightarrow_\beta \\
&\quad (\lambda x'.(\lambda x.*3\ x)\ ((\lambda x.*3\ x)\ x'))\ 5 \longrightarrow_\beta \\
&\quad (\lambda x'.(\lambda x.*3\ x)\ (*3\ x'))\ 5 \longrightarrow_\beta \\
&\quad (\lambda x'.*3\ (*3\ x'))\ 5 \longrightarrow_\beta \\
&\quad *3\ (*3\ 5) \longrightarrow_\delta \\
&\quad *3\ 15 \longrightarrow_\delta \\
&\quad 45
\end{aligned}
$$

Η κανονική σειρά αναγωγής είναι η εξής:

$$
\begin{aligned}
&(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.*3\ x)\ 5 \longleftrightarrow_\alpha \\
&\quad (\lambda f.\lambda x'.f\ (f\ x'))\ (\lambda x.*3\ x)\ 5 \longrightarrow_\beta \\
&\quad (\lambda x'.(\lambda x.*3\ x)\ ((\lambda x.*3\ x)\ x'))\ 5 \longrightarrow_\beta \\
&\quad (\lambda x.*3\ x)\ ((\lambda x.*3\ x)\ 5) \longrightarrow_\beta \\
&\quad *3\ ((\lambda x.*3\ x)\ 5) \longrightarrow_\beta \\
&\quad *3\ (*3\ 5) \longrightarrow_\delta \\
&\quad *3\ 15 \longrightarrow_\delta \\
&\quad 45
\end{aligned}
$$

Απάντηση άσκησης 12.7
Ο σωστός γράφος, από αυτούς του Σχήματος 12.11, που παριστάνει τη λάμδα έκφραση $(\lambda x.\lambda y. * x\ y)\ 7\ 5$ είναι ο (γ). Εδώ έχουμε διπλή λάμδα αφαίρεση, αφού το σώμα της λάμδα αφαίρεσης με δεσμευμένη μεταβλητή το x είναι μια λάμδα αφαίρεση με δεσμευμένη μεταβλητή το y.

Εδώ χρειάζεται προσοχή στον κίνδυνο να παραπλανηθούμε από την περίπτωση (α), όπου έχουμε «φωλιασμένες» λάμδα αφαιρέσεις, μόνο που το «φώλιασμα» είναι με αντίστροφη σειρά. Επίσης, έχουμε αντίστροφη σειρά στα ορίσματα της ενσωματωμένης συνάρτησης $*$. Δηλαδή, η λάμδα έκφραση για τον γράφο (α) είναι η $(\lambda y.\lambda x. * y\ x)\ 7\ 5$. Οι γράφοι των περιπτώσεων (β) και (δ) δεν είναι αυτό που θέλουμε. Ο γράφος (β) παριστάνει τη λάμδα έκφραση $(\lambda x.((\lambda y. * x\ y)\ 7))\ 5$, ενώ ο γράφος (δ) την $(\lambda x.((\lambda y. * y\ x)\ 5))\ 7$.

Απάντηση άσκησης 12.8
Ο γράφος που παριστάνει την έκφραση $(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x. * 3\ x)\ 5$ και οι διαδοχικές αναγωγές του μέχρι την κανονική μορφή φαίνονται στο Σχήμα 12.12.

Ο αρχικός μας γράφος είναι ο (α). Στην πρώτη αναγωγή, κατεβαίνοντας το αριστερό μονοπάτι, συναντάμε, έχοντας περάσει επάνω από δύο @-κόμβους, έναν λ-κόμβο. Οπότε, ο @-κόμβος αντικαθίσταται πριν από τον λ-κόμβο με το δεξί παιδί του λ-κόμβου (σώμα της λάμδα αφαίρεσης), αντικαθιστώντας, όμως, και τις εμφανίσεις της μεταβλητής f με δείκτες στο δεξί παιδί του @-κόμβου. Έτσι, παίρνουμε τον γράφο (β).

Για την αναγωγή του γράφου (β), διασχίζοντας το αριστερό μονοπάτι, φτάνουμε στον λ-κόμβο και αντικαθιστούμε τον γονέα του @-κόμβο, που είναι η ρίζα του γράφου (β), με το δεξί παιδί του λ-κόμβου, όπου στη θέση της μεταβλητής x βάζουμε το 5. Προσέξτε όμως! Η αντικατάσταση της μεταβλητής x δεν γίνεται στον γράφο που είναι κάτω από τον επόμενο λ-κόμβο, ο οποίος τυχαίνει να έχει δεσμευμένη μεταβλητή με το ίδιο όνομα. Γίνεται μόνο στην ελεύθερη εμφάνιση της x. Αυτό το πρόβλημα το είχαμε αντιμετωπίσει στην κλασική διατύπωση της κανονικής σειράς αναγωγών, στην Άσκηση 12.5, κάνοντας στην αρχή μια α-μετατροπή. Τελικά, έχουμε τον γράφο (γ).

Στο επόμενο βήμα, πάλι βρίσκουμε λ-κόμβο κάτω από τον @-κόμβο, που είναι ρίζα του γράφου. Και εδώ κάνουμε την κατάλληλη αναγωγή. Ένα ιδιαίτερο σημείο εδώ που θέλει προσοχή είναι ότι η μεταβλητή x στον δεξί υπογράφο του λ-κόμβου αντικαθίσταται από έναν γράφο που έχει απόγονο αυτόν τον λ-κόμβο. Τώρα, πήραμε τον γράφο (δ).

Η συνέχεια είναι απλή. Παίρνουμε τον γράφο (ε) με την αναγωγή του γράφου που έχει ρίζα τον λ-κόμβο, τον γράφο (στ) και, τέλος, τον γράφο (ζ), που είναι απλώς το 45.

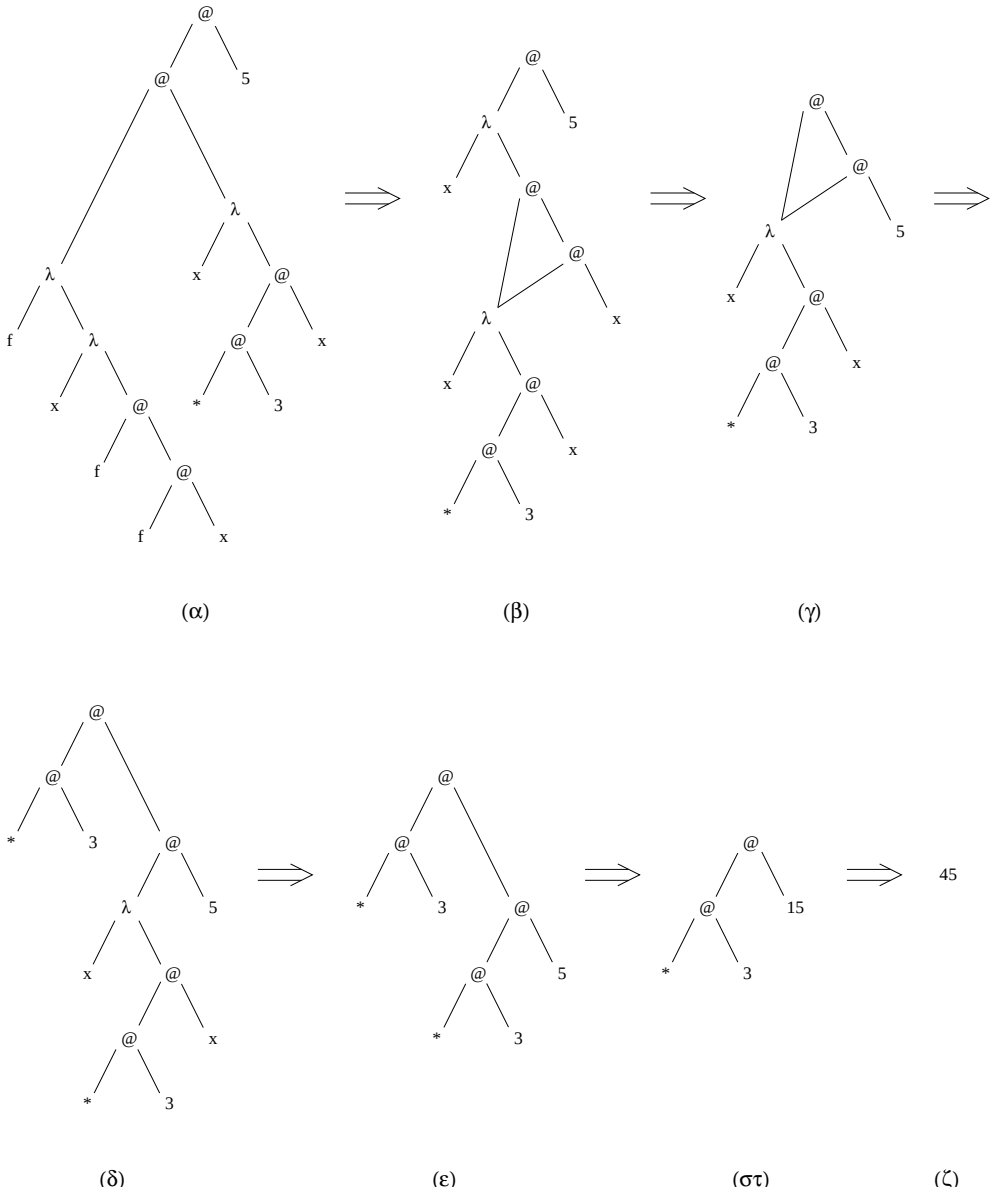
Προβλήματα

Πρόβλημα 12.1
Διατυπώστε τις λάμδα αφαιρέσεις που αντιστοιχούν στις συναρτήσεις `fst` και `snd` της Haskell.

Πρόβλημα 12.2
Ποιος είναι ο ρόλος της λάμδα αφαίρεσης $(\lambda f.\lambda x.f\ (f\ (f\ x)))$; Απλοποιήστε με διαδοχικές αναγωγές και μετατροπές τη λάμδα έκφραση:

$$(\lambda f.\lambda x.f\ (f\ (f\ x)))\ (\lambda x. + x\ 7)\ 6$$

ώστε να φτάσει σε κανονική μορφή, ακολουθώντας την εφαρμοστική σειρά αναγωγής.



Σχήμα 12.12: Αναγωγή της λάμδα έκφρασης $(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.*\ 3\ x)\ 5$.

Πρόβλημα 12.3
Απλοποιήστε τη λάμδα έκφραση:

$$(\lambda f.\lambda x.f\ (f\ (f\ x)))\ (\lambda x.+\ x\ 7)\ 6$$

του Προβλήματος 12.2, ακολουθώντας την κανονική σειρά αναγωγής.

Πρόβλημα 12.4
Απλοποιήστε την έκφραση:

$$S\ (S\ (K\ +)\ I)\ I\ 8$$

εφαρμόζοντας κατάλληλες αναγωγές για τους I-, K- και S- συνδυαστές.

Πρόβλημα 12.5

Δείξτε σχηματικά, μέσω γράφων, τις αναγωγές που κάνατε στα Προβλήματα 12.2 και 12.3.

Βιβλιογραφικές αναφορές

[1] A. Church, *The Calculi of Lambda-Conversion*, Princeton University Press, 1941.

[2] A. J. Field and P. G. Harrison, *Functional Programming*, Addison Wesley, 1988.

[3] P. Hudak, Conception, Evolution, and Application of Functional Programming Languages, *ACM Comput. Surv.*, 21(3), 359-411, 1989.

[4] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

Γλωσσάρι

α-μετατροπή (α-conversion): Κανόνας μετασχηματισμού στον λάμδα λογισμό με τον οποίο επιτυγχάνεται η μετονομασία μεταβλητών.

αναγωγή γράφων (graph reduction): Τεχνική υλοποίησης συναρτησιακών γλωσσών, στην οποία ο γράφος που αντιστοιχεί σε μια λάμδα έκφραση απλοποιείται όσο το δυνατόν περισσότερο, εφαρμόζοντας ένα σύνολο από μετασχηματισμούς.

αναδρομή (recursion): Ορισμός κανόνα στον λογικό προγραμματισμό, που χρησιμοποιεί στο σώμα του το κατηγόρημα της κεφαλής του. Επίσης, ορισμός συνάρτησης στον συναρτησιακό προγραμματισμό μέσω του ίδιου της του εαυτού.

αντικατάσταση (substitution): Σύνολο από αναθέσεις όρων σε μεταβλητές, ως τιμές.

αποκοπή (cut): Ενσωματωμένο κατηγόρημα στην Prolog, για τον έλεγχο της οπισθοδρόμησης.

αρχή της ανάλυσης (resolution principle): Κανόνας συμπερασμού στη λογική πρώτης τάξης, με τον οποίο προκύπτει μια πρόταση από το συνδυασμό δύο άλλων.

άτομο (atom): Έκφραση στη λογική πρώτης τάξης, που χρησιμοποιείται για τη διατύπωση προτάσεων.

αφηρημένη μηχανή του Warren (Warren abstract machine): Ιδεατή μηχανή, η οποία μπορεί να εκτελεί προγράμματα που έχουν προκύψει από τη μεταγλώττιση προγραμμάτων Prolog στη γλώσσα αυτής της μηχανής.

β-αναγωγή (β-reduction): Κανόνας μετασχηματισμού στον λάμδα λογισμό, με τον οποίο εφαρμόζεται μια λάμδα αφαίρεση σε μια λάμδα έκφραση.

β-αφαίρεση (β-abstraction): Μετασχηματισμός ο οποίος είναι ο αντίστροφος της β-αναγωγής.

βαθμός (arity): Το πλήθος των ορισμάτων ενός κατηγορήματος ή ενός συναρτησιακού συμβόλου.

βάση Herbrand (Herbrand base): Το σύνολο των βασικών ατόμων μιας γλώσσας πρώτης τάξης.

βασικό άτομο (ground atom): Ένα άτομο χωρίς μεταβλητές.

βασικός όρος (ground term): Ένας όρος χωρίς μεταβλητές.

γεγονός (fact): Η απλούστερη δυνατή μορφή γνώσης στον λογικό προγραμματισμό και την Prolog, με την οποία εκφράζεται ότι ένας ισχυρισμός είναι αληθινός.

γενικότερος ενοποιητής (most general unifier): Ενοποιητής που δεν μπορεί να προκύψει ως εξειδίκευση άλλου ενοποιητή.

γλώσσα πρώτης τάξης (first order language): Το σύνολο των συντακτικά αποδεκτών τύπων στη λογική πρώτης τάξης, για δεδομένα σύνολα κατηγορημάτων, συναρτησιακών συμβόλων και μεταβλητών.

currying : Μετασχηματισμός μιας συνάρτησης που εφαρμόζεται σε ένα ζευγάρι τύπου (a, b), σε συνάρτηση που εφαρμόζεται σε δύο ορίσματα τύπων a και b, αντίστοιχα.

δ-αναγωγή (δ-reduction): Κανόνας μετασχηματισμού στον λάμδα λογισμό, με τον οποίο εφαρμόζεται μια ενσωματωμένη σταθερά, ως συνάρτηση, σε μια λάμδα έκφραση.

δ-αφαίρεση (δ-abstraction): Μετασχηματισμός ο οποίος είναι ο αντίστροφος της δ-αναγωγής.

δεσμευμένη μεταβλητή (bound variable): Στον λάμδα λογισμό, μεταβλητή που εισάγεται με μια λάμδα αφαίρεση.

δηλωτικός προγραμματισμός (declarative programming): Μεθοδολογία επίλυσης προβλημάτων, κατά την οποία διατυπώνονται τα αξιώματα που ισχύουν στον κόσμο ενός προβλήματος, αντί να περιγράφεται αναλυτικά η διαδικασία επίλυσής του.

διαδικαστικός προγραμματισμός (procedural programming): Μεθοδολογία επίλυσης προβλημάτων, κατά την οποία διατυπώνεται ένας αλγόριθμος για την επίλυση του προβλήματος που ενδιαφέρει.

ελάχιστο μοντέλο Herbrand (least Herbrand model): Η τομή όλων των μοντέλων ενός οριστικού προγράμματος.

ελεύθερη μεταβλητή (free variable): Στον λάμδα λογισμό, μεταβλητή που δεν είναι δεσμευμένη.

ενδοθεματικός τελεστής (infix operator): Τελεστής που βρίσκεται μεταξύ των δύο ορισμάτων στα οποία εφαρμόζεται.

ενθουσιώδης υπολογισμός (eager evaluation): Υιοθέτηση της εφαρμοστικής σειράς αναγωγής στον λάμδα λογισμό.

ενοποίηση (unification): Η διαδικασία υπολογισμού του γενικότερου ενοποιητή δύο όρων ή ατόμων.

ενοποιητής (unifier): Αντικατάσταση που, όταν εφαρμοστεί σε δύο όρους ή δύο άτομα, τους καθιστά απολύτως ίδιους.

ερμηνεία Herbrand (Herbrand interpretation): Ένα υποσύνολο της βάσης Herbrand.

ερώτηση (query): Η υποβολή σε ένα λογικό πρόγραμμα ενός προβλήματος προς επίλυση.

εφαρμοστική σειρά αναγωγής (applicative order reduction): Σειρά αναγωγής λάμδα εκφράσεων στον λάμδα λογισμό, στην οποία ανάγεται πρώτα η πιο αριστερή, πιο εσωτερική αναγώγιμη έκφραση.

η-μετατροπή (η-conversion): Κανόνας μετασχηματισμού στον λάμδα λογισμό, με τον οποίο εκφράζεται η ισοδυναμία μιας λάμδα αφαίρεσης με μια λάμδα έκφραση που δεν περιλαμβάνει ως ελεύθερη μεταβλητή τη δεσμευμένη μεταβλητή της λάμδα αφαίρεσης.

κανόνας (rule): Ένας τρόπος διατύπωσης γνώσης, υπό τη μορφή συνεπαγωγής, στον λογικό προγραμματισμό και την Prolog.

κανόνας αναγωγής (reduction rule): Κανόνας μετασχηματισμού στον λάμδα λογισμό, για την απλοποίηση λάμδα εκφράσεων.

κανόνας συμπερασμού (inference rule): Ένα εργαλείο παραγωγής νέας γνώσης από ήδη υπάρχουσα.

κανονική σειρά αναγωγής (normal order reduction): Σειρά αναγωγής λάμδα εκφράσεων στον λάμδα λογισμό, στην οποία ανάγεται πρώτα η πιο αριστερή, πιο εξωτερική αναγώγιμη έκφραση.

κατηγόρημα (predicate): Ένα σύμβολο με το οποίο μπορούμε να εκφράσουμε στον λογικό προγραμματισμό έναν ισχυρισμό, μια ιδιότητα αντικειμένου ή μια σχέση μεταξύ αντικειμένων.

κενή λίστα (empty list): Λίστα χωρίς στοιχεία.

κεφαλή λίστας (head of list): Το πρώτο στοιχείο μιας λίστας.

κόμβος εφαρμογής (apply node): Κόμβος σε έναν γράφο που χρησιμοποιείται για την αναπαράσταση της εφαρμογής μιας λάμδα έκφρασης σε μια λάμδα έκφραση.

λ-κόμβος (λ-node): Κόμβος σε έναν γράφο που χρησιμοποιείται για την αναπαράσταση μιας λάμδα αφαίρεσης.

λάμδα αφαίρεση (lambda abstraction): Μια κατηγορία λάμδα έκφρασης στον λάμδα λογισμό, με την οποία ορίζεται μια ανώνυμη συνάρτηση.

λάμδα έκφραση (lambda expression): Μια έκφραση στον λάμδα λογισμό.

λάμδα λογισμός (lambda calculus): Αξιωματικά ορισμένο μαθηματικό σύστημα, για την περιγραφή της υπολογιστικής συμπεριφοράς των μαθηματικών συναρτήσεων.

λειτουργική σημασιολογία (operational semantics): Μεθοδολογία μελέτης της σημασίας των οριστικών προγραμμάτων, που προτείνει έναν αλγόριθμο με τον οποίο μπορούμε να αποδείξουμε αν κάτι είναι λογικό επακόλουθο ενός οριστικού προγράμματος.

λίστα (list): Διατεταγμένη ακολουθία από στοιχεία, τόσο στην Prolog όσο και στη Haskell.

λογική πρώτης τάξης (first order logic): Το είδος της λογικής που αποτελεί το θεωρητικό υπόβαθρο του λογικού προγραμματισμού.

λογικός προγραμματισμός (logic programming): Δηλωτική φιλοσοφία προγραμματισμού, που συνίσταται στη διατύπωση αξιωμάτων υπό τη μορφή γεγονότων και κανόνων.

λογικός προγραμματισμός με περιορισμούς (constraint logic programming): Επέκταση του λογικού προγραμματισμού με ένα σύνολο περιορισμών, η οποία στοχεύει στην αποδοτική αντιμετώπιση προβλημάτων ικανοποίησης περιορισμών.

μεταβλητή (variable): Παριστάνει, τόσο στον λογικό όσο και στον συναρτησιακό προγραμματισμό, ένα τυχαίο αντικείμενο, που παίρνει τιμές από ένα πεδίο.

μεταθεματικός τελεστής (postfix operator): Τελεστής που έπεται του ορίσματος επάνω στο οποίο εφαρμόζεται.

μοντέλο Herbrand (Herbrand model): Μια ερμηνεία Herbrand, στην οποία ένας τύπος είναι αληθής.

μοντελοθεωρητική σημασιολογία (model-theoretic semantics): Μεθοδολογία μελέτης της σημασίας των οριστικών προγραμμάτων, που βασίζεται σε ερμηνείες και μοντέλα.

modus ponens : Κανόνας συμπερασμού στη λογική πρώτης τάξης, ο οποίος εφαρμόζεται στην περίπτωση των προτάσεων Horn.

οκνηρός υπολογισμός (lazy evaluation): Υιοθέτηση της κανονικής σειράς αναγωγής στον λάμδα λογισμό.

οριστική πρόταση (definite clause): Πρόταση στη λογική πρώτης τάξης, της οποίας η κεφαλή αποτελείται μόνο από ένα άτομο.

οριστικό πρόγραμμα (definite program): Σύνολο οριστικών προτάσεων.

οριστικός στόχος (definite goal): Πρόταση στη λογική πρώτης τάξης, της οποίας η κεφαλή είναι κενή.

όρος (term): Έκφραση στη λογική πρώτης τάξης, με την οποία παριστάνεται ένα αντικείμενο, τυχαίο ή συγκεκριμένο, απλό ή σύνθετο.

ουρά λίστας (tail of list): Η λίστα την οποία συνιστούν όλα τα στοιχεία μιας λίστας, εκτός από το πρώτο.

περιφραστική λίστα (list comprehension): Διατύπωση μιας λίστας στη Haskell με έμμεσο τρόπο.

πρόβλημα ικανοποίησης περιορισμών (constraint satisfaction problem): Πρόβλημα που μπορεί να μοντελοποιηθεί από ένα σύνολο μεταβλητών και ένα σύνολο περιορισμών μεταξύ των μεταβλητών αυτών.

προθεματικός τελεστής (prefix operator): Τελεστής που προηγείται του ορίσματος επάνω στο οποίο εφαρμόζεται.

πρόταση (clause): Καθολικά ποσοτικοποιημένη διάζευξη από στοιχειώδεις τύπους, στη λογική πρώτης τάξης.

πρόταση Horn (Horn clause): Πρόταση στη λογική πρώτης τάξης, της οποίας η κεφαλή είτε αποτελείται μόνο από ένα άτομο είτε είναι κενή.

προτασιακή λογική (propositional logic): Η κατηγορία της λογικής στην οποία διατυπώνονται ισχυρισμοί, με τη βοήθεια προτασιακών συμβόλων.

σημασιολογία σταθερού σημείου (fixpoint semantics): Μεθοδολογία μελέτης της σημασίας των οριστικών προγραμμάτων, που παρέχει μια διαδικασία κατασκευής του ελάχιστου μοντέλου Herbrand ενός προγράμματος.

σταθερό σημείο (fixpoint): Ένα στοιχείο του πεδίου ορισμού μιας συνάρτησης, του οποίου η εικόνα μέσω της συνάρτησης είναι το ίδιο το στοιχείο.

στοιχειώδης τύπος (literal): Ένα άτομο ή η άρνηση ενός ατόμου, στη λογική πρώτης τάξης.

σύμπαν Herbrand (Herbrand universe): Το σύνολο των βασικών όρων μιας γλώσσας πρώτης τάξης.

συνάρτηση ανώτερης τάξης (higher-order function): Συνάρτηση που δέχεται ως όρισμα ή δίνει ως αποτέλεσμα κάποια συνάρτηση, ή και τα δύο.

συναρτησιακό σύμβολο (function symbol): Σύμβολο που χρησιμοποιείται στον λογικό προγραμματισμό για τη δόμηση σύνθετων όρων.

συναρτησιακός προγραμματισμός (functional programming): Δηλωτική φιλοσοφία προγραμματισμού, που συνίσταται στη διατύπωση αξιωμάτων υπό τη μορφή ορισμού συναρτήσεων.

συνδυαστής (combinator): Λάμδα αφαίρεση, που δεν περιέχει ελεύθερες μεταβλητές.

Y-συνδυαστής (Y-combinator): Συνδυαστής που χρησιμοποιείται για τον ορισμό αναδρομικών συναρτήσεων στον λάμδα λογισμό.