ΑΡΧΕΣ ΓΛΩΣΣΩΝ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Ν. Παπασπύρου και Π. Ροντογιάννης

Ακαδημαϊκό Έτος 2007-2008

Περιεχόμενα

1 Εισαγωγή **1**

2 Το Συντακτικό των Γλωσσών Προγραμματισμού **4**
2.1 Γραμματικές χωρίς συμφραζόμενα 4
2.2 Ο Φορμαλισμός BNF 5

3 Η Σημασιολογία των Γλωσσών Προγραμματισμού **7**
3.1 Εισαγωγή 7
3.2 Μέθοδοι Σημασιολογίας 8
3.2.1 Η Μηχανική Σημασιολογία 9
3.2.2 Η Αξιωματική Σημασιολογία 9
3.2.3 Η Ενδεικτική Σημασιολογία 9
3.2.4 Παρατηρήσεις 9
3.3 Ενα Παράδειγμα: Δυαδικές Ακολουθίες 10
3.4 Συνθεσιμότητα 12

4 Σημασιολογία Μιας Απλής Προστακτικής Γλώσσας **13**
4.1 Εισαγωγή 13
4.2 Εκφράσεις και Εντολές 13
4.2.1 Συντακτικό 13
4.2.2 Σημασιολογία 15
4.2.3 Παρατηρήσεις 17
4.3 Εντολές Ανάθεσης 18

5 Αξιωματική Σημασιολογία και Απόδειξη Ορθότητας Προγραμμάτων **20**
5.1 Η Γλώσσα Προγραμματισμού 20
5.2 Η Τεχνική των Floyd-Hoare 21
5.3 Το Αξίωμα του `skip` και ο Κανόνας των Παρενθέσεων 23
5.4 Το Αξίωμα της Εντολής Ανάθεσης 23
5.5 Ο Κανόνας της Ενδυνάμωσης της Προσυνθήκης 24
5.6 Ο Κανόνας της Αποδυνάμωσης της Μετασυνθήκης 24
5.7 Οι κανόνες της Σύζευξης και της Διάζευξης 25
5.8 Ο Κανόνας των Σύνθετων Εντολών 25
5.9 Ο Κανόνας του `if` 26
5.10 Ο Κανόνας του `while` 26

6 Υλοποίηση Γλωσσών Προγραμματισμού **29**
6.1 Μεταγλωττιστές 29
6.1.1 Λεκτική Ανάλυση 29

6.1.2 Συντακτική Ανάλυση . . . 30
6.1.3 Σημασιολογική Ανάλυση . . . 30
6.1.4 Βελτιστοποίηση . . . 31
6.1.5 Παραγωγή Τελικού Κώδικα . . . 31
6.2 Διαδικασίες . . . 31
6.2.1 Μέθοδοι Περάσματος Παραμέτρων . . . 32
6.2.2 Εγγραφές Ενεργοποίησης . . . 34

7 Λογικός Προγραμματισμός: Η Γλώσσα Prolog 36
7.1 Τα Γεγονότα . . . 36
7.2 Οι Ερωτήσεις . . . 37
7.3 Οι Κανόνες . . . 38
7.4 Οι Όροι . . . 38
7.5 Εκτέλεση Προγραμμάτων Prolog . . . 40
7.6 Απλά Αναδρομικά Προγράμματα σε Prolog . . . 42
7.7 Αναδρομικός Προγραμματισμος με Λίστες . . . 42
7.8 Αναδρομικός Προγραμματισμος με Δέντρα . . . 44
7.9 Τελεστές . . . 46
7.10 Αριθμητικές Πράξεις και Prolog . . . 47
7.11 Η Αποκοπή . . . 48
7.12 Ασκήσεις . . . 50

8 Συναρτησιακός Προγραμματισμός: Η Γλώσσα Haskell 52
8.1 Συναρτησιακός Προγραμματισμός . . . 52
8.2 Εισαγωγή στη Γλώσσα Haskell . . . 54
8.3 Τύποι της Haskell . . . 54
8.4 Ορίζοντας Συναρτήσεις στη Haskell . . . 57
8.5 Αναδρομικά Προγράμματα με Ακεραίους . . . 59
8.6 Αναδρομικά Προγράμματα με Λίστες . . . 60
8.7 Μέθοδος Κλήσης Συναρτήσεων στην Haskell . . . 64
8.8 Πολυμορφισμός στη Haskell . . . 65
8.9 Άπειρες Δομές Δεδομένων . . . 65
8.10 Συναρτήσεις Υψηλής Τάξης . . . 66
8.11 Τύποι Οριζόμενοι από το Χρήστη . . . 67
8.12 Ασκήσεις . . . 68

9 Λάμβδα λογισμός 70
9.1 Μια διαισθητική εισαγωγή . . . 71
9.2 Λάμβδα όροι . . . 73
9.3 Αντικατάσταση . . . 76
9.4 Μετατροπές . . . 77
9.4.1 α-μετατροπή . . . 78
9.4.2 β-μετατροπή . . . 79
9.4.3 η-μετατροπή . . . 80
9.4.4 Μετατροπή, αναγωγή και ισότητα . . . 80
9.5 Κανονικές μορφές . . . 82
9.6 Ιδιότητες του λογισμού . . . 85
9.7 Η εκφραστική δύναμη του λάμβδα λογισμού . . . 88
9.7.1 Λογικές τιμές . . . 88

9.7.2 Διατεταγμένα ζεύγη 89
9.7.3 Φυσικοί αριθμοί 90
9.7.4 Λάμβδα λογισμός και πέρασμα παραμέτρων 93
Ασκήσεις 94

Κεφάλαιο 1

Εισαγωγή

Ο σχεδιασμός και η υλοποίηση απλών και αποτελεσματικών γλωσσών προγραμματισμού παραμένει ένας από τους πιό φιλόδοξους και δύσκολους στόχους της επιστήμης των υπολογιστών. Οι ερευνητικές δραστηριότητες που σχετίζονται με την περιοχή των γλωσσών προγραμματισμού ξεκινούν από τις πιό εφαρμοσμένες (όπως για παράδειγμα είναι οι *τεχνικές υλοποίησης*) και φτάνουν μέχρι τις πιό θεωρητικές (όπως είναι η *σημασιολογία* που αφορά το μαθηματικό νόημα των γλωσσών αυτών).

Έχουν δημιουργηθεί μέχρι σήμερα αρκετές δεκάδες (αν όχι εκατοντάδες) γλωσσών, οι οποίες μπορούν να κατηγοριοποιηθούν με βάση κάποια κοινά χαρακτηριστικά τους. Στη συνέχεια δίνουμε μια τέτοια κατάταξη, η οποία καλύπτει ένα αρκετά μεγάλο μέρος των σύγχρονων γλωσσών προγραμματισμού:

- **Προστακτικές (Imperative)**. Είναι η πιό ευρέως διαδεδομένη κατηγορία γλωσσών προγραμματισμού. Περιλαμβάνει όλες τις "κλασικές" γλώσσες, όπως *C, Pascal, Fortran*, κλπ. Το συντακτικό των γλωσσών αυτών βρίσκεται σε πολύ στενή σχέση με την αρχιτεκτονική του υπολογιστή και κατά συνέπεια ο προγραμματιστής πρέπει συχνά να γνωρίζει και να λαμβάνει υπόψιν του τεχνικές λεπτομέρειες που είναι ανεξάρτητες από το πρόβλημα που θέλει να επιλύσει. Οι γλώσσες αυτές έχουν πάντως βρεί εφαρμογές σε όλα σχεδόν τα πεδία της επιστήμης των υπολογιστών.

- **Συναρτησιακές (Functional)**. Η κατηγορία αυτή βασίζεται στην έννοια της μαθηματικής *συνάρτησης*. Συναρτησιακές γλώσσες είναι η *Lisp*, η *ML*, η *Miranda*, η *Gofer*, η *Haskell* και άλλες. Ολες βασίζονται στο λεγόμενο *λ-λογισμό* που αναπτύχθηκε από τον A. Church γύρω στο 1940. Το μαθηματικό υπόβαθρο των γλωσσών αυτών συντελεί στο να μπορεί κανείς να αποδείξει με σχετική ευκολία την ορθότητα προγραμμάτων όπως και να ορίσει μαθηματικούς μετασχηματισμούς που μπορούν να μετατρέψουν ένα πρόγραμμα σε κάποιο άλλο σημασιολογικά ισοδύναμο αλλά πιό αποτελεσματικό από το αρχικό. Οι συναρτησιακές γλώσσες δεν έχουν ακόμη κάποιο

ευρύ πεδίο εφαρμογών (με εξαίρεση ίσως τη Lisp που έχει χρησιμοποιηθεί ευρέως στο πεδίο της τεχνητής νοημοσύνης).

- **Λογικές (Logic)**. Οι γλώσσες αυτές βασίζονται στη μαθηματική λογική. Κύριος εκπρόσωπος της κατηγορίας είναι η γλώσσα Prolog η οποία έχει αναπτυχθεί με βάση το λεγόμενο υποσύνολο Horn της κατηγορηματικής λογικής πρώτης τάξης. Για τις γλώσσες αυτές ισχύουν τα ίδια όπως και για τις συναρτησιακές: είναι απλές στον προγραμματισμό, έχουν ξεκάθαρη σημασιολογία και ευνοούν τις αποδείξεις ορθότητας προγραμμάτων. Ενα βασικό τους μειονέκτημα (το οποίο ισχύει και για τις συναρτησιακές) είναι οτι υστερούν από άποψη ταχύτητας σε σχέση με τις προστακτικές γλώσσες. Οι λογικές γλώσσες έχουν διάφορες εφαρμογές, με κύρια έμφαση στα έμπειρα συστήματα όπως και στο λεγόμενο συμβολικό υπολογισμό (πχ. προγράμματα για τη συμβολική επίλυση εξισώσεων).

- **Περιορισμών (Constraint)**. Οι γλώσσες αυτές παρέχουν στο χρήστη τη δυνατότητα να εκφράσει με φυσικό τρόπο διάφορους περιορισμούς που υπάρχουν στο πρόβλημα που επιχειρεί να επιλύσει. Για παράδειγμα, οι περιορισμοί αυτοί μπορεί να είναι ένα σύστημα εξισώσεων ή ανισώσεων (γραμμικών ή μή). Μια αρκετά επιτυχημένη τέτοια γλώσσα είναι η CLP (Constraint Logic Programming) που είναι μια επέκταση της Prolog που υποστηρίζει διάφορα είδη περιορισμών. Ενα κύριο τμήμα των γλωσσών αυτών είναι ο λεγόμενος επιλυτής περιορισμών (constraint solver) ο οποίος ανάλογα με τις εφαρμογές που καλείται να επιλύσει η γλώσσα, μπορεί να είναι από πολύ απλός μέχρι αρκετά πολύπλοκος. Η περιοχή των γλωσσών περιορισμών είναι σχετικά πρόσφατη αλλά έχει προφανώς πολλές εφαρμογές σε διάφορα πεδία της επιστήμης.

- **Αντικειμενοστρεφείς (Object-Oriented)**. Σε μια αντικειμενοστρεφή γλώσσα προγραμματισμού, ο υπολογισμός λαμβάνει χώρα υπό τη μορφή αντικειμένων (objects) τα οποία ανταλλάσουν μηνύματα (messages) μεταξύ τους. Οταν ένα αντικείμενο λαμβάνει ένα μήνυμα, καλεί μια κατάλληλη μέθοδο (method) για την εξυπηρέτηση του μηνύματος αυτού. Τα αντικείμενα είναι οργανωμένα σε μια ιεραρχία από κλάσεις (classes). Ολα τα αντικείμενα μιας δεδομένης κλάσης χρησιμοποιούν τις ίδιες μεθόδους για την εξυπηρέτηση παρομοίων μηνυμάτων. Οι αντικειμενοστρεφείς γλώσσες προγραμματισμού είναι ιδιαίτερα δημοφιλείς στις μέρες μας. Αρκετά επιτυχημένες τέτοιες γλώσσες είναι η *C++*, η *Smalltalk*, και η πιό πρόσφατη *Java*. Ιδιαίτερα, η Java έγινε πολύ γρήγορα αποδεκτή λόγω της στενής της σχέσης με το Internet.

- **Ροής-Δεδομένων (Dataflow)**. Τα προγράμματα ροής δεδομένων μπορεί κανείς να τα θεωρήσει σα δίκτυα που αποτελούνται από κόμβους (nodes) και κανάλια (channels)

που συνδέουν τους κόμβους αυτούς. Τα δεδομένα που θέλουμε να επεξεργαστούμε μπορούμε να φανταστούμε οτι "ρέουν" στα κανάλια, και γίνονται αντικείμενο επεξεργασίας όταν φτάνουν στους κόμβους. Γνωστές τέτοιες γλώσσες προγραμματισμού είναι η *Lucid*, η *Val*, η *Id*, η *Sisal*, και άλλες. Για τις γλώσσες αυτές έχουν προταθεί και ειδικές αρχιτεκτονικές πάνω στις οποίες μπορούν να εκτελεστούν πιο αποτελεσματικά (είναι οι λεγόμενες *αρχιτεκτονικές ροής δεδομένων*). Οι γλώσσες αυτές βασίζονται σε απλές μαθηματικές αρχές οπότε έχουν τα ίδια πλεονεκτήματα με τις συναρτησιακές και τις λογικές. Μια επέκταση των γλωσσών αυτών είναι οι λεγόμενες *οπτικές (visual)* γλώσσες προγραμματισμού. Οι γλώσσες ροής δεδομένων έχουν μέχρι σήμερα βρεί εφαρμογές κυρίως στο πεδίο του επιστημονικού υπολογισμού.

- **Παράλληλες (Parallel)**. Με την ανάπτυξη των παράλληλων υπολογιστών δημιουργήθηκε η ανάγκη νέων γλωσσών προγραμματισμού που θα ταίριαζαν περισσότερο στο νέο αυτό μοντέλο υπολογισμού. Εχει προταθεί μια πληθώρα παραλλήλων γλωσσών που περιλαμβάνει από επεκτάσεις κλασικών γλωσσών μέχρι εξειδικευμένες γλώσσες για συγκεκριμένες εφαρμογές. Το θεωρητικό υπόβαθρο αρκετών από τις γλώσσες αυτές είναι οι λεγόμενες *άλγεβρες διεργασιών (process algebras)* με κύριους εκπροσώπους τη *CCS* και τη *CSP*. Οι εφαρμογές των παραλλήλων γλωσσών είναι μεγάλες σε προβλήματα τα οποία χρειάζονται ιδιαίτερα υψηλή υπολογιστική ισχύ για να επιλυθούν.

Η έρευνα στην περιοχή των γλωσσών προγραμματισμού είναι όπως προαναφέραμε αρκετά πλατιά. Μερικά από τα θέματα που είναι αρκετά δημοφιλή στις μέρες μας είναι η *σημασιολογία (semantics)*, τα *συστήματα τύπων (type systems)*, η *ανάλυση και μετασχηματισμοί προγραμμάτων (program analysis and transformations)*, η *ορθότητα τεχνικών μετάφρασης (compiler correctness)*, κ.α. Στα επόμενα κεφάλαια των σημειώσεων αυτών θα εξεταστούν οι βασικές αρχές που διέπουν το σχεδιασμό και την υλοποίηση των σύγχρονων γλωσσών προγραμματισμού.

Κεφάλαιο 2

Το Συντακτικό των Γλωσσών Προγραμματισμού

2.1 Γραμματικές χωρίς συμφραζόμενα

Το συντακτικό (syntax) μιας γλώσσας προγραμματισμού καθορίζει τον τρόπο με τον οποίο μπορούν να δημιουργηθούν προγράμματα στη γλώσσα αυτή. Ο βασικός μαθηματικός φορμαλισμός που χρησιμοποιείται για την περιγραφή του συντακτικού, είναι αυτός των *γραμματικών χωρίς συμφραζόμενα (context-free languages)*. Οι γραμματικές αυτές δεν χρησιμοποιούνται μόνο για την περιγραφή γλωσσών προγραμματισμού, αλλά γενικότερα γλωσσών των οποίων οι φράσεις μπορούν να παραχθούν από την απλή χρήση ενός συνόλου κανόνων. Μια γραμματική χωρίς συμφραζόμενα αποτελείται από τέσσερα στοιχεία:

- Ένα σύνολο από τερματικά σύμβολα (terminals). Τα τερματικά σύμβολα αποτελούν τις ατομικές οντότητες οι οποίες αν συνδυαστούν κατάλληλα δημιουργούν ορθές φράσεις της γλώσσας.

- Ένα σύνολο από μή-τερματικά σύμβολα (non-terminals). Τα σύμβολα αυτά είναι μεταβλητές που αναπαριστούν φράσεις της γλώσσας.

- Ένα σύνολο από κανόνες (rules ή productions) που καθορίζουν πως δημιουργούνται οι φράσεις της γλώσσας. Ένας κανόνας αποτελείται από ένα μη-τερματικό σύμβολο που βρίσκεται στα αριστερά, το σύμβολο `::=`, και από μία ακολουθία από τερματικά και μη-τερματικά σύμβολα που βρίσκονται στα δεξιά.

- Ένα επιλεγμένο μη-τερματικό σύμβολο που παίζει το ρόλο του αρχικού συμβόλου της γλώσσας. Το σύμβολο αυτό αντιπροσωπεύει τη βασική δομή της γλώσσας που αναπαριστούμε.

Παράδειγμα 2.1 Το συντακτικό της γλώσσας των δυαδικών ακολουθιών μπορεί να περιγραφεί με τη βοήθεια μιας context-free γραμματικής, ως εξής:

$$
\begin{aligned}
\mathbf{S} &::= \mathtt{0} \\
\mathbf{S} &::= \mathtt{1} \\
\mathbf{S} &::= \mathbf{S}\mathtt{0} \\
\mathbf{S} &::= \mathbf{S}\mathtt{1}
\end{aligned}
$$

Σε πολλές περιπτώσεις, για να αποφεύγουμε την ύπαρξη πολλών κανόνων με το ίδιο μη-τερματικό σύμβολο στα αριστερά, θα χρησιμοποιούμε το σύμβολο | που διαβάζεται ως "ή". Έτσι, η παραπάνω γραμματική μπορεί να γραφεί και ως:

$$
\begin{aligned}
\mathbf{S} &::= \mathtt{0} \\
&\;\;| \quad \mathtt{1} \\
&\;\;| \quad \mathbf{S}\mathtt{0} \\
&\;\;| \quad \mathbf{S}\mathtt{1}
\end{aligned}
$$

Στη συνέχεια θα χρησιμοποιούμε τον παραπάνω τρόπο γραφής, ο οποίος είναι και πιό βολικός.

Αξίζει να σημειώσουμε ότι η παραπάνω γραμματική περιέχει δύο τερματικά σύμβολα, το 0 και το 1 και ένα μη-τερματικό σύμβολο, το S. Το S είναι και το αρχικό σύμβολο παραγωγής της γλώσσας της γραμματικής. □

Με τη χρήση των κανόνων μιας γραμματικής μπορεί κανείς να παράγει φράσεις που ανήκουν στη γλώσσα που αντιστοιχεί στη γραμματική. Μια τέτοια παραγωγή λαμβάνει χώρα ως εξής: ξεκινάμε από το αρχικό σύμβολο της γλώσσας και χρησιμοποιούμε αναδρομικά τους διάφορους κανόνες της γλώσσας μέχρι να παραχθεί μια φράση που αποτελείται μόνο από τερματικά σύμβολα. Η φράση αυτή ανήκει στη γλώσσα που παράγεται από τη συγκεκριμένη γραμματική. Είναι εύκολο να διαπιστώσει κανείς ότι η γραμματική του παραπάνω παραδείγματος παράγει το σύνολο όλων των ακολουθιών που αποτελούνται από τα ψηφία 0 και 1.

2.2 Ο Φορμαλισμός BNF

Ειδικά για τις γλώσσες προγραμματισμού χρησιμοποιείται μια παραλλαγή των γραμματικών χωρίς συμφραζόμενα, γνωστή και ως BNF (Backus Naur Form). Στο BNF τα μη-τερματικά σύμβολα τοποθετούνται ανάμεσα σε "<" και ">".

Παράδειγμα 2.2 Έστω ότι θέλουμε να περιγράψουμε τις ακολουθίες που αναπαριστούν

πραγματικούς αριθμούς. Η αναπαράσταση σε BNF δίνεται από τους ακόλουθους κανόνες:

$$\begin{array}{lcl} <\mathbf{real-number}> & ::= & <\mathbf{integer-part}>.<\mathbf{fraction}> \\ <\mathbf{integer-part}> & ::= & <\mathbf{digit}> \mid <\mathbf{integer-part}><\mathbf{digit}> \\ <\mathbf{fraction}> & ::= & <\mathbf{digit}> \mid <\mathbf{digit}><\mathbf{fraction}> \\ <\mathbf{digit}> & ::= & \mathsf{0} \mid \mathsf{1} \mid \mathsf{2} \mid \mathsf{3} \mid \mathsf{4} \mid \mathsf{5} \mid \mathsf{6} \mid \mathsf{7} \mid \mathsf{8} \mid \mathsf{9} \end{array}$$

□

Έχουν κατά καιρούς προταθεί διάφορες επεκτάσεις του BNF που το καθιστούν περισσότερο εκφραστικό. Μία από αυτές είναι το EBNF (Extended BNF) το οποίο επιτρέπει *λίστες* από στοιχεία. Έτσι, όταν γράφουμε $\{\mathbf{S}\}$ εννοούμε μηδέν ή περισσότερες επαναλήψεις του στοιχείου **S**.

Παράδειγμα 2.3 Ο κανόνας:

$$<\mathbf{statement-list}> \;::=\; \{<\mathbf{statement};>\}$$

είναι ισοδύναμος με τους ακόλουθους δύο κανόνες του κλασικού BNF:

$$\begin{array}{lcl} <\mathbf{statement-list}> & ::= & <\mathbf{empty}> \\ & \mid & <\mathbf{statement}>;<\mathbf{statement-list}> \end{array}$$

Το $<\mathbf{empty}>$ στο παραπάνω παράδειγμα αναπαριστά την κενή συμβολοσειρά. □

Επεκτάσεις του BNF όπως η παραπάνω, βοηθούν στη δημιουργία πιό σύντομων και κατανοητών περιγραφών.

Στη συνέχεια δεν πρόκειται να ασχοληθούμε περισσότερο με θέματα που αφορούν αποκλειστικά το συντακτικό των γλωσσών προγραμματισμού. Στο επόμενο κεφάλαιο θα εξετάσουμε την έννοια της *σημασιολογίας* των γλωσσών η οποία περιγράφει το μαθηματικό νόημα των προγραμμάτων.

Κεφάλαιο 3

Η Σημασιολογία των Γλωσσών Προγραμματισμού

3.1 Εισαγωγή

Οσοι σχεδιάζουν, υλοποιούν ή ακόμη και χρησιμοποιούν μιά γλώσσα προγραμματισμού, χρειάζεται να γνωρίζουν τη γλώσσα από δύο τελείως διαφορετικές σκοπιές:

- Από τη σκοπιά του *συντακτικού* (syntax)
- Από τη σκοπιά της *σημασιολογίας* (semantics)

Για την αποτελεσματική περιγραφή του συντακτικού, υπάρχει όπως προαναφέραμε η θεωρία των γραμματικών χωρίς συμφραζόμενα.

Για την περιγραφή της σημασιολογίας όμως, για πολύ καιρό οι ερευνητές κατέφευγαν σε άτυπες μεθόδους (όπως για παράδειγμα είναι η χρησιμοποίηση της φυσικής γλώσσας).

Παράδειγμα 3.1 Ας θεωρήσουμε την περίπτωση του βρόχου `while` ο οποίος έχει τη μορφή

`while` ⟨λογική συνθήκη⟩ `do` ⟨εντολές⟩

Μιά άτυπη περιγραφή της σημασιολογίας του `while` μπορεί να είναι η ακόλουθη:

> "Αρχικά γίνεται ο έλεγχος της λογικής συνθήκης. Αν το αποτέλεσμα είναι αληθές, τότε γίνεται είσοδος στο βρόχο και οι εντολές που βρίσκονται μέσα στον βρόχο εκτελούνται μία φορά. Στη συνέχεια η συνθήκη ελέγχεται και πάλι. Οταν η συνθήκη γίνει ψευδής, ο βρόχος παρακάμπτεται και ο έλεγχος μεταφέρεται στην πρώτη εντολή που ακολουθεί τη δομή του βρόχου".

□

Τα πράγματα όμως δεν είναι πάντα τόσο απλά όσο στην παραπάνω περίπτωση. Ο αρχικός ορισμός της ALGOL 60 γιά παράδειγμα, περιείχε διφορούμενους ορισμούς, και έτσι ένας νέος "διορθωμένος" ορισμός προτάθηκε το 1963. Τα προβλήματα συνεχίστηκαν, και το 1967 ο D. Knuth έγραψε μιά αναφορά με τα λάθη των ορισμών του 1963. Εγραφε:

> "Οταν η ALGOL 60 πρωτοδημοσιεύτηκε το 1960, πολλά νέα στοιχεία δημιουργήθηκαν γιά τις γλώσσες προγραμματισμού... Ηταν δύσκολο αρχικά για οποιονδήποτε να καταλάβει την σημασία του κάθε νέου στοιχείου σε σχέση με όλα τα άλλα στοιχεία τα οποία διέθετε η γλώσσα, και πολλοί άνθρωποι όταν διάβαζαν την αναφορά της γλώσσας συχνά ανακάλυπταν πράγματα τα οποία πριν κανείς δεν γνώριζε οτι ήταν δυνατά. Συχνά τα νεοανακαλυφθέντα αυτά στοιχεία έδιναν αντιπαραδείγματα στις μέχρι τότε γνωστές τεχνικές για την υλοποίηση της γλώσσας, και σε πολλές περιπτώσεις μπορούσε κανείς να κατασκευάσει προγράμματα τα οποία ήταν δυνατό να υλοποιηθούν με διαφορετικούς τρόπους (δίνοντας διαφορετικά αποτελέσματα)".

Παράδειγμα 3.2 Είναι στην ALGOL 60 ισοδύναμες οι εκφράσεις `x+f(x)` και `f(x)+x`; Η απάντηση εξαρτάται από το εάν η `f` αλλάζει την τιμή του `x`. Εάν συμβαίνει κάτι τέτοιο, τότε η τιμή των δύο παραπάνω εκφράσεων μπορεί να είναι διαφορετική ανάλογα με τη σειρά που η υλοποίηση της γλώσσας υπολογίζει τις τιμές των εκφράσεων `x` και `f(x)`. □

Μιά αυστηρή μαθηματική θεωρία της σημασιολογίας των γλωσσών προγραμματισμού είναι επομένως αναγκαία έτσι ώστε να εξασφαλιστούν:

- Η ορθή περιγραφή και υλοποίηση των γλωσσών προγραμματισμού
- Η συστηματική ανάπτυξη και απόδειξη ορθότητας των προγραμμάτων.
- Η ανάλυση και αξιολόγηση των υπαρχόντων γλωσσών προγραμματισμού
- Ο σχεδιασμός νέων, απλούστερων αλλά και πιό ισχυρών γλωσσών προγραμματισμού.

Η θεωρία αυτή της σημασιολογίας και οι εφαρμογές της είναι το αντικείμενο του κεφαλαίου αυτού.

3.2 Μέθοδοι Σημασιολογίας

Τρείς είναι οι κύριες μέθοδοι που έχουν αναπτυχθεί για την περιγραφή της σημασιολογίας των γλωσσών προγραμματισμού: η *μηχανική σημασιολογία (operational semantics)*, η *αξιωματική σημασιολογία (axiomatic semantics)* και η *δηλωτική σημασιολογία (denotational semantics)*.

3.2.1 Η Μηχανική Σημασιολογία

Στη μεθοδολογία αυτή ορίζεται αρχικά ένα "αφηρημένο μοντέλο μηχανής", το οποίο αποτελείται από την *κατάσταση (state)* και από ένα σύνολο *βασικών εντολών*. Κατόπιν, η σημασιολογία μιας γλώσσας προγραμματισμού μπορεί να οριστεί με βάση το μοντέλο αυτό. Η βασική ιδέα είναι ότι αν και το μοντέλο το οποίο χρησιμοποιείται είναι πολλές φορές μη-ρεαλιστικό από πρακτική σκοπιά, είναι τόσο απλό ώστε να μην αφήνει περιθώρια για παρανοήσεις. Η σημασιολογία της γλώσσας καθορίζεται ως μετάφραση των βασικών της εντολών στην γλώσσα του αφηρημένου μοντέλου. Γλώσσες που έχουν οριστεί με τη μέθοδο αυτή είναι η PL/1 (χρησιμοποιώντας τη μέθοδο VDM), όπως επίσης και η ALGOL 68.

3.2.2 Η Αξιωματική Σημασιολογία

Στη μεθοδολογία αυτή, ένα αξίωμα συνδέεται με κάθε είδος εντολής που υπάρχει στη γλώσσα προγραμματισμού. Το αξίωμα αυτό προσδιορίζει τι ακριβώς μπορεί να θεωρηθεί ως αληθές μετά την εκτέλεση της εντολής, σε σχέση με αυτά που ίσχυαν πρίν από την εκτέλεση της εντολής. Γιά παράδειγμα, αν θεωρήσουμε το βρόχο

`while` **B** `do` **C**

όπου **B** είναι μιά έκφραση και **C** μιά εντολή, γνωρίζουμε ότι όταν ο βρόχος ολοκληρωθεί (εάν ποτέ ολοκληρωθεί) η τιμή της έκφρασης **B** θα είναι ψευδής.

Το προτέρημα της μεθόδου είναι ότι πολλές φορές τα αξιώματα που εισάγονται μπορούν να εκφραστούν στην ίδια τη γλώσσα προγραμματισμού (με ελάχιστες αλλαγές στο συντακτικό της). Αυτό είναι σημαντικό διότι εάν η απόδειξη ορθότητας ενός προγράμματος γίνει ποτέ μέλημα του ίδιου του προγραμματιστή, είναι λογικό να περιμένει κανείς οτι τόσο η απόδειξη όσο και ο προγραμματισμός θα γίνονται στην ίδια γλώσσα.

3.2.3 Η Ενδεικτική Σημασιολογία

Στη μεθοδολογία αυτή ορίζονται "συναρτήσεις υπολογισμού" (valuation functions), οι οποίες αντιστοιχίζουν τα συντακτικά στοιχεία της γλώσσας σε γνωστά μαθηματικά αντικείμενα (π.χ. αριθμούς, τιμές αληθείας, συναρτήσεις, κλπ). Οι συναρτήσεις υπολογισμού ορίζονται συνήθως αναδρομικά: η τιμή που θα λάβει μιά έκφραση της γλώσσας καθορίζεται με βάση τις τιμές των υποεκφράσεων που την αποτελούν.

3.2.4 Παρατηρήσεις

Και οι τρεις παραπάνω μέθοδοι αποτελούν αποδεκτές τεχνικές για τον καθορισμό της σημασιολογίας μιας γλώσσας προγραμματισμού. Διαφέρουν όμως σημαντικά ως προς τη

φιλοσοφία τους. Ανακεφαλαιώνοντας, μπορεί κανείς να πει οτι η Μηχανική μέθοδος ταιριάζει περισσότερο σε αυτούς που υλοποιούν μια γλώσσα, η Αξιωματική είναι πιό αποδεκτή από τους προγραμματιστές ενώ η Ενδεικτική είναι απαραίτητο να χρησιμοποιείται από τους σχεδιαστές της γλώσσας. Στη συνέχεια θα επικεντρώσουμε το ενδιαφέρον μας στις τεχνικές που χρησιμοποιούνται από την Ενδεικτική μεθοδολογία.

3.3 Ενα Παράδειγμα: Δυαδικές Ακολουθίες

Σαν ένα εισαγωγικό παράδειγμα, θεωρούμε τη στοιχειώδη "γλώσσα" των δυαδικών ακολουθιών, για την οποία δίνουμε το συντακτικό και τη σημασιολογία της.

Συντακτικό

Το συντακτικό της γλώσσας των δυαδικών ακολουθιών μπορεί να περιγραφεί με τη βοήθεια μιας context-free γραμματικής, ως εξής:

$$\begin{array}{rcl} \mathbf{S} & ::= & \mathtt{0} \\ & | & \mathtt{1} \\ & | & \mathbf{S}\mathtt{0} \\ & | & \mathbf{S}\mathtt{1} \end{array}$$

Εστω Seq το σύνολο των ακολουθιών που παράγονται με βάση τον παραπάνω ορισμό. Ο ορισμός αυτός μας λέει ότι το Seq αποτελείται από πεπερασμένες ακολουθίες των ψηφίων 0 και 1. Όμως, προς το παρόν δεν γνωρίζουμε τι αναπαριστούν οι ακολουθίες αυτές, δεν ξέρουμε τίποτα δηλαδή γιά το *νόημα* τους.

Ενδεικτική Σημασιολογία:

Μπορούμε να δώσουμε νόημα στις δυαδικές ακολουθίες ψηφίων, αν αντιστοιχίσουμε σε κάθε μία από αυτές έναν μοναδικό φυσικό αριθμό. Γιά το σκοπό αυτό χρειάζεται να ορίσουμε μιά συνάρτηση που ουσιαστικά μετατρέπει κάθε ακολουθία σε έναν φυσικό. Τη συνάρτηση αυτή θα τη συμβολίζουμε με $[\![\,\cdot\,]\!]$ (ο συμβολισμός αυτός θα χρησιμοποιείται πολύ συχνά από εδώ και μπρός, κάθε φορά που θέλουμε να ορίσουμε τη σημασιολογία μιας γλώσσας). Στη συγκεκριμένη περίπτωση, η $[\![\,\cdot\,]\!]$ μετατρέπει ακολουθίες δυαδικών ψηφίων σε φυσικούς αριθμούς (δηλαδή $[\![\,\cdot\,]\!] : Seq \rightarrow Nat$), και ορίζεται ως ακολούθως:

$$\begin{array}{rcl} [\![0]\!] & = & 0 \\ [\![1]\!] & = & 1 \\ [\![\mathbf{S}0]\!] & = & 2 \times [\![\mathbf{S}]\!] \\ [\![\mathbf{S}1]\!] & = & 2 \times [\![\mathbf{S}]\!] + 1 \end{array}$$

Ο παραπάνω ορισμός προσφέρει μιά μέθοδο γιά τον υπολογισμό του δεκαδικού αριθμού που αντιστοιχεί σε δεδομένο δυαδικό.

Παράδειγμα 3.3 Το νόημα της ακολουθίας `1100` μπορεί να δοθεί χρησιμοποιώντας τον ορισμό της σημασιολογίας των δυαδικών ακολουθιών, ως εξής:

$$\begin{aligned}
[\![\texttt{1100}]\!] &= 0 \\
&= 2 \times [\![\texttt{110}]\!] \\
&= 2 \times (2 \times [\![\texttt{11}]\!]) \\
&= 2 \times (2 \times (2 \times [\![\texttt{1}]\!] + 1)) \\
&= 2 \times (2 \times (2 \times 1 + 1)) \\
&= 12
\end{aligned}$$

□

Παρατηρήστε οτι στον παραπάνω ορισμό υπάρχει μία εξίσωση για κάθε κανόνα του συντακτικού. Δεν είναι εντελώς προφανές οτι οι παραπάνω εξισώσεις ορίζουν μια μαθηματική συνάρτηση: το σύμβολο $[\![\cdot]\!]$ εμφανίζεται και στη δεξιά αλλά και στην αριστερή πλευρά των εξισώσεων, έχουμε δηλαδή έναν αναδρομικό ορισμό της $[\![\cdot]\!]$. Μπορεί να δειχθεί οτι η $[\![\cdot]\!]$ είναι όντως συνάρτηση, και αυτό διαισθητικά οφείλεται στο γεγονός οτι στο δεξιό μέλος των παραπάνω εξισώσεων η $[\![\cdot]\!]$ εφαρμόζεται σε μιά υποέκφραση της έκφρασης στην οποία εφαρμόζεται στο αριστερό (πχ. στην τρίτη εξίσωση, η $[\![\cdot]\!]$ στα αριστερά εφαρμόζεται στην έκφραση $\mathbf{S}\texttt{0}$ ενώ στα δεξιά στην υποέκφραση $\mathbf{S}$).

Ενα άλλο σημαντικό σημείο που πρέπει να τονιστεί αφορά εξισώσεις της μορφής:

$$[\![\texttt{0}]\!] = 0$$

πού δόθηκε παραπάνω. Δικαιολογημένα μπορεί κανείς να αναρρωτηθεί αν εξισώσεις σαν και αυτή έχουν κάποιο ουσιαστικό νόημα, ή απλώς επαναλαμβάνουν το ίδιο πράγμα.

Η αλήθεια είναι ότι η εξίσωση αυτή δεν είναι κυκλική. Το `0` που εμφανίζεται στα αριστερά είναι ένα σύμβολο της *γλώσσας-αντικείμενο (object language)*. Η γλώσσα-αντικείμενο είναι η γλώσσα την οποία μελετάμε και στην οποία θέλουμε να δώσουμε νόημα. Στη συγκεκριμένη περίπτωση η γλώσσα αυτή είναι το σύνολο των δυαδικών ακολουθιών, αλλά θα μπορούσε κάλλιστα να είναι μιά οποιαδήποτε γλώσσα προγραμματισμού (πχ. C, Pascal).

Αντίθετα, to 0 που εμφανίζεται στα δεξιά είναι ο φυσικός αριθμός μηδέν που γνωρίζουμε από τα μαθηματικά. Οπως έχουμε πει, σκοπός της ενδεικτικής σημασιολογίας είναι να μετατρέπει τα συντακτικά στοιχεία μιας γλώσσας προγραμματισμού σε γνωστά μαθηματικά αντικείμενα. Τα μαθηματικά αποτελούν λοιπόν την λεγόμενη *μεταγλώσσα (metalanguage)* στην οποία και θα μετατρέπουμε τα προγράμματα της γλώσσας αντικείμενο.

3.4 Συνθεσιμότητα

Η σημασιολογική περιγραφή των δυαδικών ακολουθιών που παρουσιάσαμε στην προηγούμενη παράγραφο έχει την σημαντική ιδιότητα ότι:

> "Το νόημα κάθε σύνθετης εκφράσης δίνεται ως συνάρτηση των νοημάτων των υποεκφράσεων που την αποτελούν"

Η ιδιότητα αυτή ονομάζεται *συνθεσιμότητα*.

Παράδειγμα 3.4 Οι εξισώσεις στον ορισμό της σημασιολογίας των δυαδικών ακολουθιών υπακούουν στον κανόνα της συνθεσιμότητας, διότι ανάγουν τον υπολογισμό της σημασιολογίας των εκφράσεων **S0** και **S1** στην σημασιολογία της απλούστερης έκφρασης **S**. □

Ενα παράδειγμα σημασιολογίας που δεν υπακούει στον κανόνα της συνθεσιμότητας είναι το ακόλουθο:

$$[\![\texttt{while }\mathbf{B}\texttt{ do }\mathbf{C}]\!] = [\![\texttt{if }\mathbf{B}\texttt{ then }(\mathbf{C};\texttt{while }\mathbf{B}\texttt{ do }\mathbf{C})]\!]$$

Παρόλο που η ιδιότητα που εκφράζεται από την παραπάνω εξίσωση είναι κάτι που θα θέλαμε να ισχύει, η εξίσωση δεν υπακούει στον κανόνα της συνθεσιμότητας διότι η έκφραση που περικλείεται από το σύμβολο $[\![\cdot]\!]$ στη δεξιά πλευρά δεν αποτελεί υποέκφραση αυτής που εμφανίζεται στην αριστερή πλευρά. Ακόμη και αν γράφαμε κάποια εξίσωση της μορφής:

$$[\![\texttt{while }\mathbf{B}\texttt{ do }\mathbf{C}]\!] = \cdots[\![\mathbf{B}]\!]\cdots[\![\mathbf{C}]\!]\cdots[\![\texttt{while }\mathbf{B}\texttt{ do }\mathbf{C}]\!]\cdots$$

και πάλι δεν θα ίσχυε ο κανόνας της συνθεσιμότητας. Παρόλ'αυτά, όπως μπορεί να αποδειχτεί, μια εξίσωση όπως η παραπάνω μπορεί να λυθεί και η λύση της να χρησιμοποιηθεί ώστε να δοθεί ένας ορισμός της σημασιολογίας του `while` ο οποίος υπακούει στον κανόνα της συνθεσιμότητας. Θεματα όμως όπως αυτό ξεφεύγουν από τα πλαίσια των σημειώσεων αυτών και δεν πρόκειται να τα μελετήσουμε εκτενέστερα.

Κεφάλαιο 4

Σημασιολογία Μιας Απλής Προστακτικής Γλώσσας

4.1 Εισαγωγή

Στο κεφάλαιο αυτό θα μελετήσουμε μια γλώσσα προγραμματισμού που αποτελείται από *εκφράσεις (expressions)* και *εντολές (commands)*. Η βασική σημασιολογική έννοια που θα χρησιμοποιήσουμε είναι αυτή της *κατάστασης (state)*, των τιμών δηλαδή που έχουν οι μεταβλητές στη μνήμη του υπολογιστή. Γενικά, η τιμή που έχει μιά έκφραση του προγράμματος εξαρτάται από την παρούσα κατάσταση του υπολογιστή. Η κατάσταση αυτή μπορεί να αλλάξει με την εκτέλεση μιας εντολής (πχ. με την καταχώρηση μιας νέας τιμής σε μιά από τις μεταβλητές του προγράμματος). Η πορεία εκτέλεσης των εντολών ενός προγράμματος καθορίζεται από τις λεγόμενες *δομές ελέγχου* (όπως είναι για παράδειγμα το `if-then-else`, το `for`, το `while`, κλπ). Οι γλώσσες που υποστηρίζουν το παραπάνω στύλ προγραμματισμού ονομάζονται *προστακτικές*.

Η προστακτική γλώσσα που θεωρούμε σε αυτό το κεφάλαιο είναι "απλή" με την έννοια οτι δεν υποστηρίζει `goto`, σύνθετες δομές δεδομένων, δείκτες, και άλλα χαρακτηριστικά που είναι πολύ συνηθισμένα σε μια τυπική προστακτική γλώσσα. Καθώς σκοπός μας είναι η παρουσίαση των κύριων ιδεών της σημασιολογίας των γλωσσών προγραμματισμού, δεν θα εξετάσουμε στις σημειώσεις αυτές τα "προχωρημένα" αυτά χαρακτηριστικά.

4.2 Εκφράσεις και Εντολές

4.2.1 Συντακτικό

Η απλή προστακτική γλώσσα που θα ορίσουμε υποστηρίζει δύο διαφορετικούς τύπους δεδομένων: τιμές αληθείας (boolean) και φυσικούς αριθμούς (natural). Επιπλέον τύποι δεδο-

μένων (όπως int, real, char, κλπ) μπορούν να προστεθούν στη γλώσσα χωρίς ιδιαίτερο πρόβλημα. Τα συντακτικά στοιχεία της γλωσσας μπορούν να χωριστούν σε τρεις κατηγορίες: λογικές εκφράσεις (boolean expressions), αριθμητικές εκφράσεις (numerical expressions) και εντολές (commands).

Οι βασικές αριθμητικές εκφράσεις είναι μια σταθερά (`0`) και ένας τελεστής (`succ`) ο οποίος παράγει τον επόμενο φυσικό αριθμό. Οι βασικές λογικές εκφράσεις είναι η σταθερά `true` όπως και εκφράσεις που μπορούν να παραχθούν με τη χρήση τελεστών (πχ. `not`, `and`, κλπ). Γενικά, στη γλώσσα μπορούν να προστεθούν περισσότερα σύμβολα (πχ. `false`, `1`, `2`, κλπ) και περισσότεροι τελεστές, χωρίς πρόβλημα.

Οι κύριες εντολές που υποστηρίζει η γλώσσα είναι: η "κενή" εντολή (`skip`) η οποία δεν έχει κανένα απολύτως αποτέλεσμα, η σύνθεση εντολών που συμβολίζεται με ";", και ένας μηχανισμός βρόχου. Ο βρόχος `for` **N** `do` **C** εκτελείται υπολογίζοντας την τιμή της αριθμητικής έκφρασης **N** και μετά εκτελώντας το σώμα **C** τόσες φορές όσες είναι η τιμή της αριθμητικής έκφρασης. Ο βρόχος αυτός ονομάζεται και *οριστικός* διότι ο αριθμός των επαναλήψεων καθορίζεται πριν αρχίσει η εκτέλεση του.

Αρχικά, η γλώσσα δεν διαθέτει μηχανισμούς αλλαγής της κατάστασης (δηλαδή της μνήμης). Ο μηχανισμός αυτός, που είναι η γνωστή εντολή ανάθεσης (assignment), θα μελετηθεί στην επόμενη παράγραφο.

Το συντακτικό της γλώσσας μπορεί να δοθεί με τη βοήθεια context-free γραμματικής, στην οποία το σύμβολο **C** αναπαριστά μια εντολή, το σύμβολο **B** μιά λογική έκφραση και το **N** μιά αριθμητική έκφραση.

Οι κανόνες παραγωγής γιά τις εντολές είναι οι ακόλουθοι:

$$
\begin{array}{rcl}
\mathbf{C} & ::= & \texttt{skip} \\
& | & \mathbf{C}_0 ; \mathbf{C}_1 \\
& | & \texttt{for } \mathbf{N} \texttt{ do } \mathbf{C} \\
& | & \texttt{if } \mathbf{B} \texttt{ then } \mathbf{C}_0 \texttt{ else } \mathbf{C}_1 \\
& | & (\mathbf{C})
\end{array}
$$

Οι κανόνες παραγωγής γιά τις λογικές εκφράσεις είναι οι ακόλουθοι:

$$
\begin{array}{rcl}
\mathbf{B} & ::= & \texttt{true} \\
& | & \texttt{not } \mathbf{B} \\
& | & \mathbf{B}_0 \texttt{ and } \mathbf{B}_1 \\
& | & \mathbf{N}_0 < \mathbf{N}_1 \\
& | & \mathbf{N}_0 = \mathbf{N}_1 \\
& | & \mathbf{B}_0 = \mathbf{B}_1 \\
& | & \texttt{if } \mathbf{B} \texttt{ then } \mathbf{B}_0 \texttt{ else } \mathbf{B}_1 \\
& | & (\mathbf{B})
\end{array}
$$

Τέλος, οι κανόνες παραγωγής γιά τις αριθμητικές εκφράσεις είναι:

$$\begin{array}{rcl} \mathbf{N} & ::= & \mathtt{0} \\ & | & \mathtt{succ}\ \mathbf{N} \\ & | & \mathtt{if}\ \mathbf{B}\ \mathtt{then}\ \mathbf{N}_0\ \mathtt{else}\ \mathbf{N}_1 \\ & | & (\mathbf{N}) \end{array}$$

4.2.2 Σημασιολογία

Η βασική έννοια που χρησιμοποιείται για τη σημασιολογία των προστακτικών γλωσσών προγραμματισμού είναι αυτή της κατάστασης (state). Η κατάσταση ουσιαστικά αναπαριστά τη μνήμη του υπολογιστή. Οπως γνωρίζουμε, η τιμή μιας έκφρασης σε μιά προστακτική γλώσσα προγραμματισμού εξαρτάται από τις μεταβλητές που υπάρχουν στην έκφραση αυτή, οι τιμές των οποίων φυλάγονται στη μνήμη του υπολογιστή. Επομένως, η σημασιολογία των προστακτικών γλωσσών πρέπει πάντα να λαμβάνει υπ'όψιν της και να χρησιμοποιεί την έννοια της κατάστασης.

Στην παρούσα φάση της, η γλώσσα που ορίσαμε στις προηγούμενες παραγράφους δεν διαθέτει ούτε μεταβλητές αλλά ούτε και εντολές ανάθεσης. Επειδή όμως η γλώσσα αυτή θα επεκταθεί στις επόμενες παραγράφους ώστε να υποστηρίζει και τα χαρακτηριστικά αυτά, και επειδή ήδη έχουμε εισάγει κάποιες (έστω απλές) εντολές, θα δώσουμε ακολούθως τη σημασιολογία της γλώσσας με βάση την έννοια της κατάστασης.

Εστω S λοιπόν ένα σύνολο καταστάσεων (το οποίο προς το παρόν δεν το καθορίζουμε επακριβώς). Διαισθητικά, **το νόημα μιας έκφρασης της γλώσσας είναι μιά συνάρτηση η οποία γιά κάθε κατάσταση του υπολογιστή δίνει την τιμή της έκφρασης σε σχέση με την κατάσταση αυτή. Το νόημα μιας εντολής είναι μια συνάρτηση η οποία μετασχηματίζει την τρέχουσα κατάσταση του υπολογιστή σε μια νέα κατάσταση.**

Για τον ορισμό της σημασιολογίας της γλώσσας θα χρησιμοποιήσουμε τρεις διαφορετικές συναρτήσεις: την συνάρτηση $\mathcal{C}[\![\cdot]\!]$ η οποία δίνει νόημα στις εντολές της γλώσσας. Την συνάρτηση $\mathcal{B}[\![\cdot]\!]$ η οποία δίνει νόημα στις λογικές εκφράσεις, και την συνάρτηση $\mathcal{N}[\![\cdot]\!]$ η οποία δίνει νόημα στις αριθμητικές εκφράσεις. Οι συναρτήσεις αυτές ορίζονται αναδρομικά, και ο ορισμός της κάθε μίας χρησιμοποιεί τις υπόλοιπες.

Η συνάρτηση $\mathcal{B}[\![\cdot]\!]$ ορίζεται ως εξής:

$$\mathcal{B}[\![\texttt{true}]\!]s = true$$

$$\mathcal{B}[\![\texttt{not }\mathbf{B}]\!]s = \begin{cases} true, & if\ \mathcal{B}[\![\mathbf{B}]\!]s = false \\ false, & if\ \mathcal{B}[\![\mathbf{B}]\!]s = true \end{cases}$$

$$\mathcal{B}[\![\mathbf{B}_0 \texttt{ and } \mathbf{B}_1]\!]s = \begin{cases} true, & if\ \mathcal{B}[\![\mathbf{B}_0]\!]s = true\ and\ \mathcal{B}[\![\mathbf{B}_1]\!]s = true \\ false, & otherwise \end{cases}$$

$$\mathcal{B}[\![\mathbf{N}_0 < \mathbf{N}_1]\!]s = \begin{cases} true, & if\ \mathcal{N}[\![\mathbf{N}_0]\!]s < \mathcal{N}[\![\mathbf{N}_1]\!]s \\ false, & otherwise \end{cases}$$

$$\mathcal{B}[\![\mathbf{N}_0 = \mathbf{N}_1]\!]s = \begin{cases} true, & if\ \mathcal{N}[\![\mathbf{N}_0]\!]s = \mathcal{N}[\![\mathbf{N}_1]\!]s \\ false, & otherwise \end{cases}$$

$$\mathcal{B}[\![\mathbf{B}_0 < \mathbf{B}_1]\!]s = \begin{cases} true, & if\ \mathcal{B}[\![\mathbf{B}_0]\!]s < \mathcal{B}[\![\mathbf{B}_1]\!]s \\ false, & otherwise \end{cases}$$

$$\mathcal{B}[\![\texttt{if }\mathbf{B}\texttt{ then }\mathbf{B}_0\texttt{ else }\mathbf{B}_1]\!]s = \begin{cases} \mathcal{B}[\![\mathbf{B}_0]\!]s, & if\ \mathcal{B}[\![\mathbf{B}]\!]s = true \\ \mathcal{B}[\![\mathbf{B}_1]\!]s, & otherwise \end{cases}$$

$$\mathcal{B}[\![(\mathbf{B})]\!]s = \mathcal{B}[\![\mathbf{B}]\!]s$$

Η συνάρτηση $\mathcal{N}[\![\cdot]\!]$ ορίζεται ως εξής:

$$\mathcal{N}[\![\texttt{0}]\!]s = 0$$

$$\mathcal{N}[\![\texttt{succ }\mathbf{N}]\!]s = \mathcal{N}[\![\mathbf{N}]\!]s + 1$$

$$\mathcal{N}[\![\texttt{if }\mathbf{B}\texttt{ then }\mathbf{N}_0\texttt{ else }\mathbf{N}_1]\!]s = \begin{cases} \mathcal{N}[\![\mathbf{N}_0]\!]s, & if\ \mathcal{B}[\![\mathbf{B}]\!]s = true \\ \mathcal{N}[\![\mathbf{N}_1]\!]s, & otherwise \end{cases}$$

$$\mathcal{N}[\![(\mathbf{N})]\!]s = \mathcal{N}[\![\mathbf{N}]\!]s$$

Τέλος, η σημασιολογία των εντολών δίνεται από τη συνάρτηση $\mathcal{N}[\![\cdot]\!]$ η οποία ορίζεται

ως εξής:

$$\begin{array}{lcl}
\mathcal{C}[\![\texttt{skip}]\!]s & = & s \\
\mathcal{C}[\![\mathbf{C}_0;\mathbf{C}_1]\!]s & = & \mathcal{C}[\![\mathbf{C}_1]\!](\mathcal{C}[\![\mathbf{C}_0]\!]s) \\
\mathcal{C}[\![\texttt{for } \mathbf{N} \texttt{ do } \mathbf{C}]\!]s & = & \mathcal{C}[\![\mathbf{C}]\!]^n(s),\ where\ n = \mathcal{N}[\![N]\!]s \\
\mathcal{C}[\![\texttt{if } \mathbf{B} \texttt{ then } \mathbf{C}_0 \texttt{ else } \mathbf{C}_1]\!]s & = & \begin{cases} \mathcal{C}[\![\mathbf{C}_0]\!]s, & if\ \mathcal{B}[\![\mathbf{B}]\!]s = true \\ \mathcal{C}[\![\mathbf{C}_1]\!]s, & otherwise \end{cases} \\
\mathcal{C}[\![(\mathbf{C})]\!]s & = & \mathcal{C}[\![\mathbf{C}]\!]s
\end{array}$$

4.2.3 Παρατηρήσεις

Με βάση την σημασιολογία που ορίσαμε παραπάνω, μπορούμε να αποδείξουμε διάφορες ιδιότητες που ισχύουν γιά τα δομικά στοιχεία της γλώσσας. Μια τεχνική που χρησιμοποιείται πολύ συχνά για το σκοπό αυτό, είναι η λεγόμενη *δομική επαγωγή (structural induction).* Η τεχνική αυτή είναι ουσιαστικά μια εφαρμογή της συνηθισμένης επαγωγής στην συντακτική δομή των εκφράσεων μιας γλώσσας (πιό συγκεκριμένα, στο βάθος του συντακτικού δέντρου των εκφράσεων της γλώσσας). Μια ιδιότητα αποδεικνύεται:

- Για κάθε ένα από τα πρωτόγονα στοιχεία του συντακτικού της γλώσσας.
- Για κάθε σύνθετη έκφραση της γλώσσας, με βάση την υπόθεση οτι οι εκφράσεις που την αποτελούν έχουν την ιδιότητα.

Παράδειγμα 4.1 Με τη χρήση δομικής επαγωγής μπορεί να δειχτεί ότι για τη γλώσσα που έχουμε ορίσει (η οποία δεν περιέχει εντολές ανάθεσης) ισχύει:

$$\mathcal{C}[\![\mathbf{C}]\!]s = s$$

για κάθε εντολή $\mathbf{C}$ και για κάθε κάθε κατάσταση s. Για την επαγωγική βάση αρκεί να εξετάσουμε την περίπτωση $\mathbf{C} = \texttt{skip}$, για την οποία προφανώς το ζητούμενο ισχύει καθώς $\mathcal{C}[\![\texttt{skip}]\!]s = s$. Μένει να δείξουμε οτι το ζητούμενο ισχύει γιά τις πιό σύνθετες εντολές, με βάση την υπόθεση ότι ισχύει γιά όλες τις υποεντολές που τις συνθέτουν. Αυτό απαιτεί να εξετάσουμε τις τέσσερις περιπτώσεις μιας σύνθετης εντολής (δηλαδή τη σύνθεση εντολών, το `for`, το `if` και την παρενθετοποίηση). Θα δείξουμε την περίπτωση της σύνθεσης:

$$\begin{array}{lcll}
\mathcal{C}[\![\mathbf{C}_0;\mathbf{C}_1]\!]s & = & \mathcal{C}[\![\mathbf{C}_1]\!](\mathcal{C}[\![\mathbf{C}_0]\!]s) & (Semantics) \\
& = & \mathcal{C}[\![\mathbf{C}_1]\!](s) & (Hypothesis) \\
& = & s & (Hypothesis)
\end{array}$$

Οι υπόλοιπες περιπτώσεις σύνθετων εντολών μπορούν να δειχτούν με ανάλογο τρόπο. □

4.3 Εντολές Ανάθεσης

Η πιό σημαντική ίσως έλλειψη της γλώσσας που ορίσαμε στις προηγούμενες παραγράφους είναι η απουσία μεθόδων γιά να προσπελάσει και να αλλάξει κανείς την κατάσταση (δηλαδή τη μνήμη). Γιά το σκοπό αυτό πρέπει αρχικά να οριστεί ένα σύνολο *ονομάτων μεταβλητών (variable-identifiers)*. Το σύνολο αυτό το συμβολίζουμε με Var, και περιέχει στοιχεία όπως `i`, `j`, κλπ. Οι μεταβλητές που ανήκουν στο Var χωρίζονται σε δύο υποσύνολα: τις λογικές (boolean) και τις αριθμητικές (natural). Για το σκοπό αυτό υποθέτουμε την ύπαρξη μιας συνάρτησης π, η οποία προσδιορίζει τον τύπο της κάθε μεταβλητής. Για παράδειγμα, αν `i` είναι μια λογική μεταβλητή, τότε $\pi(\texttt{i}) = boolean$, ενώ αν `j` είναι μια αριθμητική μεταβλητή, τότε $\pi(\texttt{j}) = natural$.

Το συντακτικό της γλώσσας μπορεί να επαυξηθεί ώστε να επιτρέπει τη χρήση μεταβλητών αλλά και εντολών ανάθεσης. Πιο συγκεκριμένα, επεκτείνουμε τους κανόνες παραγωγής των εκφράσεων και των εντολών της γλώσσας, ως εξής:

$$\begin{array}{lll} \mathbf{B} & ::= & \ldots \\ & | & \mathbf{i},\ where\ \mathbf{i} \in Var,\ \pi(\mathbf{i}) = boolean \end{array}$$

$$\begin{array}{lll} \mathbf{N} & ::= & \ldots \\ & | & \mathbf{i},\ where\ \mathbf{i} \in Var,\ \pi(\mathbf{i}) = natural \end{array}$$

$$\begin{array}{lll} \mathbf{C} & ::= & \ldots \\ & | & \mathbf{i} := \mathbf{B},\ \pi(\mathbf{i}) = boolean \\ & | & \mathbf{i} := \mathbf{N},\ \pi(\mathbf{i}) = natural \end{array}$$

Για να περιγράψουμε τη σημασιολογία των εντολών ανάθεσης καθώς και των μεταβλητών (οι οποίες μπορούν να χρησιμοποιηθούν στις εκφράσεις της γλώσσας), χρειάζεται κατ'αρχήν να ορίσουμε μαθηματικά το σύνολο των καταστάσεων S. Χρειάζεται δηλαδή να ορίσουμε μαθηματικά την έννοια της μνήμης του υπολογιστή. Διαισθητικά, μια κατάσταση $s \in S$ είναι μια συνάρτηση η οποία αντιστοιχίζει σε κάθε μεταβλητή της γλώσσας μια τιμή. Η συνάρτηση αυτή s πρέπει να "σέβεται" τον τύπο της κάθε μεταβλητής. Ετσι η s αντιστοιχίζει σε λογικές μεταβλητές λογικές τιμές, ενώ στις αριθμητικές μεταβλητές εκχωρεί αριθμητικές τιμές. Με βάση τις παρατηρήσεις αυτές, το σύνολο S των καταστάσεων μπορεί να οριστεί ως εξής:

$$\begin{array}{ll} S = \{ & s : Var \to Nat \cup \{true, false\} \mid \\ & s(\mathbf{i}) \in Nat \quad if \quad \pi(\mathbf{i}) = natural \quad and \\ & s(\mathbf{i}) \in \{true, false\} \quad if \quad \pi(\mathbf{i}) = boolean \quad \} \end{array}$$

Η σημασιολογία της γλώσσας μπορεί τώρα να επεκταθεί ώστε να καλύψει και την περίπτωση των μεταβλητών και των εντολών ανάθεσης. Στήν δεύτερη από τις παρακάτω εξι-

σώσεις, χρησιμοποιούμε το σύμβολο $\mathbf{E}$ για να αναπαραστήσουμε μια οποιαδήποτε έκφραση της γλώσσας (λογική είτε αριθμητική).

$$\begin{aligned} [\![\mathbf{i}]\!]s &= s(\mathbf{i}) \\ [\![\mathbf{i} := \mathbf{E}]\!]s &= (s \mid \mathbf{i} \mapsto [\![\mathbf{E}]\!]s) \end{aligned}$$

όπου $(s \mid \mathbf{i} \mapsto [\![\mathbf{E}]\!]s)$ είναι μια νέα κατάσταση s' έτσι ώστε $s'(\mathbf{i}) = [\![\mathbf{E}]\!]s$ και για κάθε άλλη μεταβλητή $\mathbf{i}' \neq \mathbf{i}$, $s'(\mathbf{i}') = s(\mathbf{i})$. Διαισθητικά, η δεύτερη εξίσωση λέει ότι η εκτέλεση μιας εντολής ανάθεσης σε μια δεδομένη κατάσταση έχει ως αποτέλεσμα την παραγωγή μιας "νέας" κατάστασης στην οποία έχει αλλάξει μόνο η τιμή της μεταβλητής η οποία σχετίζεται με την ανάθεση. Η νέα τιμή της μεταβλητής αυτής είναι η τιμή του δεξιού μέλους της εντολής ανάθεσης υπολογισμένη στην "παλαιά" κατάσταση.

Με τη χρήση των παραπάνω εξισώσεων μπορεί κανείς να αποδείξει την ισοδυναμία διαφόρων εντολών της γλώσσας. Για παράδειγμα, η εντολή:

$$\mathbf{i} := (\texttt{if } \mathbf{B} \texttt{ then } \mathbf{E}_0 \texttt{ else } \mathbf{E}_1)$$

και η εντολή

$$\texttt{if } \mathbf{B} \texttt{ then } (\mathbf{i} := \mathbf{E}_0) \texttt{ else } (\mathbf{i} := \mathbf{E}_1)$$

μπορούν εύκολα να αποδειχτούν ισοδύναμες με τη χρήση των παραπάνω σημασιολογικών εξισώσεων.

Κεφάλαιο 5

Αξιωματική Σημασιολογία και Απόδειξη Ορθότητας Προγραμμάτων

Στο κεφάλαιο αυτό θα εισάγουμε τις βασικές έννοιες της Αξιωματικής Σημασιολογίας και θα παρουσιάσουμε τις εφαρμογές της στην απόδειξη ορθότητας προγραμμάτων. Η τεχνική που θα περιγραφεί παρακάτω, προτάθηκε αρχικά από τους Floyd και Hoare, και για το λόγο αυτό θα την αναφέρουμε στη συνέχεια ως *Τεχνική (ή Λογική)* των Floyd-Hoare.

Η γλώσσα προγραμματισμού που θα χρησιμοποιήσουμε στη συνέχεια είναι μια ιδιαίτερα απλή προστακτική γλώσσα. Σκοπός μας είναι η παρουσίαση μιας τεχνικής που θα μας επιτρέπει να αποδείξουμε προτάσεις που αφορούν τη συμπεριφορά των προγραμμάτων της γλώσσας αυτής.

5.1 Η Γλώσσα Προγραμματισμού

Τα προγράμματα της γλώσσας προγραμματισμού που θα χρησιμοποιήσουμε στη συνέχεια του κεφαλαίου αυτού αποτελούνται από εντολές (όπως εντολές ανάθεσης, επανάληψης, κλπ). Βασικός σκοπός μας είναι να μελετήσουμε την τεχνική των Floyd-Hoare για τα πολύ απλά χαρακτηριστικά των προστακτικών γλωσσών, οπότε κατά συνέπεια, το συντακτικό της γλώσσας που υιοθετούμε είναι αρκετά περιορισμένο. Αξίζει στο σημείο αυτό να σημειωθεί οτι η τεχνική που εξετάζουμε μπορεί να επεκταθεί και να χρησιμοποιηθεί και για πιο "πραγματικές" γλώσσες προγραμματισμού. Το συντακτικό της γλώσσας που θα εξετάσουμε στη

συνέχεια του κεφαλαίου, είναι το ακόλουθο:

$$
\begin{array}{rcl}
\mathbf{C} & ::= & \texttt{skip} \\
& | & (\mathbf{C}) \\
& | & \texttt{i} := \mathbf{E} \\
& | & \mathbf{C}_0 ; \mathbf{C}_1 \\
& | & \texttt{while } \mathbf{B} \texttt{ do } \mathbf{C} \\
& | & \texttt{if } \mathbf{B} \texttt{ then } \mathbf{C}_0 \texttt{ else } \mathbf{C}_1
\end{array}
$$

Η σημασιολογία των χαρακτηριστικών της παραπάνω γλώσσας είναι (διαισθητικά) αυτή που ισχύει για τη σημασιολογία αντίστοιχων χαρακτηριστικών που υπάρχουν σε κλασικές προστακτικές γλώσσες προγραμματισμού (πχ. Pascal, C, κλπ.).

5.2 Η Τεχνική των Floyd-Hoare

Ο C.A.R. Hoare εισήγαγε τον ακόλουθο συμβολισμό για τον καθορισμό του τι ακριβώς κάνει ένα πρόγραμμα:

$$\{\texttt{P}\}\texttt{C}\{\texttt{Q}\}$$

όπου:

- C είναι ένα πρόγραμμα της γλώσσας που μελετάμε
- P και Q είναι συνθήκες που (γενικά) σχετίζονται με τις μεταβλητές που χρησιμοποιεί το πρόγραμμα C.

Οι συνθήκες γενικά αποτελούν εκφράσεις μιας απλής λογικής γλώσσας (θα περιέχουν δηλαδή μεταβλητές του προγράμματος, σταθερές, καθώς επίσης και λογικούς τελεστές όπως $\wedge$, $\vee$, $\neg$, κλπ.). Μια συνθήκη που προηγείται μιας εντολής περιγράφει τους περιορισμούς που ισχύουν σχετικά με τις μεταβλητές του προγράμματος στο σημείο αυτό. Αντίστοιχα, μια συνθήκη που ακολουθεί μια εντολή περιγράφει τους νέους περιορισμούς που έχουν προκύψει σε σχέση με τις μεταβλητές του προγράμματος, όταν η εντολή έχει εκτελεστεί.

Τις εκφράσεις της μορφής {P}C{Q} θα τις ονομάζουμε *προδιαγραφές (specifications)*. Ο παρακάτω ορισμός καθορίζει πότε η προδιαγραφή {P}C{Q} είναι αληθής:

Ορισμός 5.1 Θα λέμε ότι η προδιαγραφή {P}C{Q} είναι αληθής εάν όταν το C εκτελείται σε μιά κατάσταση που ικανοποιεί τη συνθήκη P και αν η εκτέλεση του C τερματίζει, τότε στην κατάσταση στην οποία καταλήγει το πρόγραμμα, ικανοποιείται η συνθήκη Q.

Αξίζει να σημειωθεί ότι οι προδιαγραφές ονομάζονται συνήθως και *τύποι μερικής ορθότητας* (*partial correctness formulas*) διότι η απόδειξη του ότι ένα πρόγραμμα τερματίζει θεωρείται

ως μια ξεχωριστή διαδικασία που πρέπει να εξετάσει ο προγραμματιστής ώστε να είναι απόλυτα βέβαιος για την ορθότητα του προγράμματος του.

Παράδειγμα 5.1 Η προδιαγραφή:

$$\{\texttt{X=1}\}\texttt{X:=X+1}\{\texttt{X=2}\}$$

είναι προφανώς αληθής. Το σύμβολο `=` που εμφανίζεται στις συνθήκες, είναι η γνωστή μας από τα μαθηματικά ισότητα. □

Παράδειγμα 5.2 Η προδιαγραφή:

$$\{\texttt{X=1}\}\texttt{Y:=X}\{\texttt{Y=1}\}$$

είναι εμφανώς αληθής. □

Παράδειγμα 5.3 Η προδιαγραφή:

$$\{\texttt{X=1}\}\texttt{Y:=X+1}\{\texttt{X=2}\}$$

είναι ψευδής. □

Παράδειγμα 5.4 Η προδιαγραφή:

$$\{\texttt{X=x}\wedge\texttt{Y=y}\}\texttt{R:=X; X:=Y; Y:=R}\{\texttt{X=y} \wedge \texttt{Y=x}\}$$

είναι αληθής. Οι μεταβλητές `x` και `y` που εμφανίζονται στις συνθήκες αλλά όχι στην εντολή, ονομάζονται *βοηθητικές μεταβλητές*, και σκοπός τους είναι να δώσουν όνομα στις αρχικές τιμές των μεταβλητών `X` και `Y` αντίστοιχα. □

Παράδειγμα 5.5 Η προδιαγραφή:

$$\{\texttt{true}\}\texttt{C}\{\texttt{Q}\}$$

είναι αληθής εάν όταν το πρόγραμμα `C` τερματίζει, ισχύει η συνθήκη `Q`. □

Παράδειγμα 5.6 Η προδιαγραφή:

$$\{\texttt{P}\}\texttt{C}\{\texttt{true}\}$$

είναι πάντα αληθής για κάθε αρχική συνθήκη `P` και κάθε εντολή `C`. □

Η τεχνική των Floyd-Hoare επιτρέπει τη δημιουργία τυπικών αποδείξεων για προδιαγραφές προγραμμάτων. Μια απόδειξη στην τεχνική των Floyd-Hoare παράγεται χρησιμοποιώντας τα αξιώματα και τους κανόνες εξαγωγής συμπερασμάτων που υποστηρίζει η τεχνική αυτή. Η απόδειξη μπορεί να χρησιμοποιεί και θεωρήματα των κλασικών μαθηματικών όταν

αυτό είναι αναγκαίο. Στις επόμενες παραγράφους θα παρουσιάσουμε τα βασικά αξιώματα και τους κανόνες εξαγωγής συμπερασμάτων της τεχνικής των Floyd-Hoare. Η βασική διαφορά ανάμέσα στα αξιώματα και στους κανόνες μπορεί διαισθητικά να περιγραφεί ως εξής: ένα αξίωμα είναι μια λογική πρόταση για την οποία υποθέτουμε ότι είναι αληθής. Ένας κανόνας εξαγωγής συμπερασμάτων είναι μια μέθοδος που μπορεί να χρησιμοποιηθεί για να δείξουμε ότι μια προδιαγραφή είναι αληθής χρησιμοποιώντας σαν υπόθεση ότι κάποιες άλλες προδιαγραφές είναι αληθείς.

5.3 Το Αξίωμα του skip και ο Κανόνας των Παρενθέσεων

Το αξίωμα αυτό καθώς και ο κανόνας είναι τα πλέον τετριμμένα της τεχνικής των Floyd-Hoare. Το αξίωμα του `skip` είναι:

$$\{\texttt{P}\}\texttt{skip}\{\texttt{P}\}$$

Με άλλα λόγια, η εντολή `skip` δε μεταβάλλει οποιαδήποτε συνθήκη ισχύει πριν από την εκτέλεση της.

Ο κανόνας των Παρενθέσεων λέει ότι εάν ισχύει το $\{\texttt{P}\}\texttt{C}\{\texttt{R}\}$ τότε μπορούμε να εξάγουμε ως συμπέρασμα και το $\{\texttt{P}\}\texttt{(C)}\{\texttt{R}\}$.

5.4 Το Αξίωμα της Εντολής Ανάθεσης

Το αξίωμα της εντολής ανάθεσης αναπαριστά το γεγονός ότι η τιμή μιας μεταβλητής `V` μετά από την εκτέλεση της εντολής ανάθεσης `V:=E`, ισούται με την τιμή της έκφρασης `E` στην κατάσταση πριν την εκτέλεση της εντολής. Το αξίωμα αυτό είναι:

$$\{\texttt{P[E/V]}\}\texttt{V:=E}\{\texttt{P}\}$$

όπου `V` είναι μια μεταβλητή, `E` μια έκφραση, `P` μια συνθήκη και `P[E/V]` είναι το αποτέλεσμα της αντικατάστασης όλων των εμφανίσεων της μεταβλητής `V` στην `P` με `E`.

Παράδειγμα 5.7 Τα ακόλουθα είναι παραδείγματα εφαρμογής του αξιώματος της εντολής ανάθεσης:

$$\{\texttt{X+1=n+1}\}\texttt{X:=X+1}\{\texttt{X=n+1}\}$$

όπως και:

$$\{\texttt{E=E}\}\texttt{X:=E}\{\texttt{X=E}\}$$

αν το X δεν εμφανίζεται στο E. □

Το αξίωμα της ανάθεσης μοιάζει περίεργο με μια πρώτη ματιά, και θα πίστευε κανείς οτι θα έπρεπε να έχει τη μορφή:

$$\{P\}V:=E\{P[E/V]\}$$

Κάτι τέτοιο είναι όμως λανθασμένο διότι θα μας έδινε προδιαγραφές όπως η παρακάτω:

$$\{X=0\}X:=1\{1=0\}$$

Αξίζει να σημειωθεί ότι το (σωστό) αξίωμα της εντολής ανάθεσης που παραθέσαμε παραπάνω, δεν ισχύει (σε αυτή τουλάχιστον τη μορφή) όταν κανείς εξετάζει πιό πολύπλοκες προστακτικές γλώσσες προγραμματισμού που υποστηρίζουν πιο ισχυρά χαρακτηριστικά.

5.5 Ο Κανόνας της Ενδυνάμωσης της Προσυνθήκης

Ο κανόνας αυτός λέει ότι εάν ισχύει το $P \rightarrow R$ καθώς και το $\{R\}C\{Q\}$ τότε μπορούμε να εξάγουμε ως συμπέρασμα και το $\{P\}C\{Q\}$.

Παράδειγμα 5.8 Γνωρίζουμε από προηγούμενο παράδειγμα ότι:

$$\{X+1=n+1\}X:=X+1\{X=n+1\}$$

Επίσης, από τα κλασικά μαθηματικά γνωρίζουμε ότι:

$$(X+1=n+1) \rightarrow (X=n)$$

Επομένως, με βάση τον κανόνα της Ενδυνάμωσης της Προσυνθήκης, μπορούμε να εξάγουμε ως συμπέρασμα και ότι:

$$\{X=n\}X:=X+1\{X=n+1\}$$

Όπως έχουμε προαναφέρει, μεταβλητές όπως η n στο παράδειγμα αυτό, ονομάζονται βοηθητικές μεταβλητές, και χρησιμοποιούνται για να συσχετίσουν τιμές στην κατάσταση πριν και μετά την εκτέλεση μιας εντολής. □

5.6 Ο Κανόνας της Αποδυνάμωσης της Μετασυνθήκης

Ο κανόνας αυτός λέει ότι εάν ισχύει το $\{P\}C\{R\}$ καθώς και το $R \rightarrow Q$ τότε μπορούμε να εξάγουμε ως συμπέρασμα και το $\{P\}C\{Q\}$.

Παράδειγμα 5.9 Παραθέτουμε μια απόδειξη με βάση τους κανόνες που έχουμε ήδη παρουσιάσει. Χρησιμοποιώντας το αξίωμα της εντολής ανάθεσης, έχουμε:

$$\{\texttt{R=X} \ \wedge \ \texttt{0=0}\}\texttt{Q:=0}\{\texttt{R=X} \ \wedge \ \texttt{Q=0}\}$$

Από τα κλασικά μαθηματικά γνωρίζουμε ότι:

$$(\texttt{R=X}) \ \rightarrow \ (\texttt{R=X} \ \wedge \ \texttt{0=0})$$

Χρησιμοποιώντας τον κανόνα της Ενδυνάμωσης της Προσυνθήκης, έχουμε:

$$\{\texttt{R=X}\}\texttt{Q:=0}\{\texttt{R=X} \ \wedge \ \texttt{Q=0}\}$$

Και πάλι, χρησιμοποιώντας απλά μαθηματικά, έχουμε:

$$(\texttt{R=X} \ \wedge \ \texttt{Q=0}) \ \rightarrow \ (\texttt{R=X+Y}\times\texttt{Q})$$

Τέλος, με τη χρήση του κανόνα της Αποδυνάμωσης της Μετασυνθήκης, έχουμε:

$$\{\texttt{R=X}\}\texttt{Q:=0}\{\texttt{R=X+Y}\times\texttt{Q}\}$$

□

Οι κανόνες της Ενδυνάμωσης και της Αποδυνάμωσης ονομάζονται συχνά και κανόνες της Συνεπαγωγής.

5.7 Οι κανόνες της Σύζευξης και της Διάζευξης

Ο κανόνας της Σύζευξης καθορίζει ότι εάν ισχύει το $\{\texttt{P}_1\}\texttt{C}\{\texttt{Q}_1\}$ καθώς και το $\{\texttt{P}_2\}\texttt{C}\{\texttt{Q}_2\}$ τότε μπορούμε να εξάγουμε ως συμπέρασμα και το $\{\texttt{P}_1 \ \wedge \ \texttt{P}_2\}\texttt{C}\{\texttt{Q}_1 \ \wedge \ \texttt{Q}_2\}$.

Παρομοίως, ο κανόνας της Διάζευξης καθορίζει ότι εάν ισχύει το $\{\texttt{P}_1\}\texttt{C}\{\texttt{Q}_1\}$ καθώς και το $\{\texttt{P}_2\}\texttt{C}\{\texttt{Q}_2\}$ τότε μπορούμε να εξάγουμε ως συμπέρασμα και το $\{\texttt{P}_1 \ \vee \ \texttt{P}_2\}\texttt{C}\{\texttt{Q}_1 \ \vee \ \texttt{Q}_2\}$.

Οι παραπάνω κανόνες επιτρέπουν να χωριστεί μια απόδειξη σε άλλες ανεξάρτητες αποδείξεις. Έτσι, για παράδειγμα, μπορεί κανείς να αποδείξει το $\{\texttt{P}\}\texttt{C}\{\texttt{Q}_1 \ \wedge \ \texttt{Q}_2\}$, αποδεικνύωντας χωριστά τα $\{\texttt{P}\}\texttt{C}\{\texttt{Q}_1\}$ και $\{\texttt{P}\}\texttt{C}\{\texttt{Q}_1\}$.

5.8 Ο Κανόνας των Σύνθετων Εντολών

Ο κανόνας αυτός αφορά σύνθετες εντολές της μορφής $\texttt{C}_1\texttt{;C}_2$. Πιο συγκεκριμένα, ο κανόνας αυτός καθορίζει ότι εάν ισχύουν τα $\{\texttt{P}\}\texttt{C}_1\{\texttt{Q}\}$ και $\{\texttt{Q}\}\texttt{C}_2\{\texttt{R}\}$ μπορούμε να εξάγουμε ως συμπέρασμα και το $\{\texttt{P}\}\texttt{C}_1\texttt{;C}_2\{\texttt{R}\}$.

Παράδειγμα 5.10 Με τη χρήση του αξιώματος της εντολής ανάθεσης, μπορούμε εύκολα να εξάγουμε οτι τα ακόλουθα ισχύουν:

$$\{\texttt{X=x} \ \wedge \ \texttt{Y=y}\}\texttt{R:=X}\{\texttt{R=x} \ \wedge \ \texttt{Y=x}\}$$
$$\{\texttt{R=x} \ \wedge \ \texttt{Y=y}\}\texttt{X:=Y}\{\texttt{R=x} \ \wedge \ \texttt{X=y}\}$$
$$\{\texttt{R=x} \ \wedge \ \texttt{X=y}\}\texttt{Y:=R}\{\texttt{Y=x} \ \wedge \ \texttt{X=y}\}$$

Από τα δύο πρώτα και με βάση τον κανόνα των Σύνθετων Εντολών, μπορούμε να εξάγουμε οτι ισχύει το ακόλουθο:

$$\{\texttt{X=x} \wedge \texttt{Y=y}\}\texttt{R:=X;X:=Y}\{\texttt{R=x} \wedge \texttt{X=y}\}$$

Τα παραπάνω εύκολα δίνουν:

$$\{\texttt{X=x} \wedge \texttt{Y=y}\}\texttt{R:=X;X:=Y;Y:=R}\{\texttt{Y=x} \wedge \texttt{X=y}\}$$

□

Μπορεί κανείς εύκολα να διατυπώσει έναν κανόνα που αφορά τη σύνθεση περισσότερων από δύο εντολών.

5.9 Ο Κανόνας του if

Ο κανόνας του `if` καθορίζει ότι εάν ισχύει το $\{\texttt{P} \wedge \texttt{S}\}\texttt{C}_1\{\texttt{Q}\}$ καθώς και το $\{\texttt{P} \wedge \neg\texttt{S}\}\texttt{C}_2\{\texttt{Q}\}$ τότε μπορούμε να εξάγουμε ως συμπέρασμα το $\{\texttt{P}\}\texttt{if S then C}_1\texttt{ else C}_2\{\texttt{Q}\}$.

Παράδειγμα 5.11 Μπορεί να δειχτεί χρσιμοποιώντας τον κανόνα του `if` καθώς και άλλους προηγούμενους κανόνες και αξιώματα, ότι η προδαγραφή:

$$\{\texttt{y>1}\}\texttt{if (x>0) then (y := y-1) else (y := y+1) }\{\texttt{y>0}\}$$

είναι αληθής. □

5.10 Ο Κανόνας του while

Ο κανόνας που αφορά την εντολή `while` είναι ο ακόλουθος: αν γνωρίζουμε ότι $\{\texttt{P} \wedge \texttt{S}\}\texttt{C}\{\texttt{P}\}$ τότε μπορούμε να εξάγουμε ως συμπέρασμα και ότι $\{\texttt{P}\}\texttt{while S do C}\{\texttt{P}\wedge \neg\texttt{S}\}$.

Η συνθήκη `P` στον παραπάνω κανόνα ονομάζεται *αμετάβλητη συνθήκη (invariant)* διότι εξακολουθεί να ισχύει και μετά το τέλος της εκτέλεσης του `while`. Ο κανόνας του `while` λέει ότι αν η `P` είναι μια αμετάβλητη συνθήκη του σώματος μιας εντολής `while` (όταν ισχύει και η συνθήκη `S`) τότε η `P` είναι μία αμετάβλητη συνθήκη ολόκληρης της εντολής `while`. Με άλλα λόγια, αν η εκτέλεση της `C` μια φορά διατηρεί την αλήθεια της `P`, τότε η εκτέλεση της `C` για οποιονδήποτε αριθμό επαναλήψεων διατηρεί την αλήθεια της `P`. Επομένως, για να εφαρμόσει κανείς τον κανόνα του `while` θα πρέπει πρώτα να βρεί μια κατάλληλη αμετάβλητη συνθήκη.

Παράδειγμα 5.12 Έστω ότι θέλουμε να αποδείξουμε:

```
{true}
R:=X;
Q:=0;
while Y<=R do (R:=R-Y;Q:=Q+1)
{R<Y ∧ X=R+(Y×Q)}
```

Με άλλα λόγια, δεδομένων των αρχικών τιμών των μεταβλητών `X` και `Y`, ο παραπάνω αλγόριθμος υπολογίζει το πηλίκο `Y` και το υπόλοιπο `R` της διαίρεσης του `X` με το `Y`.

Μπορούμε να χωρίσουμε την απόδειξη σε δύο τμήματα. Κατ'αρχήν πρέπει να δείξουμε ότι:

```
{true}R:=X;Q:=0{X=R+(Y×Q)}
```

και μετά ότι:

```
{X=R+(Y×Q)}
while Y<=R do (R:=R-Y;Q:=Q+1)
{X=R+(Y×Q) ∧ ¬(Y≤R)}
```

Η πρώτη από τις δύο παραπάνω προδιαγραφές αποδεικνύεται εύκολα (με τη χρήση προηγούμενων κανόνων και αξιωμάτων). Εξετάζουμε τώρα τη δεύτερη προδιαγραφή. Θα δείξουμε πρώτα το εξής:

```
{X=R+(Y×Q)}R:=R-Y;Q:=Q+1{X=R+(Y×Q)}
```

Αυτό όμως μπορεί εύκολα να δειχτεί διότι ισχύει με βάση το αξίωμα της εντολής ανάθεσης ότι:

```
{X=(R-Y)+Y+(Y×Q)}R:=R-Y{X=R+Y+(Y×Q)}
```

και επίσης (με βάση το ίδιο αξίωμα):

```
{X=R+(Y×(Q+1))}Q:=Q+1{X=R+(Y×Q)}
```

Χρησιμοποιώντας τον Κανόνα των σύνθετων εντολών έχουμε:

```
{X=R+(Y×Q)}R:=R-Y;Q:=Q+1{X=R+(Y×Q)}
```

Με τη χρήση του Κανόνα της Ενδυνάμωσης της Προσυνθήκης παίρνουμε:

```
{(X=R+(Y×Q)) ∧ (Y≤R)}R:=R-Y;Q:=Q+1{X=R+(Y×Q)}
```

Μπορούμε τώρα να εφαρμόσουμε τον Κανόνα του `while` και να πάρουμε το ζητούμενο αποτέλεσμα. □

Η τεχνική που περιγράψαμε έχει επεκταθεί και εφαρμοστεί σε πολλά προχωρημένα χαρακτηριστικά των γλωσσών προγραμματισμού (πχ. συναρτήσεις, διαδικασίες, εντολές `for`, `goto`, κλπ.). Έχει επίσης επεκταθεί ακόμη και σε παράλληλες γλώσσες προγραμματισμού. Η τεχνική έχει μάλιστα χρησιμοποιηθεί για να αποδειχτεί η ορθότητα μη τετριμμένων παραλλήλων αλγορίθμων των οποίων η ορθότητα δεν ήταν διαισθητικά προφανής.

Κεφάλαιο 6

Υλοποίηση Γλωσσών Προγραμματισμού

Στο κεφάλαιο αυτό θα μελετήσουμε τη βασική δομή της υλοποίησης των γλωσσών προγραμματισμού. Πιο συγκεκριμένα, θα περιγράψουμε τη δομή ενός μεταγλωττιστή, δίνοντας ιδιαίτερη έμφαση στην υλοποίηση διαδικασιών (procedures).

6.1 Μεταγλωττιστές

Ένας μεταγλωττιστής (compiler) είναι ένα πρόγραμμα το οποίο διαβάζει προγράμματα μιας δεδομένης γλώσσας (πηγαία γλώσσα - source language) και τα μετατρέπει σε ισοδύναμα προγράμματα μιας άλλης γλώσσας (γλώσσα στόχος - target language). Επιπλέον, ο μεταγλωττιστής αναφέρει στον προγραμματιστή τυχόν λάθη που υπάρχουν στο πηγαίο πρόγραμμα.

Ένας μεταγλωττιστής εκτελεί δύο βασικές διεργασίες: την *ανάλυση* του πηγαίου προγράμματος και τη *σύνθεση* του τελικού προγράμματος-στόχος. Όλοι σχεδόν οι σύγχρονοι μεταγλωττιστές είναι όπως λέμε *κατευθυνόμενοι από το συντακτικό (syntax-directed).* Αυτό σημαίνει ότι η διαδικασία της μεταγλώττισης ενός πηγαίου προγράμματος καθορίζεται σχεδόν αποκλειστικά από τη συντακτική δομή του προγράμματος αυτού.

Ένας μεταγλωττιστής αποτελεί συνήθως ένα μεγάλο και πολύπλοκο πρόγραμμα. Για το λόγο αυτό, η υλοποίηση του συνήθως χωρίζεται σε φάσεις, οι οποίες και περιγράφονται στη συνέχεια.

6.1.1 Λεκτική Ανάλυση

Στη φάση αυτή ο μεταγλωττιστής διαβάζει το πηγαίο πρόγραμμα χαρακτήρα πρός χαρακτήρα. Το τμήμα του μεταγλωττιστή που επιτελεί τη λειτουργία αυτή ονομάζεται "λεκτικός

αναλυτής". Κατά τη διάρκεια της λεκτικής ανάλυσης, οι χαρακτήρες ομαδοποιούνται σε ανεξάρτητες οντότητες (tokens), όπως π.χ. λέξεις-κλειδιά, μεταβλητές, αριθμοί, κλπ. Έστω για παράδειγμα ότι ο λεκτικός αναλυτής διαβάζει την εντολή:

```
height := base + old*10;
```

Η παραπάνω εντολή θα χωριστεί στις οντότητες "`height`", "`:=`", "`base`", "`+`", "`old`", "`*`", "`10`", και "`;`". Πέρα από το διαχωρισμό του πηγαίου προγράμματος σε ένα σύνολο από οντότητες όπως οι παραπάνω, ο λεκτικός αναλυτής πραγματοποιεί και μερικές άλλες αρχικές διεργασίες που διευκολύνουν τις επόμενες φάσεις της μεταγλώττισης (για παράδειγμα, αφαίρεση σχολίων από το πηγαίο πρόγραμμα).

6.1.2 Συντακτική Ανάλυση

Η ακολουθία από ανεξάρτητες οντότητες που παράγεται από το λεκτικό αναλυτή, μεταβιβάζεται στον λεγόμενο *συντακτικό αναλυτή* (*syntax analyzer ή parser*). Ο συντακτικός αναλυτής ελέγχει αν η ακολουθία από ανεξάρτητες οντότητες σχηματίζει συντακτικά ορθές φράσεις της πηγαίας γλώσσας. Η πληροφορία που διαθέτει ο συντακτικός αναλυτής για το σκοπό αυτό είναι ένα σύνολο κανόνων που περιγράφουν το συντακτικό της πηγαίας γλώσσας και οι οποίοι έχουν συνήθως τη μορφή μιας επεκτεταμένης γραμματικής χωρίς συμφραζόμενα.

Κατά τη διάρκεια της συντακτικής ανάλυσης, ελέγχεται αν το συντακτικό του προγράμματος είναι ορθό. Σε περίπτωση που εντοπιστεί κάποιο λάθος, εξάγεται κάποιο κατάλληλο διαγνωστικό μήνυμα. Σε πολλές περιπτώσεις λάθους, ο συντακτικός αναλυτής καταφέρνει να ξεπεράσει το σφάλμα και να συνεχίσει τη διαδικασία της συντακτικής ανάλυσης. Στο τέλος της διαδικασίας, ο αναλυτής έχει σχηματίσει ένα *συντακτικό δέντρο (parse tree)* το οποίο και χρησιμοποιείται στην επόμενη φάση της μεταγλώττισης.

6.1.3 Σημασιολογική Ανάλυση

Στη φάση αυτή ελέγχεται η *στατική σημασιολογία (static semantics)* κάθε φράσης του πηγαίου προγράμματος. Για παράδειγμα, δεδομένης μιας εντολής ανάθεσης, ελέγχεται αν όλες οι μεταβλητές που χρησιμοποιούνται έχουν δηλωθεί, αν οι τύποι των μεταβλητών είναι σωστοί, κλπ. Στην περίπτωση που όλα είναι σωστά, λαμβάνει χώρα και μετάφραση, παράγεται δηλαδή όπως λέμε *ενδιάμεσος κώδικας (intermediate code)* ο οποίος υλοποιεί τη συγκεκριμένη εντολή. Συνήθως, ο έλεγχος της στατικής σημασιολογίας και η παραγωγή του ενδιάμεσου κώδικα λαμβάνουν χώρα ανεξάρτητα.

6.1.4 Βελτιστοποίηση

Ο ενδιάμεσος κώδικας που παράγεται κατά τη φάση της σημασιολογικής ανάλυσης, μετασχηματίζεται σε ισοδύναμο αλλά περισσότερο αποδοτικό κώδικα. Η φάση αυτή μπορεί να είναι πολύπλοκη και αργή. Συνήθως αποτελείται από υπο-φάσεις, μερικές από τις οποίες πρέπει να επαναληφθούν περισσότερες από μία φορές. Οι περισσότεροι μεταγλωττιστές επιτρέπουν στον προγραμματιστή να απενεργοποιήσει τη φάση των βελτιστοποιήσεων. Μερικά παραδείγματα απλών βελτιστοποιήσεων είναι η διαγραφή πολλαπλασιασμών με 1 ή προσθέσεων με 0, η αντικατάσταση μιας ακολουθίας εντολών από άλλες που έχουν το ίδιο ακριβώς αποτέλεσμα αλλά είναι πιό αποτελεσματικές, κλπ.

6.1.5 Παραγωγή Τελικού Κώδικα

Ο ενδιάμεσος κώδικας μετατρέπεται σε εκτελέσιμο κώδικα μηχανής κατά τη φάση αυτή. Η παραγωγή τελικού κώδικα απαιτεί τη λεπτομερή γνώση των χαρακτηριστικών της μηχανής στην οποία θα εκτελεστεί ο κώδικας-στόχος (π.χ. γνώση του συνόλου των εντολών της συγκεκριμένης γλώσσας μηχανής, των καταχωρητών που διαθέτει η μηχανή, κλπ.). Η παραγωγή τελικού κώδικα είναι μια ιδιαίτερα επίπονη φάση καθώς απαιτεί την εξέταση και το χειρισμό πολλών ειδικών περιπτώσεων. Ένας τρόπος για την απλούστευση της δομής ενός μεταγλωττιστή είναι η συνένωση των φάσεων της παραγωγής του ενδιάμεσου και του τελικού κώδικα.

Στην επόμενη παράγραφο θα εξετάσουμε ένα σημαντικό χαρακτηριστικό των σύγχρονων γλωσσών προγραμματισμού, τις διαδικασίες, των οποίων η υλοποίηση αποτελεί ένα από τα πιό δύσκολα σημεία ενός μεταγλωττιστή.

6.2 Διαδικασίες

Οι διαδικασίες (procedures) είναι ένα χαρακτηριστικό των σύγχρονων γλωσσών προγραμματισμού που μας επιτρέπει να δίνουμε όνομα σε ένα τμήμα κώδικα και να το "καλούμε" από άλλα σημεία του προγράμματος. Το τμήμα αυτό του κώδικα ονομάζεται *σώμα (body)* της διαδικασίας. Μια διαδικασία έχει ένα σύνολο από παραμέτρους που ονομάζονται *τυπικές παράμετροι (formal parameters)* της διαδικασίας. Μια διαδικασία "καλείται" με συγκεκριμένες παραμέτρους, οι οποίες ονομάζονται *πραγματικές παράμετροι (actual parameters)* της διαδικασίας. Όταν μια διαδικασία καλείται, τότε το σώμα της εκτελείται. Κάθε εκτέλεση του σώματος μιας διαδικασίας ονομάζεται *ενεργοποίηση (activation)*. Η υλοποίηση των διαδικασιών είναι ίσως ένα από τα περισσότερο πολύπλοκα θέματα κατά την δημιουργία ενός μεταγλωττιστή για μια γλώσσα προγραμματισμού. Για το λόγο, στη συνέχεια θα εξετάσουμε με λεπτομέρεια τα διάφορα θέματα που αφορούν την υλοποίηση διαδικασιών.

Στο σημείο αυτό αξίζει να αναφέρουμε ότι ένα από τα χαρακτηριστικά των διαδικασιών που τις καθιστούν ιδιαίτερα χρήσιμες αλλά ταυτόχρονα και δύσκολες στην υλοποίηση, είναι η ύπαρξη *αναδρομής (recursion)*. Μια διαδικασία ονομάζεται αναδρομική είτε όταν η ίδια καλεί τον εαυτό της στο σώμα της, είτε όταν καλεί κάποια άλλη διαδικασία η οποία μέσω μιας σειράς κλήσεων ξανακαλεί την αρχική διαδικασία. Σε επόμενη παράγραφο θα εξετάσουμε τις τεχνικές υλοποίησης αναδρομικών διαδικασιών.

6.2.1 Μέθοδοι Περάσματος Παραμέτρων

Ένα από τα σημαντικότερα θέματα σε σχέση με την υλοποίηση και τη χρήση των διαδικασιών, είναι οι μέθοδοι περάσματος παραμέτρων (parameter-passing methods) που χρησιμοποιεί μια γλώσσα προγραμματισμού. Η έννοια του περάσματος παραμέτρων αναφέρεται στο πως ακριβώς λαμβάνει χώρα η αντικατάσταση των τυπικών παραμέτρων μιας διαδικασίας από τις πραγματικές παραμέτρους. Έχουν προταθεί διάφορες σχετικές μέθοδοι, από τις οποίες θα εξετάσουμε στη συνέχεια τις κυριώτερες.

Κλήση με Τιμή

Στην κλήση με τιμή (call-by-value) οι τυπικές παράμετροι της διαδικασίας που καλείται παίρνουν την τιμή των πραγματικών παραμέτρων. Με άλλα λόγια, για την εκτέλεση μιας κλήσης της μορφής `p(E)`, εκτελούνται τα ακόλουθα βήματα:

- Υπολογίζεται η τιμή της έκφρασης `E` (έστω v).
- Εκτελείται το σώμα της διαδικασίας `p`. Όταν συναντιέται η τυπική παράμετρος της `p` (έστω `x`), τότε αυτή αντικαθίσταται με την τιμή v που έχει προϋπολογιστεί.

Παράδειγμα 6.1 Έστω η διαδικασία:

```
function square(x:integer):integer;
begin
    square := x*x;
end;
```

Αν καλέσουμε τη συνάρτηση `square` ως `square(2+3)`, υπολογίζεται η τιμή `2+3 = 5`, και αρχίζει η εκτέλεση του σώματος της διαδικασίας. Όταν συναντηθεί η έκφραση `x*x`, τα `x` αντικαθίστανται από το `5` και επιστρέφεται η τιμή `25` από τη συγκεκριμένη κλήση. □

Η κλήση με τιμή είναι η πιό συνηθισμένη σε γλώσσες προγραμματισμού όπως η C ή η Pascal. Ένα χαρακτηριστικό της κλήσης με τιμή είναι ότι όταν μια διαδικασία κληθεί με παράμετρο μια μεταβλητή, η τιμή της μεταβλητής δεν μπορεί να μεταβληθεί μέσα από τη διαδικασία.

Παράδειγμα 6.2 Έστω η διαδικασία:

```
procedure change(x,y:integer);
var z:integer;
begin
    z:=x; x:=y; y:=z;
end;
```

Η κλήση `change(a,b)` όπου τα `a`, `b` είναι μεταβλητές τύπου `integer`, δεν έχει κανένα αποτέλεσμα πάνω στις μεταβλητές `a` και `b`. □

Για να ξεπεραστούν προβλήματα όπως το παραπάνω, έχει προταθεί η μέθοδος περάσματος παραμέτρων με αναφορά.

Κλήση με Αναφορά

Στην κλήση με αναφορά (call-by-reference) οι τυπικές παράμετροι μιας διαδικασίας γίνονται συνώνυμες με τη θέση στη μνήμη των αντίστοιχων πραγματικών παραμέτρων. Στην Pascal για παράδειγμα, η λέξη-κλειδί `var` κάνει την τυπική παράμετρο `x` να φιλοξενεί στη διάρκεια των κλήσεων τη διεύθυνση της αντίστοιχης πραγματικής παραμέτρου.

Παράδειγμα 6.3 Έστω η διαδικασία:

```
procedure swap(var x:integer; var y:integer);
var z:integer;
begin
    z:=x; x:=y; y:=z;
end;
```

Μια κλήση της μορφής `swap(a,b)`, όπου `a` και `b` είναι μεταβλητές τύπου `integer`, θα έχει ως αποτέλεσμα την ανταλλαγή των τιμών των μεταβλητών `a` και `b`. □

Η κλήση με αναφορά πραγματοποιείται στη C με τη χρήση δεικτών. Στη Java, είναι η βασική μέθοδος περάσματος παραμέτρων όταν πρόκειται για το πέρασμα αντικειμένων.

Κλήση με Όνομα

Η κλήση με όνομα (call-by-name) φαίνεται να έχει στις μέρες μας κυρίως ιστορικό ενδιαφέρον (τουλάχιστον όσον αφορά τις προστακτικές γλώσσες προγραμματισμού). Αποτελούσε βασική μέθοδο κλήσης στην Algol 60. Στις μέρες μας χρησιμοποιείται (με ιδιαίτερη επιτυχία) μια παρόμοια μέθοδος κλήσης στις συναρτησιακές γλώσσες προγραμματισμού.

Μια συνοπτική περιγραφή της μεθόδου κλήσης με όνομα, είναι η ακόλουθη:

- Οι πραγματικές παράμετροι αντικαθίστανται *ως κείμενο* στο σώμα της διαδικασίας, στα σημεία εκείνα όπου εμφανίζονται οι αντίστοιχες τυπικές παράμετροι. Καθώς είναι πιθανό να υπάρχουν στο σώμα της διαδικασίας μεταβλητές που έχουν το ίδιο όνομα με μεταβλητές που εμφανίζονται στις πραγματικές παραμέτρους, πριν την αντικατάσταση λαμβάνει χώρα αλλαγή ονομάτων των μεταβλητών της διαδικασίας που δημιουργούν το πρόβλημα αυτό.

- Το σώμα της διαδικασίας που προκύπτει μετά την αντικατάσταση, αντικαθίσταται με τη σειρά του στο σημείο της αρχικής κλήσης. Καθώς είναι και πάλι πιθανό να υπάρχουν στο σώμα της διαδικασίας μεταβλητές που έχουν το ίδιο όνομα με μεταβλητές που είναι τοπικές στη σημείο της κλήσης, πριν τη νέα αντικατάσταση λαμβάνει χώρα αλλαγή ονομάτων των τοπικών μεταβλητών που δημιουργούν το πρόβλημα αυτό.

Στις συναρτησιακές γλώσσες προγραμματισμού όπου όπως προαναφέραμε χρησιμοποιείται μια παραλλαγή της παραπάνω μεθόδου, έχουν προταθεί τεχνικές που καθιστούν την κλήση με όνομα ιδιαίτερα αποδοτική και γρήγορη.

6.2.2 Εγγραφές Ενεργοποίησης

Η δυνατότητα δημιουργίας αναδρομικά οριζόμενων διαδικασιών, δημιουργεί μια πληθώρα νέων προβλημάτων σχετικά με την υλοποίηση τους. Έστω για παράδειγμα μια διαδικασία που καλεί αναδρομικά τον εαυτό της (όπως φαίνεται στο παρακάτω παράδειγμα).

Παράδειγμα 6.4 Η παρακάτω συνάρτηση υπολογίζει αναδρομικά το παραγοντικό ενός φυσικού αριθμού:

```
function f(n: integer): integer;
begin
    if n=0 then f:=1 else f:=n*f(n-1);
end;
```

□

Μια αναδρομική διαδικασία μπορεί να έχει περισσότερες από μία ενεργοποιήσεις (activations) που να υπάρχουν ταυτόχρονα. Κάθε μια από τις ενεργοποιήσεις αυτές απαιτεί το δικό της χώρο στη μνήμη για να εκτελεστεί (αφού η κάθε μια έχει τις δικές της τοπικές μεταβλητές οι οποίες είναι ανεξάρτητες από αυτές των υπολοίπων ενεργοποιήσεων).

Σε μια ακολουθιακή (δηλαδή όχι παράλληλη) γλώσσα προγραμματισμού, ο έλεγχος του προγράμματος μπορεί να βρίσκεται το πολύ σε μια ενεργοποίση μιας διαδικασίας κάθε φορά. Έτσι, όταν η διαδικασία `p` καλεί την `q`, η εκτέλεση της `p` σταματά, και ο έλεγχος περνάει στην `q`. Η εκτέλεση της `p` ξαναρχίζει όταν ο έλεγχος επιστρέφει από την `q`. Το ίδιο συμβαίνει και όταν έχουμε μια διαδικασία που καλεί τον εαυτό της. Η ενεργοποίηση

p1 της p καλεί την ενεργοποίηση p2 και αυτή την p3. Όταν τελειώσει η εκτέλεση της p3 ο έλεγχος επιστρέφει στην p2, και όταν ολοκληρωθεί και η p2 επιστρέφει στην p1.

Γενικά, η ροή του ελέγχου ανάμεσα στις ενεργοποιήσεις διαδικασιών, δημιουργεί ένα νοητό δέντρο γνωστό και ως *δέντρο ενεργοποιήσεων (activation tree)*. Έτσι, για παράδειγμα, όταν η ενεργοποίηση p καλεί την q, τότε ο κόμβος του δεντρου για την p έχει τον κόμβο για την q ως παιδί του. Αν η p καλέχει την q πριν την r, τότε ο κόμβος για την r εμφανίζεται δεξιά από τον κόμβο για την q.

Για την υλοποίηση των αναδρομικών διαδικασιών, ο μεταγλωττιστής θα πρέπει να φροντίσει ώστε να φυλάγονται τα δεδομένα των ενεργοποιήσεων για όσο τουλάχιστον χρόνο χρειάζεται μέχρι να ολοκληρωθεί η εκτέλεση των ενεργοποιήσεων. Τα δεδομένα τα οποία χρειάζονται για την εκτέλεση μιας ενεργοποίησης μιας διαδικασίας, μαζεύονται σε μια δομή η οποία ονομάζεται *εγγραφή ενεργοποίησης (activation record)*. Οι εγγραφές τοποθετούνται η μία μετά την άλλη σε μια κατάλληλη μεγαλύτερη δομή, που ονομάζεται *στοίβα (stack)*.

Όταν μια διαδικασία καλείται, μια νέα εγγραφή ενεργοποίσης δημιουργείται και τοποθετείται στην κορυφή της στοίβας. Όταν η διαδικασία αυτή καλέσει με τη σειρά της μια άλλη διαδικασία (ή ακόμη και τον εαυτό της), μια νέα εγγραφή ενεργοποίησης θα δημιουργηθεί και θα τοποθετηθεί πάνω από την προηγούμενη εγγραφή στη στοίβα.

Όταν η εκτέλεση μιας ενεργοποίησης μιας διαδικασίας ολοκληρωθεί, τότε η εγγραφή της απομακρύνεται από την κορυφή της στοίβας, και η εκτέλεση συνεχίζεται με την ενεργοποίηση της διαδικασίας που είχε καλέσει την ενεργοποίηση που ολοκληρώθηκε.

Τα δεδομένα τα οποία φυλάγονται σε μια εγγραφή ενεργοποίησης είναι: οι τιμές των τοπικών μεταβλητών της ενεργοποίησης, οι πραγματικές παράμετροι με τις οποίες κλήθηκε η συγκεκριμένη ενεργοποίηση της διαδικασίας (αν πρόκειται για κλήση με τιμή), η κατάσταση της μηχανής πριν την κλήση (πχ. program counter, registers, κλπ), καθώς και άλλες χρησιμες πληροφορίες.

Κεφάλαιο 7

Λογικός Προγραμματισμός: Η Γλώσσα Prolog

Τα βασικά στοιχεία της γλώσσας προγραμματισμού Prolog είναι δύο: οι ορισμοί (statements) και οι όροι (terms). Οι ορισμοί χωρίζονται στα γεγονότα (facts), στους κανόνες (rules) και στις ερωτήσεις (queries). Οι ορισμοί επιτελούν το ρόλο των εντολών στις κλασσικές γλώσσες προγραμματισμού. Οι όροι είναι ουσιαστικά η μοναδική δομή δεδομένων που διαθέτει η Prolog.

7.1 Τα Γεγονότα

Το απλούστερο είδος ορισμού που υποστηρίζει η Prolog είναι το *γεγονός*. Τα γεγονότα παρέχουν ένα τρόπο για να εκφράσει κανείς ότι μια σχέση ισχύει ανάμεσα σε κάποιες οντότητες. Για παράδειγμα, το γεγονός:

```
father(john,mary).
```

εκφράζει ότι ο `john` είναι πατέρας της `mary`, ή διαφορετικά ότι η σχέση `father` ισχύει ανάμεσα στα άτομα `john` και `mary`. Οι σχέσεις όπως το `father` ονομάζονται επίσης *κατηγορήματα* ενώ τα `mary` και `john` ονομάζονται *ορίσματα* του κατηγορήματος. Ενα κατηγόρημα μαζί με τα ορίσματα που περιλαμβάνει ονομάζεται *ατομικός τύπος*.

Μια άλλη σχέση είναι η `plus`, η οποία μπορεί να οριστεί από ένα σύνολο γεγονότων:

```
plus(0,0,0).
plus(0,1,1).
plus(1,0,1).
plus(2,0,2).
plus(0,2,2).
```

Η σχέση `plus` ορίζει ουσιαστικά την πρόσθεση φυσικών αριθμών. Προφανώς, η σχέση αυτή δεν μπορεί να οριστεί αποκλειστικά με τη χρήση γεγονότων (θα χρειάζονταν ένας άπειρος αριθμός από γεγονότα όπως τα παραπάνω). Θα δούμε αργότερα με ποιό τρόπο μπορεί να γραφεί ένας γενικός ορισμός της σχέσης αυτής.

Η Prolog επιτρέπει τη χρήση μεταβλητών στα γεγονότα (αλλά και στους κανόνες και τις ερωτήσεις). Οι μεταβλητές αρχίζουν πάντα με κεφαλαίο γράμμα. Ενα γεγονός που χρησιμοποιεί μεταβλητή είναι το ακόλουθο:

```
plus(0,X,X).
```

Το παραπάνω γεγονός εκφράζει τη γνωστή ιδιότητα ότι η πρόσθεση του μηδέν σε οποιονδήποτε αριθμό μας δίνει τον ίδιο τον αριθμό. Γενικά, η χρήση μεταβλητών μας επιτρέπει να εκφράζουμε ένα σύνολο γεγονότων.

7.2 Οι Ερωτήσεις

Οι ερωτήσεις παρέχουν τη δυνατότητα εξαγωγής πληροφορίας από ένα λογικό πρόγραμμα. Μια ερώτηση ουσιαστικά διερευνά αν μια σχέση ισχύει ανάμεσα σε κάποιες οντότητες. Για παράδειγμα, η ερώτηση:

```
?-father(john,mary).
```

διερευνά αν η σχέση `father` ισχύει ανάμεσα στα άτομα `john` και `mary`. Αν έχουμε δώσει στην Prolog το γεγονός:

```
father(john,mary).
```

τότε η απάντηση που θα πάρουμε στην παραπάνω ερώτηση θα είναι `yes`. Μπορούμε να χρησιμοποιήσουμε μεταβλητές και στην περίπτωση των ερωτήσεων. Ετσι, η ερώτηση:

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

μπορεί να διαβαστεί ως εξής: "Υπάρχει κάποιο προσωπο `X` το οποίο είναι πατέρας της `mary`;". Μια ερώτηση μπορεί να επιστρέψει περισσότερες από μια απαντήσεις. Ετσι, για παράδειγμα, αν έχουμε δώσει στην Prolog τα δύο γεγονότα:

```
father(john,mary).
father(john,ann).
```

τότε η ερώτηση:

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

θα επιστρέψει δύο λύσεις `X=mary` και `X=ann`.

Επιπλέον, μπορεί κανείς να θέσει στην Prolog και σύνθετες ερωτήσεις. Για παράδειγμα, η ερώτηση

```
?-father(X,mary),father(X,ann).
```

αναζητά το πρόσωπο εκείνο το οποίο είναι πατέρας τόσο της `mary` όσο και της `ann`.

Μια σύνθετη ερώτηση αποτελείται επομένως από υποερωτήσεις, οι οποίες ονομάζονται *κλήσεις*. Οι κλήσεις σε μια σύνθετη ερώτηση χωρίζονται μεταξύ τους με κόμμα το οποίο έχει την έννοια του λογικού "και". Για να αληθεύει μια σύνθετη ερώτηση θα πρέπει να αληθεύουν όλες οι απλούστερες ερωτήσεις που την αποτελούν.

7.3 Οι Κανόνες

Οι κανόνες στην Prolog μας επιτρέπουν να ορίσουμε νέες σχέσεις σα συνάρτηση σχέσεων που έχουν ήδη οριστεί. Για παράδειγμα, ο κανόνας:

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

ορίζει τη σχέση `son` σα συνάρτηση των σχέσεων `father` και `male`. Ο παραπάνω ορισμός μπορεί να διαβαστεί ως εξής: "Ο `X` είναι γιός του `Y` αν ο `Y` είναι πατέρας του `X` και ο `X` είναι γένους αρσενικού". Αξίζει να σημειώσουμε οτι το σύμβολο "`:-`" διαβάζεται σαν "εάν" ενώ το σύμβολο "`,`" διαβάζεται σαν "και".

Μπορεί κανείς να ορίσει και αναδρομικούς κανόνες, όπως είναι ο ακόλουθος:

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

Βλέπουμε από το παραπάνω πρόγραμμα ότι η σχέση `ancestor` (πρόγονος) μπορεί να οριστεί με τη χρήση δύο κανόνων, ο ένας από τους οποίους είναι αναδρομικός.

Γενικά, ένας κανόνας έχει τη μορφή $\mathbf{H} : -\mathbf{B}_1, \ldots, \mathbf{B}_n$, όπου το $\mathbf{H}$ ονομάζεται *κεφαλή* του κανόνα, ενώ οι ατομικοί τύποι $\mathbf{B}_1, \ldots, \mathbf{B}_n$ αποτελούν το *σώμα* του κανόνα.

7.4 Οι Όροι

Η βασική δομή δεδομένων στον λογικό προγραμματισμό είναι ο *όρος (term)*. Ενας όρος μπορεί να είναι μια *σταθερά* (όπως `john`), μια μεταβλητή (όπως `X`), ή κάποιος *σύνθετος όρος*. Ενας σύνθετος όρος αποτελείται από ένα *συναρτησιακό σύμβολο (functional symbol)* το οποίο εφαρμόζεται σε μια ακολουθία από όρους. Για παράδειγμα, ένας σύνθετος όρος είναι το `list(a,nil)`, όπου `list` είναι το συναρτησιακό σύμβολο και τα `a, nil` είναι δύο σταθερές.

Με τη χρήση των όρων μπορεί κανείς να κωδικοποιήσει σύνθετες μορφές πληροφορίας. Για παράδειγμα, μπορεί κανείς να ορίσει ένα κατηγόρημα `book` το οποίο παίρνει σαν παραμέτρους όρους που περιγράφουν τα χαρακτηριστικά της έννοιας βιβλίο. Με τον τρόπο αυτό μπορεί κανείς να δημιουργήσει μια απλή βάση δεδομένων:

```
book(author(surname(stoll),name(robert)),subject(logic)).
book(author(surname(kleene),name(steven)),subject(mathematics)).
                        ...
```

Αξίζει να σημειώσουμε τη διαφορά ανάμεσα στο `book` (το οποίο είναι το όνομα ενός κατηγορήματος) και στα `author`, `surname`, `name`, `subject` τα οποία είναι συναρτησιακά σύμβολα.

Σε ένα πρόγραμμα όπως το παραπάνω, μπορεί κανείς να κάνει ερωτήσεις. Έστω για παράδειγμα οτι θέλουμε να βρούμε τα ονοματεπώνυμα όσων έχουν γράψει βιβλία σε λογική. Η ερώτηση που μπορούμε να θέσουμε είναι η ακόλουθη:

```
?-book(author(surname(X),name(Y)),subject(logic)).
```

Αν δε μας ενδιαφέρουν τα μικρά ονόματα των συγγραφέων, μπορούμε να θέσουμε την ερώτηση ως εξής:

```
?-book(author(surname(X),_),subject(logic)).
```

όπου με "_" συμβολίζουμε τους όρους εκείνους των οποίων η τιμή δε μας ενδιαφέρει.

Με τη χρήση των όρων μπορεί κανείς να γράψει προγράμματα τα οποία δεν μπορούν να γραφούν απλώς με τη χρήση γεγονότων και κανόνων. Έστω για παράδειγμα οτι θέλουμε να ορίσουμε τη σχέση `nat(X)` η οποία αληθεύει όταν `X` είναι ένας φυσικός αριθμός. Ενας τρόπος για να γίνει αυτό είναι να συμφωνήσουμε να αναπαριστούμε τους φυσικούς αριθμούς αντί για 0, 1, 2,... με `0, s(0), s(s(0)),...`. Το συναρτησιακό σύμβολο `s` εκφράζει την έννοια του "επόμενου αριθμού". Ετσι, το πρόγραμμα για το `nat` μπορεί να γραφεί ως εξής:

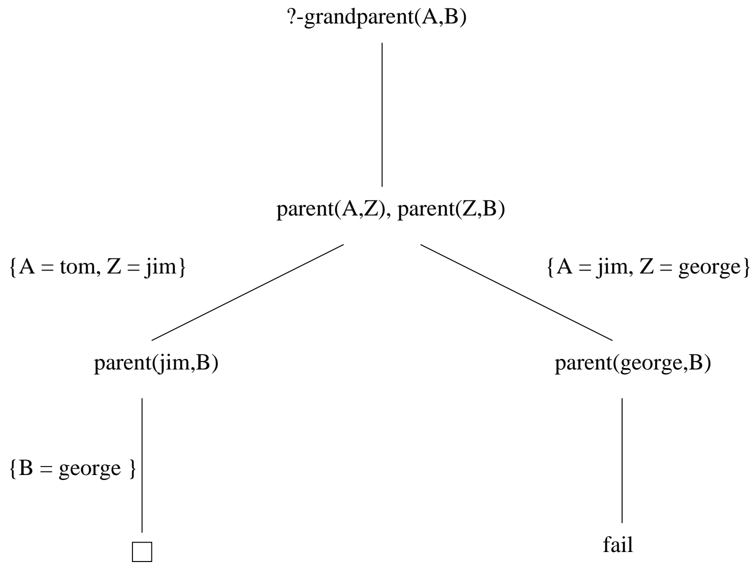
```
nat(0).
nat(s(X)):-nat(X).
```

Με τον τρόπο αυτό μπορεί κανείς να ορίσει και άλλες σχέσεις πάνω σε φυσικούς αριθμούς. Για παράδειγμα, στο ακόλουθο πρόγραμμα το `plus(X,Y,Z)` είναι αληθές αν το `Z` είναι το άθροισμα των `X` και `Y`.

```
plus(0,X,X).
plus(s(X),Y,s(Z)):-plus(X,Y,Z).
```

Θέτοντας την ερώτηση:

```
?-plus(s(0),s(0),K).
```



Σχήμα 7.1: Το Δένδρο Εκτέλεσης της Ερώτησης `?-grandparent(A,B).`

παίρνουμε την απάντηση `K = s(s(0))`.

Αξίζει να σημειωθεί οτι η Prolog δεν καθορίζει κάποια από τα ορίσματα ενός κατηγορήματος να είναι είσοδοι και κάποια έξοδοι. Ετσι, πολλά από τα προγράμματα που γράφει κανείς σε Prolog μπορούν να έχουν διαφορετικές χρήσεις. Για παράδειγμα, αν κανείς θέσει την ερώτηση:

```
?-plus(X,s(0),s(s(s(0)))).
```

παίρνει την απάντηση `X=s(s(0))`, η οποία αντιστοιχεί στον αριθμό που παίρνουμε αν αφαιρέσουμε από το τρίτο όρισμα το δεύτερο.

7.5 Εκτέλεση Προγραμμάτων Prolog

Έστω το παρακάτω πρόγραμμα Prolog:

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

Οταν θέσουμε την ερώτηση:

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

ο μηχανισμός εκτέλεσης της Prolog εκτελεί μια αναζήτηση των λύσεων, η οποία περιγράφεται από το δέντρο του Σχήματος 7.1. Αρχικά, ο μηχανισμός εκτέλεσης της Prolog αναζητά

στο πρόγραμμα κάποιο κατηγόρημα που να ονομάζεται `grandparent` έτσι ώστε να ταιριάζει με τη δεδομένη ερώτηση. Μόλις βρεί μια πρόταση που αφορά το κατηγόρημα αυτό, αντικαθιστά την δεδομένη ερώτηση με το σώμα της πρότασης, και έτσι δημιουργείται η νέα (σύνθετη) ερώτηση `parent(A,Z),parent(Z,B)`. Για να αποδείξει την ερώτηση αυτή η Prolog πρέπει να αποδείξει κάθε ένα από τα `parent(A,Z)` και `parent(Z,B)`. Ξεκινάει με το πρώτο από αυτά δηλαδή με το `parent(A,Z)`, και ψάχνει να βρεί στο πρόγραμμα κάποια πρόταση με την οποία αυτό ταιριάζει. Η πρώτη πρόταση που υπάρχει στο πρόγραμμα με την οποία ταιριάζει το `parent(A,Z)` είναι η `parent(tom,jim)` (οι δύο προτάσεις ταιριάζουν μόνο αν θεωρήσουμε οτι `A=tom` και `Z=jim`). Μένει λοιπόν να αποδείξουμε το `parent(Z,B)`, το οποίο αφού υποθέσαμε οτι `Z=jim`, είναι ισοδύναμο με `parent(jim,B)`. Η μόνη πρόταση που υπάρχει στο πρόγραμμα και που ταιριάζει με αυτό είναι η `parent(jim,george)`, κάτω από την προϋπόθεση οτι `B=george`. Ετσι, η πρώτη λύση που παίρνουμε είναι η `A=tom, B=george`.

Ο μηχανισμός της Prolog θα ψάξει τώρα και για άλλες πιθανές λύσεις. Οπισθοχωρεί στο σημείο εκείνο του δένδρου στο οποίο είχε κάποια εναλλακτική επιλογή. Το σημείο αυτό είναι ο κόμβος που περιέχει την ερώτηση `parent(A,Z),parent(Z,B)` και η εναλλακτική επιλογή που υπάρχει είναι να ταιριάξει το `parent(A,Z)` με το `parent(jim,george)` (δηλαδή να κάνει την υπόθεση οτι `A=jim` και `Z=george`). Τώρα μένει να αποδειχτεί το `parent(Z,B)`, το οποίο αφού υποθέσαμε οτι `Z=george`, είναι ισοδύναμο με `parent(george,B)`. Ομως, δεν υπάρχει καμμία πρόταση στο πρόγραμμα με την οποία να ταιριάζει το `parent(george,B)`, και επομένως δεν υπάρχει άλλη λύση στη συγκεκριμένη ερώτηση.

Πιό γενικά, ο μηχανισμός εκτέλεσης της Prolog μπορεί να περιγραφεί ως εξής. Έστω οτι έχουμε ένα πρόγραμμα και μια ερώτηση της μορφής $? - \mathbf{A}_1, \ldots, \mathbf{A}_n$. Τότε:

- Η Prolog εξετάζει αν ικανοποιούνται όλα τα $\mathbf{A}_1, \ldots, \mathbf{A}_n$, ξεκινώντας από το $\mathbf{A}_1$ και πηγαίνοντας προς το $\mathbf{A}_n$.

- Για να ικανοποιήσει ένα από τα $\mathbf{A}_i$, η Prolog διαλέγει την πρώτη πρόταση στο πρόγραμμα της οποίας η κεφαλή ταιριάζει με το $\mathbf{A}_i$, και αντικαθιστά το $\mathbf{A}_i$ με το σώμα της πρότασης αυτής (αφού πρώτα το τροποποιήσει κατάλληλα).

Η παραπάνω διαδικασία συνεχίζεται μέχρι να μην υπάρχει πλέον άλλη κλήση που να πρέπει να ικανοποιηθεί.

Η περιγραφή της παραπάνω διαδικασίας που δώσαμε, δεν είναι λεπτομερής (πχ. τι ακριβώς σημαίνει "ταιριάζει";). Στις παρούσες σημειώσεις δεν θα εξετάσουμε το μηχανισμό εκτέλεσης της Prolog με μεγαλύτερη λεπτομέρεια.

7.6 Απλά Αναδρομικά Προγράμματα σε Prolog

Με τη χρήση όρων (terms) μπορεί κανείς να γράφει απλά αλλά και εκφραστικά προγράμματα σε Prolog. Ξαναθυμίζουμε το πρώτο τέτοιο πρόγραμμα που συναντήσαμε και το οποίο ορίζει την πρόσθεση φυσικών αριθμών:

```
plus(0,X,X).
plus(s(X),Y,s(Z)):-plus(X,Y,Z).
```

Ομοίως μπορούμε να ορίσουμε τον πολλαπλασιασμό φυσικών αριθμών:

```
times(0,X,0).
times(s(X),Y,Z):-times(X,Y,W),plus(W,Y,Z).
```

Η παραπάνω σχέση χρησιμοποιεί δύο βασικά αξιώματα του πολλαπλασιασμού, το $0*X=0$ και το $(X+1)*Y=X*Y+Y$. Αξίζει να σημειώσουμε το γεγονός ότι ο ορισμός του πολλαπλασιασμού χρησιμοποιεί αυτόν της πρόσθεσης. Ομοίως, χρησιμοποιώντας τον ορισμό του πολλαπλασιασμού μπορούμε να δώσουμε την ακόλουθη σχέση για το παραγοντικό ενός φυσικού αριθμού:

```
factorial(0,s(0)).
factorial(s(N),F):-factorial(N,F1),times(s(N),F1,F).
```

Με παρόμοιο τρόπο μπορεί κανείς να ορίσει και άλλες μαθηματικές σχέσεις πάνω σε φυσικούς αριθμούς. Ενα τελευταίο παράδειγμα του τύπου αυτού δίνεται από την ακόλουθη σχέση `between(X,Y,Z)` η οποία αληθεύει όταν `X` είναι ένας φυσικός αριθμός που είναι μικρότερος ή ίσος από τον `Y` και ο `Y` είναι μικρότερος ή ίσος από τον `Z`.

```
between(0,0,Z).
between(0,s(Y),s(Z)):-between(0,Y,Z).
between(s(X),s(Y),s(Z)):-between(X,Y,Z).
```

Οπως θα δούμε αργότερα, οι φυσικοί αριθμοί μπορούν να αναπαρασταθούν με πολύ πιο βολικό τρόπο στην Prolog (όπως και στις υπόλοιπες γλώσσες προγραμματισμού). Η αναπαράσταση που χρησιμοποιήσαμε στα παραπάνω προγράμματα δεν είναι γενικά βολική και αποτελεσματική, και την υιοθετήσαμε απλώς για να δείξουμε κάποιες από τις αρχικές δυνατότητες της Prolog.

7.7 Αναδρομικός Προγραμματισμος με Λίστες

Μια πολύ χρήσιμη δομή δεδομένων που χρησιμοποιείται στον προγραμματισμό σε Prolog είναι η *λίστα*. Μια λίστα στην Prolog μπορεί να είναι κενή (οπότε και συμβολίζεται με `[]`) ή να περιέχει κάποια στοιχεία (π.χ. η `[a,b,c]` είναι μια μη-κενή λίστα).

Στην περίπτωση που μια λίστα είναι μη-κενή, θα ονομάζουμε *κεφαλή (head)* το πρώτο στοιχείο της λίστας, και *ουρά (tail)* τη λίστα που μένει αν αφαιρέσουμε την κεφαλή. Πολλές φορές θα χρησιμοποιούμε τον τελεστή `|` ο οποίος αποτελεί μια ειδική αναπαράσταση που μας δείχνει άμεσα την κεφαλή και την ουρά μιας λίστας. Ετσι, η λίστα `[a,b,c]` θα γράφεται επίσης και στη μορφή `[a|[b,c]]` ενώ η λίστα `[a]` είναι ισοδύναμη με την λίστα `[a|[]]`.

Παράδειγμα 7.1 Οι λίστες της Prolog μπορούν να περιέχουν σα στοιχεία τους και άλλες λίστες αλλά και πολύπλοκους όρους. Για παράδειγμα, η `[[]]` είναι μια επιτρεπτή λίστα, όπως επίσης και ή `[[1,X],s(s(X))]` είναι επίσης μια λίστα. Ομοίως, και οι `[a|[1|[2]]]` και `[X,Y|[a]]` είναι επίσης λίστες που είναι ισοδύναμες με τις λίστες `[a,1,2]` και `[X,Y,a]`, αντίστοιχα. □

Πρέπει εδώ να σημειώσουμε οτι οι λίστες δεν είναι τίποτα περισσότερο από μια κατηγορία όρων (terms) η οποία όμως έχει μια ειδική αναπαράσταση.

Χρησιμοποιώντας λίστες μπορεί κανείς να γράψει χρήσιμα και ευκολονόητα προγράμματα, όπως δείχνουν και τα ακόλουθα παραδείγματα:

Παράδειγμα 7.2 Το ακόλουθο πρόγραμμα ορίζει τη σχέση `member(X,Y)` η οποία είναι αληθής όταν το `X` είναι ένα στοιχείο το οποίο ανήκει στη λίστα `Y`.

```
member(X,[X|Y]).
member(X,[Y|Z]):-member(X,Z).
```

Το νόημα του παραπάνω προγράμματος έχει ως εξής: "Το `X` είναι μέλος μιας λίστας αν είναι η κεφαλή της λίστας ή αν είναι μέλος της ουράς της λίστας". □

Παράδειγμα 7.3 Το ακόλουθο πρόγραμμα ορίζει τη σχέση `append(X,Y,Z)` η οποία είναι αληθής όταν `Z` είναι η λίστα που προκύπτει από την συννένωση των λιστών `X` και `Y`.

```
append([],Y,Y).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
```

Η δομή του παραπάνω προγράμματος είναι παρόμοια με αυτή του προγράμματος για τον ορισμό της σχέσης `plus`. □

Παράδειγμα 7.4 Είναι πολύ συχνό φαινόμενο στον προγραμματισμό σε Prolog να χρησιμοποιεί κανείς κάποιες απλές σχέσεις που έχει ήδη ορίσει, για να γράφει ακόμη πιο πολύπλοκα προγράμματα. Σχέσεις όπως η `member` και η `append` χρησιμοποιούνται σχεδόν σε οποιοδήποτε μεγάλο πρόγραμμα γράφει κανείς σε Prolog. Ενα παράδειγμα σχέσης που ορίζεται σα συνάρτηση της `append`, είναι η σχέση `reverse(X,Y)`, που είναι αληθής όταν η λίστα `Y` είναι η αντίστροφη της λίστας `X`:

```
reverse([],[]).
reverse([X|Xs],Ys):-reverse(Xs,Rs),append(Rs,[X],Ys).
```

Το νόημα του παραπάνω προγράμματος έχει ως εξής: "Η αντίστροφη της κενής λίστας είναι η κενή λίστα. Η αντίστροφη μιας μη-κενής λίστας μπορεί να παραχθεί αντιστρέφοντας την ουρά της και προσκολλώντας στο τέλος της αντεστραμένης ουράς την κεφαλή της αρχικής λίστας."

Το παραπάνω πρόγραμμα μπορεί όμως να γραφεί και χωρίς τη χρήση της append. Αυτό μπορεί να γίνει με την προσθήκη ενός βοηθητικού κατηγορήματος reverse1, το οποίο διαθέτει ένα επιπλέον όρισμα:

```
reverse(Xs,Ys):-reverse1(Xs,[],Ys).

reverse1([],Ys,Ys).
reverse1([X|Xs],A,Ys):-reverse1(Xs,[X|A],Ys).
```

Το όρισμα A στο παραπάνω πρόγραμμα ονομάζεται *συσσωρευτής (accumulator)* διότι συσσωρεύει βήμα-βήμα το τελικό αποτέλεσμα (δηλαδή την αντίστροφη λίστα). □

7.8 Αναδρομικός Προγραμματισμος με Δέντρα

Εκτός από τις λίστες, ένας άλλος πολύ χρήσιμος τύπος δεδομένων είναι τα *δέντρα*. Στην παράγραφο αυτή θα εξετάσουμε πως μπορεί κανείς να αναπαραστήσει δυαδικά δέντρα στην Prolog (γενικότεροι τύποι δέντρων μπορούν να αναπαρασταθούν με ανάλογο τροπο).

Οπως έχουμε αναφέρει, η μοναδική δομή δεδομένων που διαθέτει η Prolog είναι ο όρος. Επομένως, και τα δυαδικά δέντρα θα πρέπει να αναπαρασταθούν με τη χρήση όρων. Για το σκοπό αυτό χρησιμοποιούμε το συναρτησιακό σύμβολο tree το οποίο παίρνει τρεις παραμέτρους, δηλαδή γράφουμε: tree(Element,Left,Right). Η ρίζα του συγκεκριμένου δέντρου είναι το στοιχείο Element, το αριστερό υποδέντρο είναι το Left και το δεξιό υποδέντρο είναι το Right. Το κενό δέντρο θα αναπαριστάται με τη σταθερά void. Για παράδειγμα, το δέντρο με κορυφή a, αριστερό παιδί b και δεξί παιδί c μπορεί να γραφεί στη μορφή:

```
tree(a,tree(b,void,void),tree(c,void,void))
```

Το πρώτο πρόγραμμα που παραθέτουμε ελέγχει αν ένας δεδομένος όρος είναι πράγματι δυαδικό δέντρο (το πρόγραμμα είναι ανάλογο του κατηγορήματος nat που ορίσαμε για τους φυσικούς αριθμούς):

```
binary_tree(void).
binary_tree(tree(E,L,R)):-binary_tree(L),binary_tree(R).
```

Μπορούμε τώρα να δώσουμε ένα πρόγραμμα το οποίο ελέγχει αν ένα δεδομένο στοιχείο ανήκει σε ένα δυαδικό δέντρο ή όχι. Το πρόγραμμα αυτό είναι ανάλογο με το member που έχουμε ορίσει για τις λίστες:

```
tree_member(X,tree(X,_,_)).
tree_member(X,tree(Y,L,R)):-tree_member(X,L).
tree_member(X,tree(Y,L,R)):-tree_member(X,R).
```

Το παραπάνω πρόγραμμα ελέγχει αν ένα δεδομένο στοιχείο είναι ταυτόσημο με την κορυφή του δεδομένου δέντρου. Αν ναι, τότε το πρόγραμμα σταματά με επιτυχία. Αν όχι, το πρόγραμμα συνεχίζει αναδρομικά τη διερεύνηση στο αριστερό και το δεξί υποδέντρο.

Μπορούμε να ορίσουμε και κατηγορήματα τα οποία ουσιαστικά "διασχίζουν" ένα δεδομένο δέντρο με μια προκαθορισμένη σειρά. Το αποτέλεσμα της διάσχισης μπορούμε να το επιστρέφουμε σε μια λίστα. Γιά παράδειγμα, η preorder διάσχιση ενός δυαδικού δέντρου μπορεί να οριστεί από το ακόλουθο κατηγόρημα:

```
preorder(void,[]).
preorder(tree(X,L,R),Xs):-preorder(L,Ls),
                          preorder(R,Rs),
                          append([X|Ls],Rs,Xs).
```

Θυμίζουμε οτι κατά την preorder διάσχιση ενός δέντρου καταγράφουμε αρχικά την ρίζα του δέντρου και κατόπιν επισκεπτόμαστε αναδρομικά πρώτα το αριστερό υποδέντρο και μετά το δεξί. Με ανάλογο τρόπο μπορούμε να γράψουμε πρόγραμμα που υλοποιεί τη διάσχιση inorder:

```
inorder(void,[]).
inorder(tree(X,L,R),Xs):-inorder(L,Ls),
                         inorder(R,Rs),
                         append(Ls,[X|Rs],Xs).
```

Τέλος, το πρόγραμμα για την postorder είναι το ακόλουθο:

```
postorder(void,[]).
postorder(tree(X,L,R),Xs):-postorder(L,Ls),
                           postorder(R,Rs),
                           append(Ls,Rs,Ms),
                           append(Ms,[X],Xs).
```

Είναι προφανές οτι τα παραπάνω προγράμματα διαφέρουν μόνο ως προς τον τρόπο που γίνεται η συννένωση των επιμέρους αποτελεσμάτων (append).

7.9 Τελεστές

Η Prolog δίνει τη δυνατότητα στον προγραμματιστή να επεξεργαστεί ακόμη πιό πολύπλοκες δομές από οτι οι λίστες και τα δέντρα, με τη χρήση των *τελεστών (operators)*. Οι τελεστές είναι στην ουσία συναρτησιακά σύμβολα. Για παράδειγμα, όταν γράφουμε σε Prolog την έκφραση `1+3`, το `+` είναι ένας τελεστής και η έκφραση αυτή είναι ισοδύναμη με τον όρο `+(1,3)`, και όχι με τον αριθμό 4 όπως συμβαίνει στις άλλες γλώσσες προγραμματισμού (η πρόσθεση θα πραγματοποιούνταν μόνο αν ο όρος `1+3` περνούσε σαν όρισμα στο ειδικό κατηγόρημα `is` της Prolog που χρησιμοποιείται για τον υπολογισμό αριθμητικών εκφράσεων).

Επομένως, τελεστές όπως οι $+, -, *, /, \ldots$ δεν παίρνουν κάποιο συγκεκριμένο νόημα από την Prolog και κατά συνέπεια μπορούμε να τους χρησιμοποιήσουμε για να γράφουμε διάφορες εφαρμογές με κομψό και συνοπτικό τρόπο. Στο παρακάτω πρόγραμμα, οι διάφοροι τελεστές χρησιμοποιούνται για να εκφράσουμε τη διαδικασία της παραγώγισης συναρτήσεων:

```
derivative(N,X,0):-nat(N).
derivative(X,X,s(0)).
derivative(X^s(N),X,s(N)*(X^N)).
derivative(sin(X),X,cos(X)).
derivative(cos(X),X,-sin(X)).
derivative(e^X,X,e^X).
derivative(log(X),X,1/X).
derivative(F+G,X,DF+DG):-derivative(F,X,DF),derivative(G,X,DG).
derivative(F-G,X,DF-DG):-derivative(F,X,DF),derivative(G,X,DG).
derivative(F*G,X,F*DG+G*DF):-derivative(F,X,DF),derivative(G,X,DG).
derivative(1/F,X,-DF/(F*F)):-derivative(F,X,DF).
derivative(F/G,X,(G*DF-F*DG)/(G*G)):-derivative(F,X,DF),
                                      derivative(G,X,DG).
```

Το παραπάνω πρόγραμμα υπολογίζει την παράγωγο μιας μεγάλης κατηγορίας συναρτήσεων. Οι διάφοροι τελεστές που χρησιμοποιούνται, εκφράζουν πράξεις μεταξύ συναρτήσεων. Για παράδειγμα, την έκφραση `F*G` στο παραπάνω πρόγραμμα, ο προγραμματιστής την καταλαβαίνει σαν πολλαπλασιασμό συναρτήσεων, και αυτό είναι δυνατό διότι η Prolog δεν δίνει ένα προκαθορισμένο νόημα σε σύμβολα όπως το $*$.

Αν στο παραπάνω πρόγραμμα δώσουμε την ερώτηση:

```
?-derivative(cos(x)*(x^s(s(0))),x,D).
```

η οποία αναζητά την παράγωγο της συνάρτησης $cos(x) * x^2$ θα λάβουμε την απάντηση:

```
D = cos(x)*(s(s(0))*x^s(0))+x^s(s(0))*(-sin(x))
```

η οποία αντιστοιχεί στη συνάρτηση $cos(x) * (2 * x) + x^2 * (-sin(x))$.

Τέλος, αξίζει να σημειώσουμε οτι στην Prolog μπορεί κανείς να ορίσει και ο ίδιος νέους τελεστές τους οποίους κατόπιν να χρησιμοποιήσει για να γράψει πιό εύκολα και ευανάγνωστα προγράμματα.

7.10 Αριθμητικές Πράξεις και Prolog

Κάθε γλώσσα προγραμματισμού παρέχει ιδιαίτερες ευκολίες στη χρήση και επεξεργασία αριθμών. Ομοίως και η Prolog διαθέτει ειδικά κατηγορήματα, γνωστά και ως *κατηγορήματα συστήματος (system predicates)*, για το σκοπό αυτό. Το βασικό τέτοιο κατηγόρημα είναι το `is` το οποίο χρησιμοποιείται στη μορφή `V is E`. Η Prolog υπολογίζει την τιμή του `E` και αν η τιμή αυτή συμφωνεί με το `V` τότε επιτυγχάνει. Αν το `V` είναι μεταβλητή, τότε η μεταβλητή αυτή καταλαμβάνει την τιμή που έχει προκύψει από τον υπολογισμό του `E`. Οταν το `E` περιέχει μεταβλητές των οποίων η τιμή δεν είναι γνωστή τη στιγμή της εκτέλεσης, ο διερμηνέας της Prolog δίνει μήνυμα λάθους.

Παράδειγμα 7.5 Η ερώτηση

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

επιτυγχάνει, ενώ η ερώτηση

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

αποτυγχάνει διότι η Prolog βλέπει το αριστερό μέλος σαν τον όρο `+(5,3)` και δεν υπολογίζει την τιμή του. Η ερώτηση

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

θα δώσει την απάντηση `X=8`. Τέλος, η ερώτηση

```
?-X is 2+Y.
```

θα δώσει μήνυμα λάθους διότι η αριθμητική τιμή του δεξιού μέλους δεν μπορεί να υπολογιστεί λόγω της μεταβλητής `Y`. □

Αλλα κατηγορήματα συστήματος που χρησιμοποιεί η Prolog για αριθμητικές πράξεις είναι και τα ακόλουθα:

- `X=:=Y` (Οι αριθμητικές τιμές των `X` και `Y` είναι ίδιες)
- `X=\=Y` (Οι αριθμητικές τιμές των `X` και `Y` είναι διαφορετικές)
- `X<Y` (Η αριθμητική τιμή του `X` είναι μικρότερη από αυτή του `Y`)

- `X=<Y` (Η αριθμητική τιμή του `X` είναι μικρότερη ή ίση από αυτή του `Y`)
- `X>Y` (Η αριθμητική τιμή του `X` είναι μεγαλύτερη από αυτή του `Y`)
- `X>=Y` (Η αριθμητική τιμή του `X` είναι μεγαλύτερη ή ίση από αυτή του `Y`)

Με τη χρήση των παραπάνω κατηγορημάτων συστήματος μπορούμε να ξαναγράψουμε με απλούστερο τρόπο κάποια κατηγορήματα τα οποία προηγουμένως τα ορίζαμε με τη χρήση όρων (π.χ. $0, s(0), \ldots$). Ετσι, το κατηγόρημα `plus` μπορεί τώρα να οριστεί ως εξής:

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

Το μειονέκτημα του παραπάνω τρόπου γραφής είναι ότι χάνεται η αντιστρεψιμότητα κάποιων προγραμμάτων. Αν για παράδειγμα δώσουμε την ερώτηση:

```
?-plus(A,B,8).
```

θα πάρουμε μήνυμα λάθους και οχι όλα τα ζεύγη φυσικών που έχουν άθροισμα το 8, διότι η παραπάνω ερώτηση ανάγεται από το μηχανισμό εκτέλεσης της Prolog στην ερώτηση

```
?-8 is A+B.
```

Με τη χρήση όμως των κατηγορημάτων συστήματος μπορεί κανείς να γράφει προγράμματα που εκτελούν αριθμητικές πράξεις με το συνηθισμένο τρόπο:

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

Το παραπάνω πρόγραμμα υπολογίζει το άθροισμα των στοιχείων μιας λίστας με τη χρήση του `is`.

7.11 Η Αποκοπή

Τα προγράμματα Prolog είναι συχνά αναποτελεσματικά γιατί πολλές φορές το δέντρο αναζήτησης είναι αρκετά μεγάλο. Υπάρχουν περιπτώσεις όπου μας ενδιαφέρει αν ένα πρόβλημα έχει λύση και οχι το πόσες και ποιές είναι οι λύσεις αυτές. Στις περιπτώσεις αυτές δεν είναι απαραίτητο για το μηχανισμό εκτέλεσης της Prolog να διασχίσει ολόκληρο το δέντρο αναζήτησης, αλλά μόνο ένα μέρος του. Ο προγραμματιστής μπορεί να καθοδηγήσει το μηχανισμό εκτέλεσης ώστε να αποφύγει τις περιττές αναζητήσεις. Αυτό γίνεται με τη βοήθεια της *αποκοπής (cut)* η οποία συμβολίζεται με το θαυμαστικό (!). Η λειτουργία της αποκοπής μπορεί να εξηγηθεί με ένα παράδειγμα:

Παράδειγμα 7.6 Έστω μια ερώτηση της μορφής:

$$?-\mathbf{A},\mathbf{B},\mathbf{C}.$$

και το ακόλουθο τμήμα προγράμματος:

$$\begin{aligned} &\dots \\ &\mathbf{B} :- \mathbf{A}_1, \dots, \mathbf{A}_k, !, \mathbf{A}_{k+1}, \dots, \mathbf{A}_n. \\ &\mathbf{B} :- \mathbf{D}_0, \dots, \mathbf{D}_m. \\ &\mathbf{B} :- \mathbf{F}_0, \dots, \mathbf{F}_k. \end{aligned}$$

Ο ρόλος που παίζει η αποκοπή είναι ο ακόλουθος:

- Οταν ο μηχανισμός εκτέλεσης της Prolog περάσει την αποκοπή τότε τα $\mathbf{A}_1, \dots, \mathbf{A}_k$ δεν πρόκειται να ξαναεξεταστούν για επιπλέον λύσεις.
- Οταν ο μηχανισμός εκτέλεσης της Prolog περάσει την αποκοπή τότε οι εναλλακτικές προτάσεις για το **B** που υπάρχουν στο πρόγραμμα μετά την πρόταση που περιέχει την αποκοπή, δεν πρόκειται να εξεταστούν.

Είναι προφανές οτι μπορεί να υπάρχουν κάποιες λύσεις οι οποίες αποκόπτονται από την παραπάνω διαδικασία. □

Το παρακάτω παράδειγμα μας δείχνει με πιό συγκεκριμένο τρόπο το αποτέλεσμα που έχει μια αποκοπή:

Παράδειγμα 7.7 Έστω το παρακάτω πρόγραμμα:

```
father(tom,jim).
male(jim).
son(X,Y):-father(Y,X),male(X).
son(george,jim).
son(john,nick).
```

και η ερώτηση

```
?-son(S,F).
```

Ο μηχανισμός εκτέλεσης της Prolog θα δώσει τρεις πιθανές λύσεις: $\{\mathtt{S}=\mathtt{jim},\mathtt{F}=\mathtt{tom}\}$, $\{\mathtt{S}=\mathtt{george},\mathtt{F}=\mathtt{jim}\}$, $\{\mathtt{S}=\mathtt{john},\mathtt{F}=\mathtt{nick}\}$. Αν το παραπάνω πρόγραμμα αντικατασταθεί με το νέο:

```
father(tom,jim).
male(jim).
son(X,Y):-father(Y,X),!,male(X).
son(george,jim).
son(john,nick).
```

ο μηχανισμός εκτέλεσης της Prolog θα δώσει μόνο μία λύση, την $\{\mathtt{S} = \mathtt{jim}, \mathtt{F} = \mathtt{tom}\}$. Οι άλλες δύο λύσεις αποκόπτονται διότι από τη στιγμή που περάσουμε το ! οι εναλλακτικές προτάσεις για το son(S,F) δεν εξετάζονται. □

Η αποκοπή συνήθως χρησιμοποιείται όταν γνωρίζουμε εκ τω προτέρων οτι το πρόβλημα μας έχει μια και μοναδική λύση και θέλουμε να περιορίσουμε το άσκοπο ψάξιμο που μπορεί να κάνει η Prolog. Με τον τρόπο αυτό μπορεί να γράψει κανείς προγράμματα που είναι αρκετά αποτελεσματικά.

Ομως, η αποκοπή δεν ανήκει στο λογικό τμήμα της Prolog και για το λόγο αυτό μπορεί να δημιουργήσει πολλές φορές δυσνόητα προγράμματα. Επιπλέον, η λανθασμένη χρησιμοποίηση της μπορεί να περικόψει λύσεις οι οποίες είναι χρήσιμες και επιθυμητές.

7.12 Ασκήσεις

1. Γράψτε προγράμματα Prolog που ορίζουν τις ακόλουθες σχέσεις:
   - is_set(S) η οποία είναι αληθής όταν η λίστα S είναι σύνολο.
   - union(S1,S2,S3) η οποία είναι αληθής όταν το σύνολο S3 είναι η ένωση των συνόλων S1 και S2.
   - intersection(S1,S2,S3) η οποία είναι αληθής όταν το σύνολο S3 είναι η τομή των συνόλων S1 και S2.
   - subset(S1,S2) η οποία είναι αληθής όταν το σύνολο S1 είναι υποσύνολο του S2.
   - equal(S1,S2) η οποία είναι αληθής όταν το σύνολο S1 είναι ίδιο με το S2.
2. Γράψτε προγράμματα Prolog που ορίζουν τις ακόλουθες σχέσεις:
   - doublel(L,M) όπου κάθε στοιχείο της L εμφανίζεται διαδοχικά δύο φορές στην λίστα M (πχ. doublel([1,2,5],[1,1,2,2,5,5]) είναι αληθές).
   - repetition(L1,L2) η οποία είναι αληθής αν η λίστα L2 μπορεί να προκύψει από την παράθεση της λίστας L1 με τον εαυτό της μία ή και περισσότερες φορές (πχ. repetition([a,b], [a,b,a,b,a,b]) είναι αληθές).
   - nodupl(L1,L2) όπου η L2 είναι σαν την L1 με τη διαφορά ότι δεν περιέχει επαναλήψεις στοιχείων (πχ. nodupl([a,b,c,a,d,b,c], [a,b,c,d]) είναι αληθές).
   - sums(L,N) η οποία είναι αληθής αν υπάρχουν αριθμοί στη λίστα L που να έχουν άθροισμα ίσο με N (πχ. sums([1,4,7,23],28) είναι αληθές διότι 1+4+23=28).

3. Γράψτε ένα πρόγραμμα Prolog το οποίο ορίζει τη σχέση `ordered(Tree)` η οποία είναι αληθής εάν το `Tree` είναι ένα διατεταγμένο δέντρο αριθμών, δηλαδή για κάθε κόμβο του δέντρου τα στοιχεία στο αριστερό υποδέντρο είναι μικρότερα από το στοιχείο στον κόμβο ενώ τα στοιχεία στο αριστερό υποδέντρο είναι μεγαλύτερα από το στοιχείο στον κόμβο. Για παράδειγμα,

   ```
   ?-ordered(tree(3,tree(2,void,void),tree(4,void,void))).
   ```

   είναι αληθές, ενώ

   ```
   ?-ordered(tree(3,tree(7,void,void),tree(4,void,void))).
   ```

   είναι ψευδές. Για τις συγκρίσεις χρησιμοποιήστε τους σχετικούς τελεστές που υποστηρίζει η Prolog.

Κεφάλαιο 8

Συναρτησιακός Προγραμματισμός: Η Γλώσσα Haskell

8.1 Συναρτησιακός Προγραμματισμός

Συναρτησιακές γλώσσες προγραμματισμού είναι οι γλώσσες στις οποίες ο προγραμματιστής χρησιμοποιεί σχεδόν αποκλειστικά συναρτήσεις για την περιγραφή ενός αλγορίθμου ή για την επίλυση ενός προβλήματος. Θεωρήστε για παράδειγμα το ακόλουθο συναρτησιακό πρόγραμμα το οποίο υπολογίζει τον μέγιστο δύο αριθμών:

```
max(a,b) = if (a>b) then a else b
```

Το παραπάνω πρόγραμμα μοιάζει πολύ με έναν καθαρά μαθηματικό ορισμό. Τα a,b ονομάζονται *τυπικές παράμετροι (formal parameters)* της συνάρτησης max. Οταν δίνουμε συγκεκριμένες τιμές στις τυπικές παραμέτρους, θα λέμε ότι *καλούμε* τη συνάρτηση με τις παραμέτρους αυτές. Για παράδειγμα, όταν γράφουμε max(2,3) ουσιαστικά καλούμε τη συνάρτηση με πρώτη παράμετρο 2 και δεύτερη παράμετρο 3. Οι παράμετροι με τις οποίες καλούμε μια συνάρτηση (όπως τα 2 και 3 στο παράδειγμα μας) ονομάζονται *πραγματικές παράμετροι (actual parameters)*.

Η δύναμη του συναρτησιακού προγραμματισμού ξεκινάει από τη δυνατότητα που προσφέρει στον προγραμματιστή να συνδυάζει συναρτήσεις για να επιλύσει πιό πολύπλοκα προβλήματα. Θεωρήστε για παράδειγμα το πρόβλημα της εύρεσης του μεγίστου τριών αριθμών. Το πρόβλημα μπορεί να επιλυθεί εύκολα ορίζοντας μια νέα συνάρτηση η οποία χρησιμοποιεί τη συνάρτηση max που ορίσαμε παραπάνω:

```
max3(a,b,c) = max(a,max(b,c))
```

Οι συναρτήσεις που ορίζουμε σε μια συναρτησιακή γλώσσα προγραμματισμού, διαφέρουν σε ένα βασικό σημείο από τις συναρτήσεις που μπορούν να οριστούν σε μια προστακτική

γλώσσα προγραμματισμού. Πιό συγκεκριμένα, γλώσσες προγραμματισμού όπως η C και η Pascal, δεν διαθέτουν (σε αντίθεση με τις συναρτησιακές γλώσσες) την ιδιότητα η οποία συχνά αναφέρεται ως *διαφάνεια αναφοράς (referential transparency)*. Ο όρος αυτός σημαίνει ότι:

> Κάθε έκφραση σε ένα πρόγραμμα αντιστοιχεί σε μια και μοναδική τιμή. Αν η ίδια έκφραση ξαναϋπολογιστεί τότε θα δώσει και πάλι την ίδια τιμή (δηλαδή ο υπολογισμός της τιμής μιας έκφρασης δεν αλλάζει ποτέ την τιμή της).

Ο όρος μπορεί να επεξηγηθεί καλύτερα με ένα παράδειγμα. Θεωρήστε το ακόλουθο πρόγραμμα Pascal:

```
program example(output);
var flag:boolean;

function f(n:integer):integer;
begin
    if flag then f:=n
        else f:= 2*n;
    flag := not flag
end;

begin
    flag := true;
    writeln(f(1)+f(2));
    writeln(f(2)+f(1));
end.
```

Αν εκτελέσουμε το πρόγραμμα αυτό παίρνουμε δύο τιμές (5 και 4). Θα περιμέναμε όμως και στις δύο περιπτώσεις να είχαμε το ίδιο αποτέλεσμα (επηρεασμένοι ίσως από το γεγονός ότι έχουμε συνηθίσει να θεωρούμε δεδομένη την αντιμεταθετικότητα της πρόσθεσης από τα μαθηματικά). Το πρόβλημα βέβαια είναι οτι οι “συναρτήσεις” που μπορεί κανείς να ορίσει σε μια γλώσσα σαν την Pascal διαφέρουν γενικά από τις μαθηματικές συναρτήσεις. Το πρόβλημα στις γλώσσες αυτές ξεκινάει από το γεγονός ότι χρησιμοποιούν εντολές ανάθεσης (assignments) και ολικές (global) μεταβλητές.

Οι συναρτησιακές γλώσσες διαθέτουν την ιδιότητα της διαφάνειας αναφοράς και έτσι τα προγράμματα που μπορούμε να γράψουμε σ'αυτές αντιπροσωπεύουν στην ουσία μαθηματικές συναρτήσεις.

8.2 Εισαγωγή στη Γλώσσα Haskell

Η Haskell είναι μια σύγχρονη συναρτησιακή γλώσσα η οποία μπορεί να χρησιμοποιηθεί για απλές μέχρι αρκετά πολύπλοκες εφαρμογές. Οταν καλούμε τον διερμηνέα της Haskell, εμφανίζεται το σύμβολο ? στο οποίο ο χρήστης μπορεί να δώσει μια έκφραση για υπολογισμό. Για παράδειγμα:

```
? (2+3)*8
40
? sum [1..10]
55
?
```

Στο δεύτερο παράδειγμα, ο συμβολισμός `[1..10]` αναπαριστά τη λίστα των ακεραίων από το 1 μέχρι το 10. Το `sum` είναι μια συνάρτηση του συστήματος η οποία υπολογίζει το άθροισμα των στοιχείων μιας λίστας. Οπως θα δούμε αργότερα ο χρήστης μπορεί να ορίσει τις δικές του συναρτήσεις.

8.3 Τύποι της Haskell

Ενα σημαντικό κομμάτι της Haskell είναι το *σύστημα τύπων* της το οποίο είναι πολύ χρήσιμο για την ανίχνευση λαθών σε εκφράσεις και ορισμούς συναρτήσεων τα οποία γράφει ο προγραμματιστής. Η Haskell δίνει ένα τύπο σε κάθε έκφραση και ο τύπος αυτός χαρακτηρίζει το είδος της τιμής που αναπαριστάται από την έκφραση αυτή (καθορίζει δηλαδή αν η έκφραση αναπαριστά έναν ακέραιο, έναν αριθμό κινητής υποδιαστολής, κλπ). Γενικά, θα γράφουμε *έκφραση::τύπος* όταν θέλουμε να δείξουμε ότι μια έκφραση έχει ένα συγκεκριμένο τύπο. Ετσι για παράδειγμα:

- Οταν γράφουμε `42::Int` εννοούμε οτι το 42 είναι ακέραιος (`Int` είναι το όνομα που αναπαριστά τον τύπο των ακεραίων αριθμών).
- Οταν γράφουμε `fact::Int->Int` εννοούμε οτι `fact` είναι μια συνάρτηση που παίρνει σαν όρισμα έναν ακέραιο και επιστρέφει έναν ακέραιο.

Η Haskell διαθέτει ένα ευρύ ρεπερτόριο από τύπους οι οποίοι και περιγράφονται στις επόμενες παραγράφους.

Λογικές Τιμές Ο τύπος των λογικών τιμών συμβολίζεται με `Bool` και μπορεί να λάβει δύο τιμές: `True` και `False`. Η Haskell υποστηρίζει τις γνωστές συναρτήσεις για την επεξεργασία λογικών τιμών:

- Η τιμή της έκφρασης `a && b` είναι `True` εαν και μόνο εάν και το `a` και το `b` είναι `True`.
- Η τιμή της έκφρασης `a || b` είναι `True` αν ένα τουλάχιστο από τα `a` και `b` είναι `True`.
- Η τιμή της έκφρασης `not a` είναι `True` αν το `a` είναι `False`, και είναι `False` αν το `a` είναι `True`.

Ακέραιες Τιμές Ο τύπος των ακεραίων συμβολίζεται με `Int` και περιλαμβάνει τόσο θετικούς όσο και αρνητικούς αριθμούς. Η Haskell υποστηρίζει πολλές συναρτήσεις για την επεξεργασία ακεραίων. Για παράδειγμα:

`+`	Πρόσθεση
`*`	Πολλαπλασιασμός
`-`	Αφαίρεση
`/`	Ακέραιη Διαίρεση
`rem`	Υπόλοιπο
`odd`	Επιστρέφει `True` αν το όρισμα είναι περιττός. Διαφορετικά επιστρέφει `False`.
`even`	Επιστρέφει `True` αν το όρισμα είναι άρτιος. Διαφορετικά επιστρέφει `False`.
`abs`	Επιστρέφει την απόλυτη τιμή του ορίσματος

Αριθμοί Κινητής Υποδιαστολής Ο τύπος των αριθμών κινητής υποδιαστολής συμβολίζεται με `Float`. Οι αριθμοί αυτοί χρησιμοποιούνται σε υπολογισμούς στους οποίους χρειαζόμαστε μεγάλη ακρίβεια. Οι πράξεις με τους αριθμούς κινητής υποδιαστολής είναι γενικά ακριβείς. Σε πολύπλοκους όμως υπολογισμούς μπορεί να υπεισέλθουν σφάλματα στρογγυλοποίησης που να αλλοιώνουν το τελικό αποτέλεσμα.

Οι αριθμοί κινητής υποδιαστολής αναπαρίστανται ή με τη χρήση υποδιαστολής (πχ. `3.141`) ή με το λεγόμενο επιστημονικό συμβολισμό. Για παράδειγμα, γράφουμε `1.0e3` το οποίο είναι ισοδύναμο με `1000.0`, ενώ το `5.0e-2` είναι ισοδύναμο με `0.05`.

Χαρακτήρες Ο τύπος των χαρακτήρων συμβολίζεται με `Char`. Στοιχεία του τύπου αυτού είναι οι διάφοροι ατομικοί χαρακτήρες (όπως είναι για παράδειγμα οι χαρακτήρες που υπάρχουν σε ένα πληκτρολόγιο). Ενα στοιχείο με τύπο `Char` γράφεται σαν ένας χαρακτήρας που περικλείεται από αποστρόφους (πχ. `'a'`, `'0'`, `'Z'`).

Σε αντίθεση με κάποιες άλλες γλώσσες προγραμματισμού, ο τύπος `Char` είναι διαφορετικός από τον τύπο `Int`. Υπάρχουν όμως κάποιες συναρτήσεις που μας επιτρέπουν να μετατρέπουμε ένα χαρακτήρα στον αντίστοιχο ASCII κώδικα και αντίστροφα:

```
? ord 'a'
97
```

```
? chr 65
'A'
```

Η `ord` δηλαδή μας επιστρέφει τον αντίστοιχο ASCII κώδικα ενός χαρακτήρα, ενώ η `chr` μας επιστρέφει το χαρακτήρα που αντιστοιχεί σε ένα ASCII κώδικα.

Λίστες Αν `t` είναι ένας τύπος της Haskell, τότε `[t]` συμβολίζει τον τύπο των λιστών που περιέχουν στοιχεία τύπου `t`. Για παράδειγμα, `[Int]` είναι οι λίστες που περιέχουν ακέραιους αριθμούς. Υπάρχουν διάφοροι τρόποι για να γράψει κανείς μια λίστα:

- Η απλούστερη λίστα είναι η κενή λίστα η οποία στη Haskell συμβολίζεται με `[]`.
- Μια μη κενή λίστα μπορεί να γραφεί είτε απαριθμώντας τα στοιχεία της (για παράδειγμα `[1,3,10]`), είτε χρησιμοποιώντας τον τελεστή ":" ως εξής:
  `[1,3,10] = 1:[3,10] = 1:(3:[10]) = 1:(3:(10:[]))`

Η Haskell υποστηρίζει μια ποικιλία συναρτήσεων για την επεξεργασία λιστών. Για παράδειγμα:

- `length xs` επιστρέφει τον αριθμό των στοιχείων της λίστας `xs`.
- `xs ++ ys` επιστρέφει μια νέα λίστα που αποτελείται από τα στοιχεία της `xs` ακολουθούμενα από τα στοιχεία της `ys`.
- `concat xss` επιστρέφει τη λίστα των στοιχείων που υπάρχουν σε κάθε μια από τις λίστες στην `xss`.

Έτσι έχουμε:

```
? length [1,3,10]
3
? [1,3,10]++[2,6,5,7]
[1,3,10,2,6,5,7]
? concat [[1],[2,3],[],[4,5,6]]
[1,2,3,4,5,6]
```

Αξίζει να σημειώσουμε οτι στην Haskell όλα τα στοιχεία μιας λίστας πρέπει να έχουν τον ίδιο τύπο (δηλαδή να είναι είτε όλα ακέραιοι, είτε όλα χαρακτήρες, κλπ).

Συμβολοσειρές Ο τύπος των συμβολοσειρών συμβολίζεται με `String` και είναι ουσιαστικά μια συντομογραφία του τύπου `[Char]`. Οι συμβολοσειρές γράφονται σαν ακολουθίες χαρακτήρων οι οποίες περικλείονται απο το χαρακτήρα `"` (πχ. `"hello, world"` είναι μια έκφραση τύπου `String`). Καθώς οι συμβολοσειρές είναι ουσιαστικά λίστες χαρακτήρων, μπορούμε να εφαρμόσουμε σ'αυτές τις συναρτήσεις που χρησιμοποιούμε και για γενικές λίστες:

```
? length "Hello"
5
? "Hello, " ++ "World"
Hello, World
? concat ["mary","ann"]
maryann
```

Ν-άδες Αν `t1, t2,...,tn` είναι τύποι, τότε ο τύπος της n-άδας συμβολίζεται με `(t1, t2,...,tn)`. Για παράδειγμα:

- `(Int, [Int], Float)` αναπαριστά το σύνολο των τριάδων των οποίων το πρώτο στοιχείο είναι ακέραιος, το δεύτερο μια λίστα ακεραίων και το τρίτο ένας αριθμός κινητής υποδιαστολής.
- `(Char, Bool)` αναπαριστά το σύνολο των δυάδων των οποίων το πρώτο στοιχείο είναι ένας χαρακτήρας και το δεύτερο μια λογική τιμή.

Συναρτήσεις Αν `t1` και `t2` είναι τύποι, τότε `t1->t2` είναι ο τύπος μιας συνάρτησης που παίρνει ορίσματα τύπου `t1` και επιστρέφει αποτελέσματα τύπου `t2`. Για παράδειγμα:

- Μια συνάρτηση `add` η οποία παίρνει σαν όρισμά της μια δυάδα αριθμών και επιστρέφει το άθροισμά τους, έχει τύπο: `(Int,Int) -> Int`.
- Η συνάρτηση `ord` έχει τύπο: `Char -> Int`.
- Η συνάρτηση `chr` έχει τύπο: `Int -> Char`.

8.4 Ορίζοντας Συναρτήσεις στη Haskell

Παρόλο που η Haskell υποστηρίζει ένα ευρύ φάσμα από συναρτήσεις συστήματος (όπως είναι για παράδειγμα η `product`, `abs`, `chr`, `ord`, κλπ), υπάρχουν πολλές περιπτώσεις στις οποίες ο χρήστης επιθυμεί να ορίσει κάποιες επιπλέον συναρτήσεις (οι οποίες για παράδειγμα θα επιλύουν κάποιο ειδικό πρόβλημα). Μια συνάρτηση της Haskell έχει τη γενική μορφή:

$$f\ pat_1\ \cdots\ pat_n\ =\ E$$

Κάθε ένα από τα $pat_1, \ldots, pat_n$ αναπαριστά ένα όρισμα της συνάρτησης και ονομάζεται *πρότυπο* (pattern), ενώ E είναι μια έκφραση της γλώσσας. Τα $pat_1, \ldots, pat_n$ μπορεί να είναι σταθερές τιμές, μεταβλητές ή να έχουν ακόμη πιό πολύπλοκη μορφή όπως θα δούμε στη συνέχεια. Επιπλέον, μια συνάρτηση μπορεί να ορίζεται με περισσότερες από μία εξισώσεις.

Παράδειγμα 8.1 Η συνάρτηση `succ` που ορίζεται ως:

```
succ n = n+1
```

έχει ως μοναδικό της όρισμα τη μεταβλητή `n`. Η συνάρτηση `not` όμως, η οποία ορίζεται με τις δύο εξισώσεις:

```
not True  = False
not False = True
```

έχει σαν ορίσματα της και στις δύο περιπτώσεις μια σταθερή λογική τιμή (`True` και `False` αντίστοιχα). □

Και άλλες σταθερές (όπως ακέραιοι, χαρακτήρες, συμβολοσειρές, κλπ) μπορούν να χρησιμοποιηθούν ως πρότυπα. Για παράδειγμα, μπορούμε να ορίσουμε την ακόλουθη συνάρτηση `hello`:

```
hello "Panos" = "Hi"
hello name    = "Hello, "++name++" nice to meet you!"
```

Βλέπουμε ότι η παραπάνω συνάρτηση ορίζεται με δύο εξισώσεις στην πρώτη από τις οποίες χρησιμοποιείται σαν όρισμα μια σταθερά τύπου `String` ενώ στη δεύτερη μια μεταβλητή. Μπορούμε να καλέσουμε την παραπάνω συνάρτηση:

```
?hello "Panos"
Hi
?hello "John"
Hello John, nice to meet you!
```

Πρέπει να σημειώσουμε οτι η σειρά με την οποία γράφονται οι εξισώσεις στη Haskell είναι πολύ σημαντική, διότι όταν καλούμε μια συνάρτηση το σύστημα χρησιμοποιεί τον πρώτο ορισμό ο οποίος μπορεί να εφαρμοστεί. Ετσι, αν αντιστρέψουμε τη σειρά των εντολών στο παραπάνω πρόγραμμα, και ξανακαλέσουμε τη συνάρτηση, θα έχουμε:

```
?hello "Panos"
Hello Panos, nice to meet you!
```

Ενας άλλος τρόπος για να ορίσει κανείς συναρτήσεις στη Haskell είναι με τη χρήση *εξισώσεων περιπτώσεων* (guarded equations). Ενας τέτοιος ορισμός έχει τη γενική μορφή:

$$\begin{array}{lcl} f\ x_1 \cdots x_n & \mid & condition_1 \ = \ e_1 \\ & \cdots & \\ & \mid & condition_n \ = \ e_n \end{array}$$

Η Haskell υπολογίζει τις τιμές των συνθηκών τη μία μετά την άλλη μέχρι να βρεί την πρώτη που είναι αληθής, οπότε και επιστρέφει την τιμή της αντίστοιχης έκφρασης.

Παράδειγμα 8.2 Η ακόλουθη συνάρτηση `oddity::Int->String` εξετάζει αν το όρισμα της είναι άρτιος ή περιττός και επιστρέφει `even` και `odd` αντίστοιχα:

```
oddity n | even n    = "even"
         | otherwise = "odd"
```

□

Η συνθήκη `otherwise` είναι ισοδύναμη με τη λογική σταθερά `True`. Επομένως, γράφοντας `otherwise` σε μια συνθήκη έχει σα συνέπεια η αντίστοιχη έκφραση να χρησιμοποιείται οπωσδήποτε αν καμμία προηγούμενη συνθήκη δεν έχει ικανοποιηθεί.

8.5 Αναδρομικά Προγράμματα με Ακεραίους

Η αναδρομή είναι μια ιδιαίτερα σημαντική τεχνική η οποία μπορεί να χρησιμοποιηθεί για να ορίσει κανείς συναρτήσεις που επεξεργάζονται διάφορους τύπους δεδομένων. Στην παράγραφο αυτή περιγράφουμε την έννοια της αναδρομής και παρουσιάζουμε διαφορετικούς τρόπους που μπορεί κανείς να χρησιμοποιήσει για να ορίσει αναδρομικά προγράμματα.

Ας υποθέσουμε οτι θέλουμε να υπολογίσουμε το παραγοντικό ενός αριθμού n. Το πρόβλημα μπορεί να χωριστεί σε δύο τμήματα:

- Το n είναι μηδέν οπότε και το $n!$ είναι ίσο με 1.
- Διαφορετικά, $n! = 1 * 2 * \cdots * (n-1) * n = (n-1)! * n$. Κατά συνέπεια μπορούμε να υπολογίσουμε την τιμή του $n!$ υπολογίζοντας πρώτα την τιμή του $(n-1)!$ και κατόπιν πολλαπλασιάζοντας με n.

Η διαδικασία αυτή μπορεί να εκφραστεί στη Haskell ως εξής:

```
fact n = if (n==0) then 1 else n * fact (n-1)
```

Το παραπάνω πρόγραμμα μοιάζει να είναι κυκλικό, αλλά δεν είναι έτσι. Καθώς στο δεξιό μέλος της εξίσωσης η τιμή του `n` μειώνεται κατά 1, θα φτάσει κάποια στιγμή που θα γίνει ίση με 0, και από εκεί και πέρα δε θα χρειάζεται να ξανακληθεί η συνάρτηση `fact`. Ας δούμε πως υπολογίζεται το `fact 3`:

```
fact 3 = 3 * fact  2
       = 3 * 2 * fact  1
       = 3 * 2 * 1 * fact  0
       = 3 * 2 * 1 * 1
       = 6
```

Ενας άλλος τρόπος για να γράφει κανείς το ίδιο πρόγραμμα είναι με τη χρήση εξισώσεων περιπτώσεων:

```
fact n | (n==0)      = 1
       | otherwise   = n * fact (n-1)
```

Μπορούμε επίσης να γράψουμε την ίδια συνάρτηση με τη χρήση προτύπων:

```
fact 0 = 1
fact n = n * fact (n-1)
```

Τελος μπορούμε να γράψουμε το ίδιο πρόγραμμα χρησιμοποιώντας τα λεγόμενα `(n+k)` πρότυπα της Haskell:

```
fact 0     = 1
fact (n+1) = (n+1) * fact n
```

Το `(n+1)` που εμφανίζεται στο αριστερό μέρος της δεύτερης εξίσωσης συμβουλεύει τη Haskell ότι πρέπει να περιμένει ένα όρισμα που είναι μεγαλύτερο ή ίσο του ένα. Ο τελευταίος αυτός ορισμός διαφέρει από όλους τους προηγούμενους που δώσαμε στο ότι αν ο χρήστης ζητήσει την τιμή του `fact (-1)` θα λάβει διαγνωστικό μήνυμα λάθους ενώ όλοι οι άλλοι ορισμοί θα οδηγήσουν σε μη-τερματισμό.

8.6 Αναδρομικά Προγράμματα με Λίστες

Οι τεχνικές που χρησιμοποιούνται για τον ορισμό αναδρομικών προγραμμάτων με ακέραιους, μπορούν να χρησιμοποιηθούν για να ορίσουμε αναδρομικές συναρτήσεις σε λίστες. Εστω για παράδειγμα ότι θέλουμε να ορίσουμε μια συνάρτηση `length` η οποία υπολογίζει το μήκος μιας λίστας[1]. Υπάρχουν δύο περιπτώσεις που πρέπει κανείς να εξετάσει:

- Αν η λίστα είναι κενή, τότε η συνάρτηση `length` πρέπει να επιστρέφει την τιμή 0.
- Αν η λίστα δεν είναι κενή τότε μπορεί να γραφεί στη μορφή `x:xs` όπου `x` είναι η κεφαλή της λίστας και `xs` είναι η ουρά της. Τότε η αρχική λίστα είναι κατά ένα στοιχείο μεγαλύτερη από την `xs`, και επομένως έχει μήκος `1+length(xs)`.

Οι παραπάνω παρατηρήσεις οδηγούν στο ακόλουθο πρόγραμμα:

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

[1] Κάποιες από τις συναρτήσεις που ορίζουμε στη συνέχεια έχουν το ίδιο όνομα με αντίστοιχες συναρτήσεις που υποστηρίζει η Haskell.

Ο τρόπος που η `length` λειτουργεί για να υπολογίσει το μήκος μιας λίστας φαίνεται παρακάτω:

```
length [2,5,7]  =  1 + length [5,7]
                =  1 + (1 + length [7])
                =  1 + (1 + (1 + length [ ]))
                =  1 + (1 + (1 + 0))
                =  3
```

Βλέπουμε ότι στα αναδρομικά προγράμματα με λίστες χρησιμοποιείται ένα ακόμη είδος προτύπου, το `[]` που αφορά την κενή λίστα και το `x:xs`, το οποίο αφορά τις μη-κενές λίστες. Με το πρότυπο αυτό μπορεί κανείς να ορίσει δύο βασικές συναρτήσεις που επιστρέφουν την κεφαλή και την ουρά μιας λίστας αντίστοιχα:

```
head (x:xs) = x
tail (x:xs) = xs
```

Ακολούθως, θα δώσουμε παραδείγματα από διάφορες άλλες χρήσιμες συναρτήσεις που επεξεργάζονται λίστες.

Παράδειγμα 8.3 Θέλουμε να ορίσουμε μια συνάρτηση `sum` η οποία αθροίζει όλα τα στοιχεία μιας λίστας. Αυτό μπορεί να γίνει ως εξής:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

Με ανάλογο τρόπο μπορεί κανείς να ορίσει και τη συνάρτηση `product` η οποία επιστρέφει το γινόμενο των στοιχείων μιας λίστας. □

Παράδειγμα 8.4 Θέλουμε να ορίσουμε μια συνάρτηση `append` η οποία ενώνει δύο λίστες. Αυτό μπορεί να γίνει ως εξής:

```
append [] ys     = ys
append (x:xs) ys = x:(append xs ys)
```

Για παράδειγμα, `append [1,4,1] [1,5] = [1,4,1,1,5]`. □

Παράδειγμα 8.5 Χρησιμοποιώντας τη συνάρτηση `append` μπορούμε να ορίσουμε τη συνάρτηση `reverse` η οποία αντιστρέφει μια λίστα:

```
reverse []     = []
reverse (x:xs) = append (reverse xs) [x]
```

Χρησιμοποιώντας την `reverse` μπορεί κανείς να ορίσει τη συνάρτηση `last` η οποία επιστρέφει το τελευταίο στοιχείο μιας λίστας:

```
last xs = head (reverse xs)
```

Έτσι, για παράδειγμα: `reverse [1,2,3,4] = [4,3,2,1]` και `last [1,3,5,7] = 7`. □

Παράδειγμα 8.6 Θέλουμε να ορίσουμε μια συνάρτηση `take` η οποία επιστρέφει μια λίστα με τα n πρώτα στοιχεία μιας δεδομένης λίστας (υποθέτουμε ότι η δεδομένη λίστα θα έχει πάντα τουλάχιστον n στοιχεία). Αυτό μπορεί να γραφεί ως εξής:

```
take  0     xs     = []
take (n+1) (x:xs) = x:(take n xs)
```

Παρόμοια μπορούμε να ορίσουμε μια συνάρτηση `drop` η οποία "πετάει" τα πρώτα n στοιχεία μιας δεδομένης λίστας και επιστρέφει μια λίστα με τα υπόλοιπα στοιχεία (πάλι υποθέτουμε ότι η δεδομένη λίστα έχει τουλάχιστον n στοιχεία).

```
drop  0     xs     = xs
drop (n+1) (x:xs) = drop n xs
```

Για παράδειγμα, θα έχουμε: `take 3 [1,3,4,7,8] = [1,3,4]` και `drop 3 [1,3,4,7,8] = [7,8]`. □

Παράδειγμα 8.7 Θέλουμε να ορίσουμε μια συνάρτηση `select` η οποία διαλέγει το n-οστό στοιχείο μιας δεδομένης λίστας. Αυτό μπορεί να γίνει ώς εξής:

```
select  1    (x:xs) = x
select (n+1) (x:xs) = select n xs
```

Για παράδειγμα, `select 3 [1,6,2,6,7] = 2`. □

Παράδειγμα 8.8 Η συνάρτηση `member` παίρνει σαν ορίσματα ένα στοιχείο και μια λίστα, και επιστρέφει `True` ή `False` ανάλογα με το εάν το στοιχείο ανήκει ή όχι στη λίστα.

```
member x  []    = False
member x (y:xs) = if (x==y) then True
                  else (member x xs)
```

Με παρόμοιο τρόπο μπορούμε να ορίσουμε τη συνάρτηση `delete` η οποία διαγράφει όλες τις εμφανίσεις ενός στοιχείου από μια λίστα:

```
delete x []     = []
delete x (y:xs) = if (x==y) then (delete x xs)
                  else (y:delete x xs)
```

Για παράδειγμα, `delete 3 [1,3,5,3,7] = [1,5,7]`. □

Παράδειγμα 8.9 Μπορεί κανείς να ορίσει συναρτήσεις οι οποίες συγκρίνουν δύο λίστες και ελέγχουν αν ισχύουν δεδομένες σχέσεις μεταξύ τους. Η συνάρτηση `prefix` που ορίζουμε παρακάτω, ελέγχει αν μια λίστα είναι πρόθεμα (prefix) μιας άλλης:

```
prefix  []     ys    = True
prefix (x:xs) (y:ys) = if (x==y) then (prefix xs ys)
                           else False
```

Για να ελέγξουμε αν μια λίστα είναι επίθεμα (suffix) μιας άλλης αρκεί να γράψουμε:

```
suffix xs ys = prefix (reverse xs) (reverse ys)
```

□

Παράδειγμα 8.10 Η συνάρτηση `less` που ορίζεται παρακάτω παίρνει δύο ορίσματα, έναν αριθμό και μια λίστα από αριθμούς, και επιστρέφει όλους εκείνους τους αριθμούς της λίστας που είναι μικρότεροι από το δεδομένο αριθμό. Δηλαδή:

```
?less 3 [1,2,1,3,4,5]
[1,2,1]
```

Ο ορισμός της `less` είναι ο ακόλουθος:

```
less x  []    = []
less x (y:ys) = if (y<x) then (y:less x ys)
                else (less x ys)
```

Ομοίως μπορεί κανείς να ορίσει τη συνάρτηση `greteq` ή οποία παίρνει και αυτή δύο ορίσματα, έναν αριθμό και μια λίστα από αριθμούς, και επιστρέφει όλους εκείνους τους αριθμούς της λίστας που είναι μεγαλύτεροι ή ίσοι από το δεδομένο αριθμό. Δηλαδή:

```
?greateq 3 [1,2,1,3,3,4,5,3]
[3,3,4,5,3]
```

```
greateq x  []    = []
greateq x (y:ys) = if (y>=x) then (y:greateq x ys)
                   else (greateq x ys)
```

□

Παράδειγμα 8.11 Μπορεί κανείς να χρησιμοποιήσει τις δύο παραπάνω συναρτήσεις που ορίσαμε για να δημιουργήσει μια συνάρτηση που ταξινομεί λίστες αριθμών. Η βασική ιδέα είναι η εξής: παίρνουμε την κεφαλή `x` της λίστας, και βρίσκουμε όλα τα στοιχεία της λίστας που απομένει και που είναι μικρότερα από την κεφαλή και μετά όλα τα στοιχεία που είναι μεγαλύτερα ή ίσα από την κεφαλή. Τις δύο αυτές υπολίστες τις ταξινομούμε αναδρομικά και μετά τις ενώνουμε τοποθετώντας το `x` στη μέση. Αυτό γράφεται ως εξής:

```
sort [] = []
sort (x:xs) = (sort (less x xs))++[x]++(sort (greateq x xs))
```

□

Οπως βλέπουμε, στη Haskell μπορούμε με σχετικά απλά και σύντομα προγράμματα να λύσουμε αποτελεσματικά διάφορα προβλήματα, χρησιμοποιώντας κυρίως την αναδρομικότητα της γλώσσας.

8.7 Μέθοδος Κλήσης Συναρτήσεων στην Haskell

Σε πολλές γλώσσες προγραμματισμού, οι τιμές των πραγματικών παραμέτρων μιας συνάρτησης υπολογίζονται πριν περάσουν στη συνάρτηση. Στην περίπτωση αυτή λέμε ότι οι παράμετροι περνάνε *με τιμή (call by value).* Το κύριο πλεονέκτημα της κλήσης με τιμή είναι ότι μπορεί να υλοποιηθεί με ιδιαίτερα εύκολο τρόπο. Το μειονέκτημα της είναι ότι πολλές φορές πραγματοποιεί υπολογισμούς που δεν χρειάζονται (αν για παράδειγμα κάποιο από τα ορίσματα της συνάρτησης δεν χρησιμοποιηθεί στο σώμα της συνάρτησης).

Ένας άλλος τρόπος περάσματος παραμέτρων είναι η *κλήση με ζήτηση (call by need),* στην οποία όλες οι παράμετροι περνάνε στο σώμα της συνάρτησης χωρίς να έχει υπολογιστεί αρχικά η τιμή τους. Η τιμή των παραμέτρων αυτών υπολογίζεται μόνο αν κάτι τέτοιο πραγματικά χρειαστεί. Η Haskell χρησιμοποιεί την κλήση με ζήτηση, σε αντίθεση με άλλες γλώσσες προγραμματισμού οι οποίες χρησιμοποιούν την κλήση με τιμή.

Παράδειγμα 8.12 Εστω η συνάρτηση:

```
ignore x = "I don't need to evaluate my argument"
```

Αν καλέσουμε τη συνάρτηση:

```
?ignore(1/0)
I don't need to evaluate my argument
```

Ενω (από την εμπειρία μας από άλλες γλώσσες) θα περιμέναμε η Haskell να δώσει κάποιο διαγνωστικό λάθους λόγω διαίρεσης με μηδέν, αυτό δε συμβαίνει. Ο λόγος είναι ότι η Haskell υπολογίζει την τιμή του ορίσματος μιας συνάρτησης μόνο όταν το όρισμα αυτό πραγματικά χρειάζεται για τον υπολογισμό της τιμής της συνάρτησης. □

Παράδειγμα 8.13 Θεωρήστε τη συνάρτηση:

```
f x y = if (x<=1) then x+1 else y
```

Εστω ότι καλούμε τη συνάρτηση αυτή ως: `f 1 (fact 30)`. Αν το πέρασμα των παραμέτρων γίνεται με τιμή, τότε πρέπει πριν αρχίσει ο υπολογισμός της τιμής της συνάρτησης, να

υπολογιστούν οι τιμές των παραγματικών παραμέτρων. Ετσι, υπολογίζεται αρχικά η τιμή του `fact 30`, παρόλο που δεν πρόκειται να χρειαστεί στο σώμα της συνάρτησης (αφού το `x` έχει την τιμή 1). Στην περίπτωση που η κλήση γίνει με ζήτηση, η τιμή του `fact 30` δεν θα υπολογιστεί γιατί δεν θα χρειαστεί κατά τον υπολογισμό του αποτελέσματος. □

8.8 Πολυμορφισμός στη Haskell

Η συνάρτηση `sum` που έχουμε ορίσει, παίρνει σαν παράμετρο μιά λίστα από αριθμούς και επιστρέφει σαν αποτέλεσμα έναν αριθμό που είναι το άθροισμα όλων των στοιχείων της λίστας. Με άλλα λόγια, η `sum` εφαρμόζεται μόνο σε λίστες που περιέχουν αριθμητικά δεδομένα. Αν θεωρήσουμε όμως τη συνάρτηση `length` δεν είναι αμέσως προφανές ποιός είναι ο τύπος της. Η `length` μπορεί να υπολογίσει το μήκος μιας λίστας ανεξάρτητα από το είδος των στοιχείων της λίστας. Μπορούμε δηλαδή να καλέσουμε την `length` με παράμετρο μια λίστα ακεραίων ή με μια λίστα χαρακτήρων, κοκ. Ετσι, για παράδειγμα μπορούμε να γράψουμε:

```
?lengthl([1,1,3,4,7,17])
6
?lengthl("Hello")
5
```

Έτσι, ο τύπος της `length` είναι `[a]->Int`, όπου το `a` στην παράσταση αυτή αναπαριστά έναν οποιονδήποτε άλλο τύπο (και για το λόγο αυτό το `a` ονομάζεται *μεταβλητή τύπου*).

Μια συνάρτηση της οποίας ο τύπος περιέχει μια ή περισσότερες μεταβλητές τύπου, ονομάζεται *πολυμορφική*. Το να υποστηρίζει μια γλώσσα πολυμορφικές συναρτήσεις είναι πολύ σημαντικό διότι με τον τρόπο αυτό ο προγραμματιστής αποφεύγει σε πολλές περιπτώσεις να ορίσει διαφορετικές συναρτήσεις για διαφορετικούς τύπους δεδομένων.

8.9 Άπειρες Δομές Δεδομένων

Ενα βασικό πλεονέκτημα της κλήσης με ζήτηση είναι οτι επιτρέπει την επεξεργασία απείρων δομών δεδομένων. Προφανώς, δεν μπορεί κανείς να κατασκευάσει ή να αποθηκεύσει ολόκληρες τέτοιες δομές. Μπορεί όμως να κατασκευάζει όλο και μεγαλύτερα τμήματά τους, επαναχρησιμοποιώντας το χώρο που καταλάμβαναν προηγούμενα τμήματα (και τα οποία πλέον δε χρειάζονται στον υπολογισμό).

Παράδειγμα 8.14 Μπορούμε να γράψουμε ένα πρόγραμμα που υπολογίζει τη λίστα των φυσικών αριθμών:

```
nats n = n:nats (n+1)
```

Εάν ζητήσουμε την τιμή του `nats 0`, η Haskell θα αρχίσει να τυπώνει τη λίστα των φυσικών αριθμών. Οταν ένα στοιχείο της λίστας τυπωθεί, ο χώρος μνήμης που καταλάμβανε μπορεί να επαναχρησιμοποιηθεί. □

Πολλές φορές, η χρήση των απείρων δομών δεδομένων διευκολύνει στο να γράφουμε πιό κομψά προγράμματα.

Παράδειγμα 8.15 Μπορούμε να ορίσουμε μια έκφραση που υπολογίζει το άθροισμα των 10 πρώτων φυσικών αριθμών, ως εξής:

```
?sum (take 10 (nats 0))
```

Ο υπολογισμός της παραπάνω έκφρασης είναι δυνατός διότι η `take` δεν επιχειρεί να υπολογίσει την τιμή του `nats 0` πριν αρχίσει την επεξεργασία. □

8.10 Συναρτήσεις Υψηλής Τάξης

Στα παραδείγματα τα οποία έχουμε εξετάσει μέχρι τώρα, τα ορίσματα των συναρτήσεων ήταν απλοί τύποι δεδομένων (πχ. ακέραιοι, λίστες, κλπ). Ενα βασικό χαρακτηριστικό του συναρτησιακού προγραμματισμού είναι ότι και οι ίδιες οι συναρτήσεις μπορούν να περαστούν σαν ορίσματα σε άλλες συναρτήσεις[2].

Παράδειγμα 8.16 Η ακόλουθη συνάρτηση `map` παίρνει σαν πρώτο της όρισμα μια συνάρτηση `f` την οποία και εφαρμόζει σε κάθε στοιχείο του δεύτερου της ορίσματος (το οποίο είναι μια λίστα).

```
map f   []     = []
map f (x:xs) = (f x):(map f xs)
```

Αν τώρα ορίσουμε στο πρόγραμμα μας και τη συνάρτηση `sq a = a*a`, μπορούμε να καλέσουμε τη `map` ως εξής:

```
?map sq [1,3,5]
[1,9,25]
```

Με άλλα λόγια, η `map` εφάρμοσε τη συνάρτηση `sq` σε κάθε στοιχείο της λίστας `[1,3,5]`. □

Ενα βασικό πλεονέκτημα των συναρτήσεων υψηλής τάξης είναι οτι μπορούν να κληθούν με διαφορετικές συναρτήσεις σαν ορίσματα. Με τον τρόπο αυτό, αποφεύγουμε να γράφουμε παρόμοιο κώδικα για λειτουργίες που είναι στην ουσία παρόμοιες.

[2]Μπορεί κανείς να ορίσει συναρτήσεις υψηλής τάξης οι οποίες επιστρέφουν άλλες συναρτήσεις ως αποτέλεσμα. Η δυνατότητα αυτή δε θα μας απασχολήσει στις σημειώσεις αυτές.

Παράδειγμα 8.17 Εστω οτι θέλουμε να ορίσουμε μια συνάρτηση η οποία υπολογίζει τις ρίζες πραγματικών συναρτήσεων στο διάστημα $[a, b]$. Μπορούμε να ορίσουμε μια συνάρτηση υψηλής τάξης `root f a b` η οποία χρησιμοποιεί μια γενική μέθοδο εξεύρεσης ριζών, και η οποία θα μπορεί αργότερα να χρησιμοποιηθεί για τον εντοπισμό των ριζών οποιασδήποτε πραγματικής συνάρτησης στο διάστημα $[a, b]$. □

Παράδειγμα 8.18 Η ακόλουθη συνάρτηση υπολογίζει ένα "συναρτησιακό παραγοντικό":

```
ffact f n = if (n<1) then 1 else (f n) * (ffact f (n-1))
```

Εστω η συνάρτηση `sq a = a*a`. Αν καλέσουμε την `ffact` ως `ffact sq 5` , θα υπολογίσει το γινόμενο `(sq 5)*(sq 4)*...* (sq 1)`. Αν πάλι την καλέσουμε ως `ffact cb 3` όπου `cb b = b*b*b`, θα υπολογίσει την τιμή `(cb 3)*(cb 2)*(cb 1)`. □

Μπορούμε να χρησιμοποιήσουμε συναρτήσεις υψηλής τάξης σε συνδιασμό με άπειρες δομές δεδομένων.

Παράδειγμα 8.19 Η τιμή της έκφρασης:

```
?map sq (nats 0)
```

είναι μια λίστα που περιέχει τα τετράγωνα των φυσικών αριθμών. □

8.11 Τύποι Οριζόμενοι από το Χρήστη

Εκτός από τους συνηθισμένους τύπους δεδομένων, η Haskell επιτρέπει στο χρήστη να ορίσει νέους, πιό ισχυρούς τύπους.

Παράδειγμα 8.20 Η Haskell δεν υποστηρίζει τον τύπο δεδομένων *δυαδικό δέντρο*. Ενα δυαδικό δέντρο είναι, είτε:

- Ενα κενό δέντρο, είτε
- Ενας κόμβος που έχει δύο δυαδικά δέντρα σαν παιδιά.

Ο ορισμός του νέου τύπου δεδομένων στη Haskell μπορεί να γίνει ως εξής:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Ο παραπάνω ορισμός αφορά δέντρα που περιέχουν στοιχεία με τύπο `a`. Μπορούμε τώρα να γράψουμε προγράμματα που επεξεργάζονται το νέο τύπο δεδομένων:

```
elements  Empty        = []
elements (Node v l r) = (elements l)++[v]++(elements r)
```

Η παραπάνω συνάρτηση συγκεντρώνει όλα τα στοιχεία ενός δέντρου σε μια λίστα. Ετσι, η ερώτηση:

```
?elements (Node 5 (Node 7 Empty Empty) (Node 8 Empty Empty))
```

θα επιστρέψει την απάντηση [7,5,8]. Αξίζει να σημειωθεί οτι ο τύπος της συνάρτησης elements είναι Tree a -> [a]. □

Γενικά, ο ορισμός ενός νέου τύπου δεδομένων στη Haskell έχει τη μορφή:

```
data Datatype a1...aN = constr1 | ... | constrM
```

όπου Datatype είναι το όνομα του νέου τύπου, a1,...,aN είναι διαφορετικές μεταβλητές τύπου που αντιπροσωπεύουν τα ορίσματα του τύπου αυτού, και τα constr1,...,constrM δείχνουν πως κατασκευάζεται ο νέος τύπος. Κάθε ένα από τα constr1,...,constrM έχει τη μορφή Name b1...bR (όπου τα b1,...,bR είναι κάποιες από τις μεταβλητές a1,...,aN).

Παράδειγμα 8.21 Μπορούμε να ορίσουμε το νέο τύπο δεδομένων Day που περιγράφει τις μέρες της εβδομάδας:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Μπορούμε τώρα να γράψουμε συναρτήσεις που χρησιμοποιούν τον τύπο αυτό, όπως η ακόλουθη:

```
schedule Sun = "relax"
schedule Sat = "go shopping"
schedule x   = "back to work"
```

Ο τύπος της νέας συνάρτησης schedule είναι Day -> String. □

8.12 Ασκήσεις

1. Ορίστε σε Haskell τη συνάρτηση primesn η οποία δεδομένου ενός φυσικού αριθμού n επιστρέφει τη λίστα των n αρχικών πρώτων αριθμών. (πχ. primesn 6 = [1,2,3,5,7,11]). Σημείωση: Το υπόλοιπο της διαίρεσης στη Haskell συμβολίζεται με mod (πχ. γράφουμε (10 ‘mod‘ 3)).

2. Ορίστε σε Haskell τη συνάρτηση count_vowels η οποία μετράει τον αριθμό των χαρακτήρων που είναι φωνήεντα σε μια λίστα από strings (πχ. count_vowels ["mary","zeus","athens"] = 6).

3. Ορίστε σε Haskell την άπειρη λίστα των αριθμών Hamming σε αύξουσα σειρά και χωρίς επαναλήψεις. Οι αριθμοί Hamming είναι όλοι οι αριθμοί της μορφής $2^i * 3^j * 5^k$. Το πρόγραμμα σας θα πρέπει όταν ο χρήστης ζητάει την τιμή της μεταβλητής `hamming` να του επιστρέφει τη λίστα `[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18,...]`.

4. Γράψτε προγράμματα Haskell που ορίζουν τις ακόλουθες συναρτήσεις:

   - `is_set S` η οποία είναι αληθής όταν η λίστα `S` είναι σύνολο και ψευδής διαφορετικά.
   - `union S1 S2` η οποία επιστρέφει την ένωση των συνόλων `S1` και `S2`.
   - `intersection S1 S2` η οποία επιστρέφει την τομή των συνόλων `S1` και `S2`.
   - `subset S1 S2` η οποία είναι αληθής όταν το σύνολο `S1` είναι υποσύνολο του `S2` και ψευδής διαφορετικά.
   - `equal S1 S2` η οποία είναι αληθής όταν το σύνολο `S1` είναι ίδιο με το `S2` και ψευδής διαφορετικά.

Κεφάλαιο 9

Λάμβδα λογισμός

Ο λ-λογισμός αναπτύχθηκε αρχικά από τον Alonso Church στις αρχές της δεκαετίας του 1930, πολύ πριν αρχίσουν να χρησιμοποιούνται οι ηλεκτρονικοί υπολογιστές. Ήταν μέρος μιας γενικότερης θεωρίας, αντίστοιχης της θεωρίας συνόλων, που αποσκοπούσε στη θεμελίωση των μαθηματικών και της λογικής [Chur32, Chur33]. Παρά το γεγονός ότι η γενική θεωρία ήταν ασυνεπής, όπως αποδείχθηκε αργότερα [Klee35], το τμήμα της που ασχολείται με τις συναρτήσεις βρήκε σημαντικότατες εφαρμογές στην πληροφορική, κυρίως μετά το 1960. Το τμήμα αυτό έγινε γνωστό με το όνομα *λάμβδα-λογισμός* (lambda calculus), ή σε συντομογραφία *λ-λογισμός* (λ-calculus).

Παρά την απλότητά του, είναι αξιοσημείωτο ότι ο λ-λογισμός είναι ένα πλήρες υπολογιστικό μοντέλο. Από τη δεκαετία του 1930 ήταν ήδη γνωστά δυο πολύ σημαντικά αποτελέσματα:

- Όλες οι αναδρομικές συναρτήσεις μπορούν να παρασταθούν στο λ-λογισμό [Klee35].
- Ως υπολογιστικό μοντέλο, ο λ-λογισμός είναι ισοδύναμος με τη μηχανή Turing [Turi37].

Η μηχανή Turing αποτέλεσε τη βάση των *υπολογιστών von Neumann*, στους οποίους ανήκουν οι σημερινοί υπολογιστές, και συγχρόνως οδήγησε στη δημιουργία των πρώτων γλωσσών προστακτικού προγραμματισμού, όπως η FORTRAN και η ALGOL. Από την άλλη πλευρά:

- Ο λ-λογισμός και παραλλαγές αυτού οδήγησαν στο σχεδιασμό νέων αρχιτεκτονικών υπολογιστών. Παραδείγματα τέτοιων υπολογιστών αποτελούν οι *μηχανές αναγωγής* (reduction machines) και οι *υπολογιστές ροής δεδομένων* (data-flow computers), που όταν πρωτοδημιουργήθηκαν ήταν σε θέση να εκτελούν σχεδόν αποκλειστικά προγράμματα γραμμένα σε κάποια διάλεκτο του λ-λογισμού.
- Ο λ-λογισμός αποτέλεσε τη βάση για τη δημιουργία του *συναρτησιακού προγραμματισμού* (functional programming). Εμπνευσμένος από το λ-λογισμό, ο John McCarthy

σχεδίασε τη γλώσσα προγραμματισμού LISP στα τέλη της δεκαετίας του 1950. Αυτή με τη σειρά της έδωσε το έναυσμα για τη δημιουργία πολλών άλλων γλωσσών συναρτησιακού προγραμματισμού, όπως η Scheme, η ML, η Miranda και η Haskell.

Οι υπολογιστές που βασίστηκαν στο λ-λογισμό και ο συναρτησιακός προγραμματισμός δεν έτυχαν της ίδιας ευρείας αποδοχής που γνώρισαν οι υπολογιστές von Neumann και ο προστακτικός προγραμματισμός. Αυτό οφείλεται κυρίως στο γεγονός ότι τόσο οι πρώτες μηχανές αναγωγής όσο και οι πρώτες υλοποιήσεις συναρτησιακών γλωσσών είχαν επιδόσεις σημαντικά χειρότερες από αυτές των ανταγωνιστών τους. Με το πέρασμα του χρόνου, οι μηχανές von Neumann δείχνουν σήμερα να έχουν κυριαρχήσει πλήρως. Από το 1970 όμως, έχουν προταθεί νέες τεχνικές υλοποίησης των συναρτησιακών γλωσσών, όπως η αναγωγή γράφων (graph reduction). Η χρήση τέτοιων τεχνικών στους μεταγλωττιστές των μοντέρνων συναρτησιακών γλωσσών έδωσε μια νέα ώθηση στο συναρτησιακό προγραμματισμό που, από το 1990 και μετά, έχει αποκτήσει αξιόλογη θέση στα ερευνητικά δρώμενα.

Η κύρια όμως συμβολή του λ-λογισμού στην επιστήμη της πληροφορικής δεν έγκειται στη χρήση του ως προγραμματιστικού μοντέλου. Κατά τη δεκαετία του 1960, οι Christopher Strachey, Peter J. Landin και άλλοι παρατήρησαν ότι ο λ-λογισμός είναι ιδιαίτερα πρόσφορος ως συμβολισμός για την περιγραφή σημασιολογικών ιδιοτήτων των γλωσσών προγραμματισμού. Επίσης, παρατήρησαν ότι πολλά προβλήματα σχεδίασης και υλοποίησης των γλωσσών προγραμματισμού, και κυρίως αυτά που σχετίζονται με το μηχανισμό κλήσης υποπρογραμμάτων και τη δομή του συστήματος τύπων, μπορούν να απομονωθούν και να μελετηθούν ευκολότερα στο λ-λογισμό. Οι διαπιστώσεις αυτές και η προσπάθεια για την εύρεση σημασιολογικών μοντέλων του λ-λογισμού, οδήγησαν αργότερα στη διατύπωση της *θεωρίας πεδίων* (domain theory) και στη θεμελίωση του ερευνητικού πεδίου της *σημασιολογίας γλωσσών προγραμματισμού* (programming language semantics).

Στην αρχική του μορφή, όπως διατυπώθηκε από τον Church, ο λ-λογισμός είναι μια θεωρία *χωρίς τύπους*. Κάθε έκφραση θεωρείται ως μια συνάρτηση και μπορεί να εφαρμοστεί σε οποιαδήποτε έκφραση, συμπεριλαμβανομένου και του εαυτού της. Αργότερα, όμως, προτάθηκαν παραλλαγές του λ-λογισμού με τύπους [Curr34, Chur40]. Στη συνέχεια, ο όρος "λ-λογισμός" χωρίς περαιτέρω προσδιορισμούς θα αναφέρεται στη μορφή χωρίς τύπους.

9.1 Μια διαισθητική εισαγωγή

Ο λ-λογισμός είναι μια θεωρία συναρτήσεων. Σε αυτήν, δύο είναι οι κύριες λειτουργίες:

- Η *εφαρμογή* μιας συνάρτησης F πάνω σε ένα όρισμα A, που συμβολίζεται με $F\,A$.
- Η *αφαίρεση*. Έστω ότι x είναι μια μεταβλητή και $E[x]$ είναι μια έκφραση που εξαρτάται από τη μεταβλητή x (δηλαδή περιέχει το x). Τότε η έκφραση $\lambda x.\, E[x]$ συμβολίζει

τη συνάρτηση

$$x \;\mapsto\; E[x]$$

που όταν δέχεται ως όρισμα μια τιμή v επιστρέφει ως αποτέλεσμα την τιμή $E[v]$. Η μεταβλητή x δεν είναι απαραίτητο να εμφανίζεται στην έκφραση $E[x]$. Αν αυτό δε συμβαίνει, τότε η $\lambda x.\, E[x]$ είναι μια σταθερή συνάρτηση.

Για να γίνει αντιληπτός ο τρόπος συνεργασίας της αφαίρεσης και της εφαρμογής, θα χρησιμοποιήσουμε ως παράδειγμα συναρτήσεις από τα μαθηματικά. Η αφαίρεση

$$\lambda x.\, x^2 - 3x + 2$$

συμβολίζει μια (ανώνυμη) συνάρτηση που σε κάθε τιμή x απεικονίζει την τιμή $x^2 - 3x + 2$. Αν αυτή η συνάρτηση εφαρμοστεί στο όρισμα 8, προκύπτει το ακόλουθο αποτέλεσμα:[1]

$$(\lambda x.\, x^2 - 3x + 2)\, 8 \;=\; 8^2 - 3 \cdot 8 + 2 \;=\; 42$$

Κατά την εφαρμογή της συνάρτησης, δηλαδή, η τιμή του ορίσματος 8 αντικαθιστά την παράμετρο x στον τύπο ορισμού της συνάρτησης.

Η αφαίρεση $\lambda x.\, E[x]$ *δεσμεύει* τη μεταβλητή x μέσα στην έκφραση $E[x]$. Μια μεταβλητή που δεν είναι δεσμευμένη ονομάζεται *ελεύθερη*. Για παράδειγμα, στην έκφραση

$$\lambda x.\, x^2 - 3y + 2$$

η μεταβλητή x είναι δεσμευμένη ενώ η y είναι ελεύθερη. Είναι όμως δυνατό σε μια έκφραση κάποια εμφάνιση μιας μεταβλητής να είναι δεσμευμένη και κάποια άλλη εμφάνιση της ίδιας μεταβλητής να είναι ελεύθερη. Για παράδειγμα, στην έκφραση

$$(\lambda x.\, x^2 - 3y + 2)\,(4x + 1)$$

η πρώτη εμφάνιση του x (στο x^2) είναι δεσμευμένη, γιατί βρίσκεται στο εσωτερικό της αφαίρεσης, ενώ η δεύτερη εμφάνιση του x (στο $4x$) είναι ελεύθερη.

Παρόμοια έννοια δέσμευσης συναντάται και στα μαθηματικά. Για παράδειγμα, στο ολοκλήρωμα

$$\int_{-\pi}^{\pi} \frac{\sin x + \cos y}{\cos x - \sin y}\, dx$$

[1] Ας σημειωθεί ότι οι παρενθέσεις χρησιμοποιούνται μόνο για την αποφυγή παρερμηνειών στη δομή των εκφράσεων. Αντίθετα με τα μαθηματικά, όπου η εφαρμογή μιας συνάρτησης f σε ένα όρισμα a συμβολίζεται συνήθως με $f(a)$, η εφαρμογή στο λ-λογισμό συμβολίζεται χωρίς τη χρήση παρενθέσεων.

η μεταβλητή x στο εσωτερικό του ολοκληρώματος είναι δεσμευμένη και, κατά συνέπεια, η τιμή του ολοκληρώματος δεν εξαρτάται από την τιμή του x έξω από αυτό. Αντίθετα, η μεταβλητή y είναι ελεύθερη και άρα η τιμή του ολοκληρώματος εξαρτάται από την τιμή που έχει το y έξω από αυτό.

Στο λ-λογισμό, όπως άλλωστε και στα μαθηματικά, ιδιαίτερη θέση έχει η έννοια της αντικατάστασης μιας μεταβλητής με κάποια έκφραση. Όπως ήδη αναφέρθηκε, η αντικατάσταση χρησιμοποιείται στην εφαρμογή συναρτήσεων: στην εφαρμογή $(\lambda x.\, E[x])\, A$ το όρισμα A αντικαθιστά τις εμφανίσεις της μεταβλητής x στην έκφραση $E[x]$. Η δέσμευση μεταβλητών όμως είναι δυνατό να δημιουργήσει προβλήματα κατά την αντικατάσταση, όπως φαίνεται στα επόμενα παραδείγματα από τα μαθηματικά:

- Σε μια έκφραση δεν πρέπει να επιτρέπεται η αντικατάσταση μεταβλητών που είναι δεσμευμένες. Για παράδειγμα, στο παραπάνω ολοκλήρωμα, η (εσφαλμένη) αντικατάσταση της μεταβλητής x με την τιμή 42 θα οδηγούσε στο παράλογο αποτέλεσμα

$$\int_{-\pi}^{\pi} \frac{\sin 42 + \cos y}{\cos 42 - \sin y}\, d42$$

- Η αντικατάσταση της μεταβλητής x με μια έκφραση A δεν πρέπει να προκαλεί τη δέσμευση μεταβλητών της A που ήταν προηγουμένως ελεύθερες. Για παράδειγμα, στο προηγούμενο ολοκλήρωμα, έστω ότι θέλουμε να αντικαταστήσουμε την ελεύθερη μεταβλητή y με την έκφραση $3x + 1$. Στην έκφραση αυτή το x είναι ελεύθερο, θα μπορούσε π.χ. να έχει την τιμή 4, και σε αυτή την περίπτωση η έκφραση $3x + 1$ θα είχε την τιμή 13. Η (εσφαλμένη) όμως αντικατάσταση του y με αυτή την έκφραση θα είχε ως αποτέλεσμα το ολοκλήρωμα

$$\int_{-\pi}^{\pi} \frac{\sin x + \cos(3x+1)}{\cos x - \sin(3x+1)}\, dx$$

  στο οποίο όλες οι εμφανίσεις της μεταβλητής x είναι δεσμευμένες. Στο εσωτερικό του ολοκληρώματος, η έκφραση $3x + 1$ παύει να έχει την τιμή 13: η τιμή της τώρα μεταβάλλεται καθώς το x παίρνει τιμές στο διάστημα $-\pi$ έως π. Κατά συνέπεια, το τελικό αποτέλεσμα δε θα είναι το αναμενόμενο.

9.2 Λάμβδα όροι

Ο λ-λογισμός είναι μια τυπική γλώσσα Λ, η σύνταξη της οποίας δίνεται από τον ακόλουθο επαγωγικό ορισμό:

Ορισμός 9.1 *Έστω V ένα αριθμήσιμο σύνολο μεταβλητών. Το σύνολο Λ των* όρων του λ-λογισμού *είναι το μικρότερο σύνολο που ικανοποιεί τις παρακάτω ιδιότητες:*

$$\begin{array}{lcl} x \in V & \Rightarrow & x \in \Lambda \\ M, N \in \Lambda & \Rightarrow & (M\,N) \in \Lambda \\ x \in V, M \in \Lambda & \Rightarrow & (\lambda x.\, M) \in \Lambda \end{array}$$

Τα στοιχεία του συνόλου Λ *ονομάζονται επίσης* λ-όροι *(λ-terms) ή* λ-εκφράσεις *(λ-expressions).*

Από τον ορισμό 9.1 προκύπτει ότι οι υπάρχουν τριων ειδών λ-όροι:

1. *Μεταβλητές* (variables), δηλαδή στοιχεία του συνόλου V.
2. *Εφαρμογές* (applications), που έχουν τη μορφή $(M\,N)$, όπου M και N είναι λ-όροι.
3. *Αφαιρέσεις* (abstractions), που έχουν τη μορφή $(\lambda x.\, M)$, όπου x μια μεταβλητή και M ένας λ-όρος.

Κατά σύμβαση θα χρησιμοποιούμε μικρά γράμματα του λατινικού αλφαβήτου, (x, y, z, κ.λπ.) για να συμβολίσουμε μεταβλητές και κεφαλαία γράμματα του λατινικού αλφαβήτου (M, N, F, G, P, Q, κ.λπ.) για να συμβολίσουμε λ-όρους.

Χρησιμοποιώντας αφηρημένη σύνταξη σε μορφή BNF και θεωρώντας ότι η συντακτική κλάση των μεταβλητών παριστάνεται με το μη-τερματικό σύμβολο ⟨var⟩, η γλώσσα Λ των λ-όρων περιγράφεται ισοδύναμα ως εξής:

$$\begin{array}{lcl} \langle \text{term} \rangle & ::= & \langle \text{var} \rangle \\ & | & (\ \langle \text{term} \rangle\ \langle \text{term} \rangle\ ) \\ & | & (\ \lambda\ \langle \text{var} \rangle\ .\ \langle \text{term} \rangle\ ) \end{array}$$

Παράδειγμα 9.1 Οι παρακάτω είναι λ-όροι:

$$\begin{array}{l} (x\,y) \\ (\lambda x.\, x) \\ (\lambda x.\, (\lambda y.\, (x\,y))) \\ (((\lambda x.\, x)\, y)\, (\lambda x.\, z)) \\ ((\lambda x.\, (\lambda y.\, z))\, (\lambda x.\, x)) \\ (\lambda x.\, ((\lambda y.\, y)\, (\lambda z.\, x))) \end{array}$$

□

Παρότι η διαισθητική ερμηνεία των λ-όρων εξηγήθηκε στην ενότητα 9.1, η σύνταξη του λ-λογισμού όπως δίνεται στον ορισμό 9.1 δεν αποδίδει κανένα ιδιαίτερο νόημα στους λ-όρους, που μπορούν να θεωρούνται ως αυθαίρετες συμβολικές εκφράσεις.

Για την απλοποίηση των λ-όρων και την αποφυγή του μεγάλου αριθμού παρενθέσεων που συνεπάγεται η αυστηρή τήρηση του ορισμού 9.1, θα χρησιμοποιούμε στη συνέχεια τις ακόλουθες συμβάσεις:

1. Οι εξωτερικές παρενθέσεις δε γράφονται:

$$\lambda x.\, x \ \text{ είναι συντομογραφία του } \ (\lambda x.\, x)$$

2. Η εφαρμογή είναι αριστερά προσεταιριστική:

$$F\, M_1\, M_2\, \ldots\, M_n \ \text{ είναι συντομογραφία του } \ (\ldots((F\, M_1)\, M_2)\, \ldots\, M_n)$$

3. Η αφαίρεση εκτείνεται όσο περισσότερο είναι δυνατό, δηλαδή ως το επόμενο κλείσιμο παρένθεσης ή το τέλος του όρου:

$$\lambda x.\ M_1\, M_2\, \ldots\, M_n \ \text{ είναι συντομογραφία του } \ \lambda x.\, (M_1\, M_2\, \ldots\, M_n)$$

Παράδειγμα 9.2 Ακολουθώντας τις παραπάνω συμβάσεις, οι λ-όροι του παραδείγματος 9.1 γράφονται σε απλοποιημένη μορφή ως εξής:

$x\, y$
$\lambda x.\, x$
$\lambda x.\, \lambda y.\, x\, y$
$((\lambda x.\, x)\, y)\, (\lambda x.\, z)$
$(\lambda x.\, \lambda y.\, z)\, (\lambda x.\, x)$
$\lambda x.\, (\lambda y.\, y)\, (\lambda z.\, x)$

□

Μια από τις σχέσεις που χρησιμοποιείται για τη σύγκριση μεταξύ λ-όρων είναι η σχέση $\equiv$ που υποδηλώνει ότι δυο όροι είναι πανομοιότυποι. Πρόκειται ουσιαστικά για τη βασική σχέση ισότητας μεταξύ των στοιχείων του συνόλου Λ.

Ορισμός 9.2 *Η σχέση ταυτότητας $\equiv$ στο σύνολο των λ-όρων ορίζεται επαγωγικά ως εξής:*

$$\begin{array}{lcll} x & \equiv & y & \text{αν } x = y \\ (M\, N) & \equiv & (P\, Q) & \text{αν } M \equiv P \text{ και } N \equiv Q \\ (\lambda x.\, M) & \equiv & (\lambda y.\, N) & \text{αν } x = y \text{ και } M \equiv N \end{array}$$

όπου με $x = y$ συμβολίζεται η σχέση ισότητας στο σύνολο V (δηλαδή x και y είναι η ίδια μεταβλητή). Αν $M \equiv N$, οι όροι $M, N \in \Lambda$ ονομάζονται ταυτόσημοι *(identical).*

Η μεταβλητή x στην αφαίρεση $\lambda x.\, M$ ονομάζεται *δεσμεύουσα μεταβλητή* (binding variable). Η *εμβέλεια* (scope) της αφαίρεσης λx είναι ο λ-όρος M, εκτός από τυχόν αφαιρέσεις που αυτός περιέχει και στις οποίες η δεσμεύουσα μεταβλητή είναι πάλι η x. Εμφανίσεις της μεταβλητής x που βρίσκονται στην εμβέλεια κάποιου λx ονομάζονται *δεσμευμένες* (bound). Αντίθετα, εμφανίσεις που δε βρίσκονται στην εμβέλεια κανενός λx ονομάζονται *ελεύθερες* (free). Τα παραπάνω συνοψίζονται στον ακόλουθο ορισμό.

Ορισμός 9.3 *Το σύνολο των* ελεύθερων μεταβλητών *(free variables) ενός λ-όρου $M \in \Lambda$ συμβολίζεται με $\boldsymbol{FV}(M)$ και ορίζεται επαγωγικά ως εξής:*

$$\begin{array}{lcl} \boldsymbol{FV}(x) & = & \{\,x\,\} \\ \boldsymbol{FV}(M\,N) & = & \boldsymbol{FV}(M) \cup \boldsymbol{FV}(N) \\ \boldsymbol{FV}(\lambda x.\,M) & = & \boldsymbol{FV}(M) - \{\,x\,\} \end{array}$$

Παράδειγμα 9.3 Έστω ο παρακάτω όρος:

$$M \;\equiv\; (\lambda x.\,y\,x)\,(\lambda y.\,x\,y)$$

ελεύθερη (το y στο $\lambda x.\,y\,x$), δεσμευμένη (το x στο $\lambda x.\,y\,x$), ελεύθερη (το x στο $\lambda y.\,x\,y$), δεσμευμένη (το y στο $\lambda y.\,x\,y$)

Ακολουθώντας τον ορισμό 9.3 έχουμε:

$$\begin{array}{rcl} \boldsymbol{FV}(M) & = & \boldsymbol{FV}((\lambda x.\,y\,x)(\lambda y.\,x\,y)) \\ & = & \boldsymbol{FV}(\lambda x.\,y\,x) \cup \boldsymbol{FV}(\lambda y.\,x\,y) \\ & = & (\boldsymbol{FV}(y\,x) - \{\,x\,\}) \cup (\boldsymbol{FV}(x\,y) - \{\,y\,\}) \\ & = & ((\boldsymbol{FV}(y) \cup \boldsymbol{FV}(x)) - \{\,x\,\}) \cup ((\boldsymbol{FV}(x) \cup \boldsymbol{FV}(y)) - \{\,y\,\}) \\ & = & ((\{\,y\,\} \cup \{\,x\,\}) - \{\,x\,\}) \cup ((\{\,x\,\} \cup \{\,y\,\}) - \{\,y\,\}) \\ & = & (\{\,x,y\,\} - \{\,x\,\}) \cup (\{\,x,y\,\} - \{\,y\,\}) \\ & = & \{\,y\,\} \cup \{\,x\,\} \\ & = & \{\,x,y\,\} \end{array}$$

□

Ορισμός 9.4 *Ένας λ-όρος $M \in \Lambda$ ονομάζεται* κλειστός λ-όρος *(closed λ-term ή combinator), αν $\boldsymbol{FV}(M) = \emptyset$. Το σύνολο των κλειστών λ-όρων συμβολίζεται με Λ^0.*

Παράδειγμα 9.4 Οι ακόλουθοι λ-όροι είναι κλειστοί και στη βιβλιογραφία ονομάζονται συχνά *πρότυποι κλειστοί όροι* (standard combinators).

$$\begin{array}{lcl} \mathsf{I} & \equiv & \lambda x.\,x \\ \mathsf{K} & \equiv & \lambda x.\,\lambda y.\,x \\ \mathsf{K}_* & \equiv & \lambda x.\,\lambda y.\,y \\ \mathsf{S} & \equiv & \lambda x.\,\lambda y.\,\lambda z.\,(x\,z)\,(y\,z) \end{array}$$

□

9.3 Αντικατάσταση

Η αντικατάσταση είναι η βασική πράξη συμβολικής επεξεργασίας των λ-όρων και επηρεάζει τις ελεύθερες μεταβλητές. Ο συμβολισμός $M[x := N]$ παριστάνει το αποτέλεσμα της αντικατάστασης στον όρο M όλων των ελεύθερων εμφανίσεων της μεταβλητής x με τον όρο N.

Ορισμός 9.5 *Η αντικατάσταση (substitution) της μεταβλητής $x \in V$ στον όρο $M \in \Lambda$ με τον όρο $N \in \Lambda$ συμβολίζεται με $M[x := N]$ και ορίζεται επαγωγικά ως εξής:*

$$
\begin{array}{lcll}
x[x := N] & \equiv & N & \\
y[x := N] & \equiv & y & \text{αν } y \neq x \\
(P\,Q)[x := N] & \equiv & P[x := N]\,Q[x := N] & \\
(\lambda x.\,P)[x := N] & \equiv & \lambda x.\,P & \\
(\lambda y.\,P)[x := N] & \equiv & \lambda y.\,P[x := N] & \text{αν } y \neq x, \text{ και } (y \notin \boldsymbol{FV}(N) \text{ ή } x \notin \boldsymbol{FV}(P)) \\
(\lambda y.\,P)[x := N] & \equiv & \lambda z.\,P[y := z][x := N] & \text{αν } y \neq x, \text{ και } y \in \boldsymbol{FV}(N) \text{ και } x \in \boldsymbol{FV}(P), \\
& & & \text{όπου } z \notin \boldsymbol{FV}(P) \cup \boldsymbol{FV}(N)
\end{array}
$$

Από τον παραπάνω ορισμό της αντικατάστασης εξασφαλίζεται αφενός ότι αντικαθίστανται μόνο οι ελεύθερες εμφανίσεις μιας μεταβλητής, αφετέρου ότι στο αποτέλεσμα της αντικατάστασης καμία από τις ελεύθερες μεταβλητές του όρου N δε θα έχει γίνει δεσμευμένη. Ειδικότερα για το δεύτερο, από τον ορισμό 9.5 διαπιστώνει κανείς ότι δέσμευση μιας ελεύθερης μεταβλητής του όρου N θα μπορούσε να γίνει μόνο μέσω του τελευταίου κανόνα, στην περίπτωση που $y \in \boldsymbol{FV}(N)$ και $x \in \boldsymbol{FV}(P)$. Ο κανόνας όμως αυτός μεριμνά για τη μετονομασία της δεσμεύουσας μεταβλητής, προκειμένου να αποφευχθεί αυτό το ενδεχόμενο.[2]

Στον τελευταίο κανόνα του ορισμού 9.5 η επιλογή της μεταβλητής $z \in V$ είναι ελεύθερη, υπό την προϋπόθεση να ισχύει $z \notin \boldsymbol{FV}(P) \cup \boldsymbol{FV}(N)$. Για να οριστεί όμως με μονοσήμαντο τρόπο το αποτέλεσμα της αντικατάστασης $M[x := N]$, πρέπει σε αυτό τον κανόνα να καθορίζεται ακριβώς πώς γίνεται η επιλογή αυτής της μεταβλητής. Για το λόγο αυτό, θεωρούμε ότι τα στοιχεία του συνόλου V είναι αριθμημένα και επιλέγουμε ως z το πρώτο στοιχείο που ικανοποιεί τη σχέση $z \notin \boldsymbol{FV}(P) \cup \boldsymbol{FV}(N)$.

9.4 Μετατροπές

Στο λ-λογισμό υπάρχουν τρία είδη *μετατροπής* (conversion), που παραδοσιακά συμβολίζονται με τα τρία γράμματα α, β και η. Κάθε είδος μετατροπής περιγράφεται από έναν κανόνα της μορφής

$$M \quad \to_\chi \quad N \qquad (\chi)$$

όπου $\chi \in \{\,\alpha, \beta, \eta\,\}$ και $M, N \in \Lambda$. Ο όρος M ονομάζεται *χ-redex*, ή απλά *redex*, και ο όρος N ονομάζεται *contractum*. Για παράδειγμα, στον κανόνα της β-μετατροπής, ο όρος

[2] Ο ορισμός 9.5 σε αυτή τη μορφή προέρχεται από την εργασία [Curr58] και επαναλαμβάνεται στην [Hind86].

M που μετατρέπεται ονομάζεται β-redex. Οι τρεις κανόνες μετατροπής ορίζουν τους τρεις νόμιμους τρόπους "ισοδύναμου" μετασχηματισμού των λ-όρων και, κατ' αυτό τον τρόπο, περιγράφουν τη σημασία αυτών των όρων, την οποία εξηγήσαμε διαισθητικά στην ενότητα 9.1.

Πριν προχωρήσουμε στον ορισμό των τριων κανόνων μετατροπής είναι σκόπιμο να ορίσουμε μερικές χρήσιμες ιδιότητες σχέσεων μεταξύ λ-όρων. Η πρώτη από αυτές είναι η ιδιότητα της συμβατότητας, που δηλώνει μια μορφή κλειστότητας ως προς την κατασκευή των λ-όρων.

Ορισμός 9.6 *Μια σχέση $\sim$ πάνω στο Λ ονομάζεται* συμβατή *(compatible) αν για κάθε $x \in V$ και για κάθε $M, N, P \in \Lambda$ ισχύουν οι παρακάτω ιδιότητες:*

$$
\begin{aligned}
M \sim N \;&\Rightarrow\; (M\,P) \sim (N\,P) \\
M \sim N \;&\Rightarrow\; (P\,M) \sim (P\,N) \\
M \sim N \;&\Rightarrow\; (\lambda x.\, M) \sim (\lambda x.\, N)
\end{aligned}
$$

Εύκολα διαπιστώνει κανείς ότι η σχέση ταυτότητας $\equiv$ μεταξύ των λ-όρων είναι συμβατή.

Βάσει της συμβατότητας, μπορούν να οριστούν δυο ακόμα ιδιότητες σχέσεων μεταξύ λ-όρων που θα είναι χρήσιμες στη συνέχεια.

Ορισμός 9.7 *Μια* σχέση αναγωγής *(reduction relation) είναι μια συμβατή, ανακλαστική και μεταβατική σχέση πάνω στο Λ.*

Ορισμός 9.8 *Μια* σχέση συμφωνίας *(congruence relation) είναι μια συμβατή σχέση ισοδυναμίας πάνω στο Λ.*[3]

9.4.1 α-μετατροπή

Σύμφωνα με το συνήθη συμβολισμό του συναρτησιακού λογισμού στα μαθηματικά, τα ονόματα των παραμέτρων μιας συνάρτησης δεν έχουν ιδιαίτερη σημασία: η συνάρτηση $f(x) = 3x + 1$ είναι ταυτόσημη με τη συνάρτηση $f(y) = 3y + 1$. Αν θεωρήσουμε ότι οι λ-όροι συμβολίζουν συναρτήσεις, αντιλαμβάνεται κανείς ότι η επιλογή μιας συγκεκριμένης δεσμεύουσας μεταβλητής x στην αφαίρεση $\lambda x.\, M$ δεν είναι ιδιαίτερα σημαντική. Ο κανόνας της α-μετατροπής επιτρέπει τη μετονομασία της δεσμεύουσας μεταβλητής σε μια αφαίρεση.

Ορισμός 9.9 *Η σχέση $\rightarrow_\alpha$ ορίζεται ως η μικρότερη συμβατή σχέση πάνω στο Λ για την οποία για κάθε $M \in \Lambda$ και για κάθε $x, y \in V$ τέτοια ώστε $y \notin \boldsymbol{FV}(M)$, ισχύει*

$$\lambda x.\, M \quad \rightarrow_\alpha \quad \lambda y.\, M[x := y] \qquad (\alpha)$$

[3] Υπενθυμίζεται ότι μια σχέση ονομάζεται *σχέση ισοδυναμίας* (equivalence relation) αν είναι ανακλαστική, μεταβατική και συμμετρική.

Ο περιορισμός $y \notin \boldsymbol{FV}(M)$ εξασφαλίζει ότι η μετονομασία της δεσμεύουσας μεταβλητής δεν προκαλεί τη δέσμευση μεταβλητών του όρου M που ήταν αρχικά ελεύθερες.

Παράδειγμα 9.5 Οι παρακάτω α-μετατροπές είναι σωστές:

$$\begin{array}{ll} \lambda x.\, x & \to_\alpha\ \lambda y.\, y \\ \lambda x.\, z\, x & \to_\alpha\ \lambda y.\, z\, y \\ \lambda x.\, \lambda y.\, z\, x\, y & \to_\alpha\ \lambda y.\, \lambda w.\, z\, y\, w \end{array}$$

Παρατηρούμε ότι στην τρίτη μετατροπή, η αντικατάσταση $(\lambda y.\, z\, x\, y)[x := y]$ προκάλεσε τη μετονομασία της δεσμεύουσας μεταβλητής, προκειμένου να μη δεσμευθεί στο αποτέλεσμα της αντικατάστασης η ελεύθερη εμφάνιση της y.

Αντίθετα η παρακάτω μετατροπή είναι λανθασμένη:

$$\lambda x.\, \lambda z.\, z\, x\, y \ \not\to_\alpha\ \lambda y.\, \lambda z.\, z\, y\, y$$

γιατί δεν πληρούται ο περιορισμός $y \notin \boldsymbol{FV}(\lambda z.\, z\, x\, y)$. □

9.4.2 β-μετατροπή

Όπως ήδη αναφέρθηκε στην ενότητα 9.1, ένας όρος της μορφής $(\lambda x.\, M)\, N$ συμβολίζει την εφαρμογή της συνάρτησης $\lambda x.\, M$ στο όρισμα N. Ο υπολογισμός του αποτελέσματος αυτής της εφαρμογής γίνεται αντικαθιστώντας την παράμετρο x με το όρισμα N στον όρο M. Με άλλα λόγια, σε αυτό τον υπολογισμό, ο όρος $(\lambda x.\, M)\, N$ μετατρέπεται στον όρο $M[x := N]$. Ο κανόνας της β-μετατροπής περιγράφει ακριβώς αυτή τη διαδικασία.

Ορισμός 9.10 *Η σχέση $\to_\beta$ ορίζεται ως η μικρότερη συμβατή σχέση πάνω στο Λ για την οποία για κάθε $M, N \in \Lambda$ και για κάθε $x \in V$ ισχύει*

$$(\lambda x.\, M)\, N \quad \to_\beta \quad M[x := N] \qquad (\beta)$$

Παράδειγμα 9.6 Οι παρακάτω β-μετατροπές είναι σωστές:

$$\begin{array}{ll} (\lambda x.\, z\, x)\, w & \to_\beta\ z\, w \\ (\lambda x.\, \lambda y.\, z\, x\, y)\, w & \to_\beta\ \lambda y.\, z\, w\, y \\ (\lambda y.\, z\, y\, (\lambda x.\, x\, y))\, w & \to_\beta\ z\, w\, (\lambda x.\, x\, w) \end{array}$$

ενώ αντίθετα η επόμενη μετατροπή είναι λανθασμένη:

$$(\lambda x.\, \lambda y.\, z\, x\, y)\, (w\, y) \ \not\to_\beta\ \lambda y.\, z\, (w\, y)\, y$$

γιατί για να γίνει η αντικατάσταση $(\lambda y.\, z\, x\, y)[x := w\, y]$ πρέπει να μετονομαστεί η δεσμεύουσα μεταβλητή y, διαφορετικά η εμφάνιση της y στον όρο $w\, y$ θα δεσμευόταν. Το σωστό αποτέλεσμα της παραπάνω β-μετατροπής είναι

$$(\lambda x.\, \lambda y.\, z\, x\, y)\, (w\, y) \ \to_\beta\ \lambda t.\, z\, (w\, y)\, t$$

όπου t μια νέα μεταβλητή. □

9.4.3 η-μετατροπή

Σύμφωνα με το συνήθη ορισμό της ισότητας συναρτήσεων στα μαθηματικά, δυο συναρτήσεις είναι ίσες αν και μόνο αν επιστρέφουν το ίδιο αποτέλεσμα, όταν εφαρμοστούν στο ίδιο όρισμα. Η αρχή αυτή εκφράζεται από τον κανόνα της η-μετατροπής στο λ-λογισμό.

Έστω η συνάρτηση που παριστάνεται από τον λ-όρο $(\lambda x.\, M\, x)$, όπου η μεταβλητή x δεν εμφανίζεται ελεύθερη στον όρο M. Σύμφωνα με τον ορισμό 9.10, το αποτέλεσμα της εφαρμογής αυτής της συνάρτησης σε ένα τυχαίο όρισμα N β-μετατρέπεται σε $M\, N$. Από την άλλη πλευρά, αν ο όρος M θεωρηθεί ως μια συνάρτηση, τότε το αποτέλεσμα της εφαρμογής του στο όρισμα N είναι ίσο με $M\, N$. Επομένως, οι συναρτήσεις $(\lambda x.\, M\, x)$ και M είναι ίσες και είναι εύλογο να θεωρήσει κανείς ότι η δεύτερη προκύπτει από την απλοποίηση της πρώτης.

Ορισμός 9.11 *Η σχέση $\rightarrow_\eta$ ορίζεται ως η μικρότερη συμβατή σχέση πάνω στο Λ για την οποία για κάθε $M \in \Lambda$ και για κάθε $x \in V$ τέτοια ώστε $x \notin \boldsymbol{FV}(M)$, ισχύει*

$$\lambda x.\, M\, x \quad \rightarrow_\eta \quad M \qquad (\eta)$$

Παράδειγμα 9.7 Οι παρακάτω η-μετατροπές είναι σωστές:

$$\begin{array}{lll} \lambda x.\, z\, x & \rightarrow_\eta & z \\ \lambda y.\, z\, x\, y & \rightarrow_\eta & z\, x \end{array}$$

Δεν ισχύει όμως το ίδιο για την επόμενη μετατροπή:

$$\lambda x.\, z\, x\, x \;\not\rightarrow_\eta\; z\, x$$

γιατί δεν πληροίται ο περιορισμός $x \notin \boldsymbol{FV}(z\, x)$. □

9.4.4 Μετατροπή, αναγωγή και ισότητα

Οι τρεις κανόνες μετατροπής (α), (β) και (η) που περιγράφηκαν στις προηγούμενες ενότητες ορίζουν τις τρεις βασικές σχέσεις μετατροπής $\rightarrow_\alpha$, $\rightarrow_\beta$ και $\rightarrow_\eta$ για το λ-λογισμό. Η συνολική σχέση μετατροπής $M \rightarrow N$, που ορίζεται παρακάτω, υποδηλώνει ότι ο όρος M μετατρέπεται *σε ένα βήμα* στον όρο N με κάποιον από αυτούς τους τρεις κανόνες.

Ορισμός 9.12 *Με $\rightarrow$ συμβολίζουμε την ένωση των σχέσεων $\rightarrow_\alpha$, $\rightarrow_\beta$ και $\rightarrow_\eta$. Δηλαδή, η σχέση $\rightarrow$ ορίζεται ως η μικρότερη σχέση για την οποία ισχύουν:*

$$\begin{array}{lll} M \rightarrow_\alpha N & \Rightarrow & M \rightarrow N \\ M \rightarrow_\beta N & \Rightarrow & M \rightarrow N \\ M \rightarrow_\eta N & \Rightarrow & M \rightarrow N \end{array}$$

Συχνά είναι χρήσιμο να αναφέρεται κανείς σε μεγαλύτερες ακολουθίες βημάτων μετατροπής με τη γενική μορφή

$$M \;\equiv\; N_0 \;\rightarrow\; N_1 \;\rightarrow\; N_2 \;\rightarrow\; \ldots \;\rightarrow\; N_{n-1} \;\rightarrow\; N_n \;\equiv\; N$$

Στην περίπτωση αυτή, ο όρος M μετατρέπεται στον όρο N *με μια ακολουθία βημάτων*, πιθανώς κενή. Η σχέση αυτή συμβολίζεται με $M \twoheadrightarrow N$. Αν $n = 1$ τότε η μετατροπή γίνεται σε ένα βήμα $M \rightarrow N$ και άρα η σχέση $\twoheadrightarrow$ εμπεριέχει την $\rightarrow$. Επίσης, αν $n = 0$ τότε η ακολουθία είναι κενή και δε συμβαίνει μετατροπή, δηλαδή $M \equiv N$ και άρα η σχέση $\twoheadrightarrow$ εμπεριέχει την $\equiv$.

Ορισμός 9.13 *Με $\twoheadrightarrow$ συμβολίζουμε το ανακλαστικό και μεταβατικό κλείσιμο της σχέσης $\rightarrow$. Δηλαδή, η σχέση $\twoheadrightarrow$ ορίζεται ως η μικρότερη σχέση για την οποία ισχύουν:*

$$\begin{array}{lcl} M \rightarrow N & \Rightarrow & M \twoheadrightarrow N \\ & & M \twoheadrightarrow M \\ M \twoheadrightarrow N, N \twoheadrightarrow P & \Rightarrow & M \twoheadrightarrow P \end{array}$$

Εύκολα διαπιστώνει κανείς ότι η σχέση $\twoheadrightarrow$ είναι σχέση αναγωγής (βλ. ορισμό 9.7). Όταν ισχύει $M \twoheadrightarrow N$, λέμε ότι ο όρος M *ανάγεται* στον όρο N.

Όταν $M \twoheadrightarrow N$, όλα τα βήματα στην ακολουθία μετατροπής έχουν την ίδια κατεύθυνση, από αριστερά προς τα δεξιά. Αν σε αυτή την ακολουθία επιτρέψουμε και την ύπαρξη "αντίστροφων" βημάτων, όπως π.χ. στην περίπτωση

$$M \;\equiv\; N_0 \;\rightarrow\; N_1 \;\leftarrow\; N_2 \;\rightarrow\; N_3 \;\leftarrow\; N_4 \;\leftarrow\; N_5 \;\rightarrow\; N_6 \;\equiv\; N$$

τότε οδηγούμαστε σε μια γενικότερη σχέση την οποία συμβολίζουμε με $=$. Η σχέση αυτή, όπως και η $\twoheadrightarrow$, εμπεριέχει τις $\rightarrow$ και $\equiv$, αλλά εμπεριέχει και την ίδια την $\twoheadrightarrow$.

Ορισμός 9.14 *Με $=$ συμβολίζουμε τη σχέση ισοδυναμίας που προκύπτει από την $\twoheadrightarrow$. Δηλαδή, η σχέση $=$ ορίζεται ως η μικρότερη σχέση για την οποία ισχύουν:*

$$\begin{array}{lcl} M \twoheadrightarrow N & \Rightarrow & M = N \\ M = N & \Rightarrow & N = M \\ M = N, N = P & \Rightarrow & M = P \end{array}$$

Εύκολα διαπιστώνει κανείς ότι η σχέση $=$ είναι σχέση συμφωνίας (βλ. ορισμό 9.8). Όταν ισχύει $M = N$, λέμε ότι ο όρος M *είναι ίσος* με τον όρο N.

Στο σημείο αυτό πρέπει να τονισθεί ότι με τη σχέση ισότητας $M = N$ δεν εννοείται ότι οι δύο όροι ταυτίζονται, κάτι που σύμφωνα με τον ορισμό 9.2 συμβολίζεται ως $M \equiv N$. Η διάκριση αυτών των δυο σχέσεων και η επιλογή του συμβόλου $=$ για τη σχέση συμφωνίας του ορισμού 9.14 μπορεί να μπερδέψει προς στιγμήν τον αναγνώστη. Η διαισθητική αιτιολόγηση αυτής της επιλογής θα πρέπει να περιμένει μέχρι την πρόταση 9.2 στην ενότητα 9.6.

Τόσο η σχέση αναγωγής $\twoheadrightarrow$ όσο και η σχέση ισότητας $=$ προέκυψαν ως επεκτάσεις της συνολικής σχέσης μετατροπής $\rightarrow$. Παρόμοιες σχέσεις μπορούν να ορισθούν για τις επιμέρους σχέσεις μετατροπής. Για παράδειγμα, η σχέση $\twoheadrightarrow_\beta$ ορίζεται ως το ανακλαστικό και μεταβατικό κλείσιμο της σχέσης $\rightarrow_\beta$ και η σχέση $=_\alpha$ ορίζεται ως η σχέση ισοδυναμίας που προκύπτει από την $\twoheadrightarrow_\alpha$.

9.5 Κανονικές μορφές

Θεωρώντας τον λ-λογισμό ως ένα υπολογιστικό μοντέλο επεξεργασίας συναρτήσεων, οι διάφορες σχέσεις μετατροπής παριστάνουν την πραγματοποίηση βημάτων επεξεργασίας πάνω στους λ-όρους, που γενικά αποσκοπούν στον υπολογισμό κάποιου αποτελέσματος. Πιο συγκεκριμένα:

- Η β-μετατροπή είναι η κατ' εξοχήν υπολογιστική πράξη στο λ-λογισμό, αφού διαισθητικά παριστάνει την εφαρμογή μιας συνάρτησης σε κάποιο όρισμα.

- Η α-μετατροπή δεν προάγει τη διαδικασία του υπολογισμού. Εφόσον τα ονόματα των δεσμευμένων μεταβλητών δεν έχουν ουσιαστική σημασία, η μετονομασία των μεταβλητών που επιτελείται μέσω της α-μετατροπής δεν αποτελεί ουσιαστικό υπολογιστικό βήμα.[4]

- Η η-μετατροπή, παρότι δε συμβάλλει άμεσα στην υπολογιστική διαδικασία, υλοποιεί ένα μηχανισμό για την απλοποίηση λ-όρων που παριστάνουν συναρτήσεις.[5]

Με όλα αυτά κατά νου, εύκολα διαπιστώνει κανείς ότι η σχέση μετατροπής $M \rightarrow N$ παριστάνει ένα βήμα στη διαδικασία υπολογισμού, ενώ η σχέση $M \twoheadrightarrow N$ παριστάνει μια (πιθανώς κενή) ακολουθία από τέτοια βήματα. Όταν $M \twoheadrightarrow N$, μπορεί να θεωρηθεί ότι ο όρος N έχει προκύψει κατά την *αποτίμηση* (evaluation) του όρου M. Με αυτή την έννοια, αποτίμηση ενός όρου ονομάζεται η διαδοχική εφαρμογή κανόνων μετατροπής σε αυτόν.

Όταν σε έναν όρο M δεν μπορούμε να εφαρμόσουμε κανένα αμιγώς υπολογιστικό βήμα, δηλαδή καμία β- ή η-μετατροπή, τότε μπορεί να θεωρηθεί ότι η αποτίμησή του έχει ολοκληρωθεί και ότι ο όρος αυτός αποτελεί ένα τελικό αποτέλεσμα. Τέτοιοι πλήρως αποτιμημένοι όροι ονομάζονται *κανονικές μορφές*.

Ορισμός 9.15 *Ένας όρος $M \in \Lambda$* είναι σε κανονική μορφή *(normal form) αν δεν περιέχει κανένα β-redex ή η-redex.*

[4] Για το λόγο αυτό, στη βιβλιογραφία συχνά η α-μετατροπή παραλείπεται εντελώς και δυο όροι $M, N \in \Lambda$ για τους οποίους ισχύει $M =_\alpha N$ θεωρούνται ταυτόσημοι.

[5] Στη βιβλιογραφία συχνά η η-μετατροπή αντιμετωπίζεται ως δευτερεύουσας σημασίας και πολλές φορές αγνοείται εντελώς.

Η κανονική μορφή του ορισμού 9.15 ονομάζεται συχνά στη βιβλιογραφία και ως *βη-κανονική μορφή* ($\beta\eta$-normal form). Περιορίζοντας περισσότερο το είδος της μετατροπής, ένας όρος M είναι σε *β-κανονική μορφή* (β-normal form) αν δεν περιέχει κανένα β-redex και ομοίως σε *η-κανονική μορφή* (η-normal form) αν δεν περιέχει κανένα η-redex.

Παράδειγμα 9.8 Οι όροι $\lambda x.\, x$ και $\lambda f.\, f\,(\lambda x.\, x\, f)$ είναι σε κανονική μορφή. Αντίθετα, ο όρος $\lambda z.\,(\lambda f.\,\lambda x.\, f\, z\, x)\,(\lambda y.\, y)$ δεν είναι σε κανονική μορφή γιατί περιέχει το β-redex:

$$(\lambda f.\,\lambda x.\, f\, z\, x)\,(\lambda y.\, y) \;\rightarrow_\beta\; \lambda x.\,(\lambda y.\, y)\, z\, x$$

□

Στον ορισμό 9.15 παρατηρεί κανείς ότι στις κανονικές μορφές δε λαμβάνεται υπόψη η α-μετατροπή. Αυτό είναι σύμφωνο με τη θεώρηση του λ-λογισμού ως υπολογιστικού μοντέλου και, αν δε συνέβαινε, απλοί όροι όπως ο $\lambda x.\, x$ δε θα ήταν σε κανονική μορφή λόγω της δυνατής μετονομασίας κάποιας δεσμεύουσας μεταβλητής. Κατά συνέπεια η σχέση συμφωνίας $=_\alpha$, βάσει της οποίας όλοι οι όροι που προκύπτουν με α-μετατροπές είναι ισοδύναμοι, αποτελεί θεμελιώδη σχέση ισοδυναμίας μεταξύ κανονικών μορφών. Το γεγονός αυτό αποδίδεται και από την επόμενη πρόταση.

Πρόταση 9.1 *Αν ο όρος $M \in \Lambda$ είναι σε κανονική μορφή και ισχύει $M \twoheadrightarrow N$ για κάποιον όρο $N \in \Lambda$, τότε $M =_\alpha N$.*

Υπάρχουν λ-όροι η αποτίμηση των οποίων οδηγεί, μετά από διαδοχικές μετατροπές, σε κάποια κανονική μορφή. Υπάρχουν όμως και όροι για τους οποίους αυτό δεν ισχύει, δηλαδή η αποτίμησή τους μπορεί να συνεχίζεται επ' άπειρον χωρίς να καταλήγει σε κανονική μορφή.

Ορισμός 9.16 *Ένας όρος $M \in \Lambda$* έχει κανονική μορφή *αν για κάποιο όρο $N \in \Lambda$ ισχύει $M \twoheadrightarrow N$ και ο N είναι σε κανονική μορφή. Στην περίπτωση αυτή, ο όρος M ονομάζεται* κανονικοποιήσιμος *(normalizing).*

Παράδειγμα 9.9 Ο όρος $\lambda z.\,(\lambda f.\,\lambda x.\, f\, z\, x)\,(\lambda y.\, y)$ στο παράδειγμα 9.8 είδαμε ότι δεν είναι σε κανονική μορφή. Όμως, εφαρμόζοντας διαδοχικά βήματα μετατροπής προκύπτει:

$$\begin{aligned}\lambda z.\,(\lambda f.\,\lambda x.\, f\, z\, x)\,(\lambda y.\, y) &\rightarrow_\beta \lambda z.\,\lambda x.\,(\lambda y.\, y)\, z\, x \\ &\rightarrow_\beta \lambda z.\,\lambda x.\, z\, x \\ &\rightarrow_\eta \lambda z.\, z\end{aligned}$$

και άρα $\lambda z.\,(\lambda f.\,\lambda x.\, f\, z\, x)\,(\lambda y.\, y) \twoheadrightarrow \lambda z.\, z$. Καθώς ο όρος $\lambda z.\, z$ είναι σε κανονική μορφή, ο αρχικός όρος έχει αυτή την κανονική μορφή. □

Παράδειγμα 9.10 Έστω ο όρος $\Omega \equiv (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$, που δεν είναι σε κανονική μορφή. Ο όρος αυτός περιέχει μόνο ένα β-redex και κατά συνέπεια η μόνη μετατροπή που μπορεί να εφαρμοσθεί είναι η:

$$\Omega \;\equiv\; (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \;\to_\beta\; (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \;\equiv\; \Omega$$

Σε ένα βήμα μετατροπής, ο όρος Ω μετατρέπεται πάλι στον εαυτό του. Επομένως η διαδικασία αποτίμησης δεν μπορεί να τερματιστεί και ο όρος Ω δεν έχει κανονική μορφή. □

Στην περίπτωση του Ω, η διαδικασία της αποτίμησης δεν παρουσιάζει πρόοδο. Υπάρχουν όμως όροι των οποίων η αποτίμηση όχι μόνο δεν παρουσιάζει πρόοδο, αλλά οδηγεί στην ανεξέλεγκτη αύξησή τους.

Παράδειγμα 9.11 Έστω ο όρος $M \;\equiv\; (\lambda x.\, x\, x\, y)\, (\lambda x.\, x\, x\, y)$, που περιέχει μόνο ένα β-redex. Η μόνη ακολουθία μετατροπών που μπορεί να εφαρμοσθεί είναι η ακόλουθη:

$$\begin{array}{lll} M & \equiv & (\lambda x.\, x\, x\, y)\, (\lambda x.\, x\, x\, y) \\ & \to_\beta & (\lambda x.\, x\, x\, y)\, (\lambda x.\, x\, x\, y)\, y \;\;\equiv\;\; M\, y \\ & \to_\beta & (\lambda x.\, x\, x\, y)\, (\lambda x.\, x\, x\, y)\, y\, y \;\;\equiv\;\; M\, y\, y \\ & \to_\beta & (\lambda x.\, x\, x\, y)\, (\lambda x.\, x\, x\, y)\, y\, y\, y \;\;\equiv\;\; M\, y\, y\, y \\ & \to_\beta & \ldots \end{array}$$

Κάθε αποτέλεσμα σε αυτή την ακολουθία μετατροπών δεν είναι σε κανονική μορφή και έχει ένα μόνο β-redex. □

Αν ένας όρος M περιέχει περισσότερα του ενός redex, η αποτίμησή του μπορεί να ακολουθήσει περισσότερους διαφορετικούς δρόμους, όπως φαίνεται στο ακόλουθο παράδειγμα.

Παράδειγμα 9.12 Έστω ο όρος $M = (\lambda x.\, (\lambda y.\, x\, y)\, z)\, w$ που περιέχει δύο β-redex για τις δύο αφαιρέσεις με δεσμεύουσες μεταβλητές τις x και y. Μετατρέποντας πρώτα το πρώτο από αυτά προκύπτει η εξής αποτίμηση:

$$M \;\to_\beta\; (\lambda y.\, w\, y)\, z \;\to_\beta\; w\, z$$

ενώ μετατρέποντάς τα με την αντίστροφη σειρά προκύπτει η εξής αποτίμηση:

$$M \;\to_\beta\; (\lambda x.\, x\, z)\, w \;\to_\beta\; w\, z$$

□

Το γεγονός ότι οι δύο δρόμοι αποτίμησης στο προηγούμενο παράδειγμα καταλήγουν στην ίδια κανονική μορφή δεν είναι τυχαίο, όπως θα δούμε στην επόμενη ενότητα. Υπάρχουν όμως όροι για τους οποίους η εύρεση ή όχι κανονικής μορφής εξαρτάται από τη σειρά με την οποία θα γίνουν οι μετατροπές.

Παράδειγμα 9.13 Έστω ο όρος $M = (\lambda z.\, y)\, ((\lambda x.\, x\, x)\, (\lambda x.\, x\, x))$, που περιέχει τον όρο Ω του παραδείγματος 9.10. Ο όρος M περιέχει δύο β-redex για τις δύο αφαιρέσεις με δεσμεύουσες μεταβλητές την z και την πρώτη από τις δύο x. Μετατρέποντας το πρώτο, οδηγούμαστε αμέσως σε κανονική μορφή:

$$M \;\to_\beta\; y$$

Αντίθετα, μετατρέποντας το δεύτερο οδηγούμαστε και πάλι στον ίδιο όρο, ακριβώς όπως συμβαίνει με τη μετατροπή του όρου Ω:

$$M \;\to_\beta\; (\lambda z.\, y)\,((\lambda x.\, x\, x)\,(\lambda x.\, x\, x)) \;\equiv\; M$$

Διαπιστώνει λοιπόν κανείς ότι, αν και ο όρος M διαθέτει κανονική μορφή, αν επιλέγουμε κάθε φορά να μετατρέπουμε το δεύτερο β-redex τότε η διαδικασία της αποτίμησης δε θα τερματιστεί ποτέ και άρα η κανονική μορφή δε θα βρεθεί. □

Ορισμός 9.17 *Ένας όρος M ονομάζεται* ισχυρά κανονικοποιήσιμος *(strongly normalizing) αν όλες οι ακολουθίες μετατροπής που ξεκινούν με τον M καταλήγουν σε κανονική μορφή.*

Όπως είδαμε στο παράδειγμα 9.13, υπάρχουν κανονικοποιήσιμοι όροι που δεν είναι ισχυρά κανονικοποιήσιμοι.

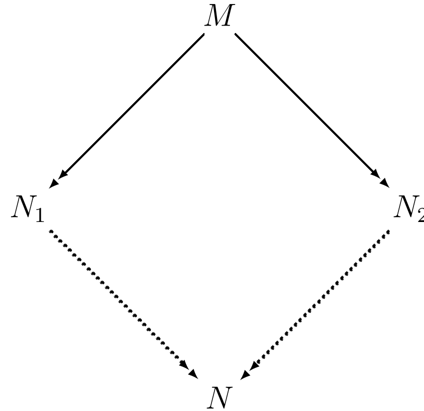
Η ύπαρξη περισσότερων της μιας δυνατών ακολουθιών μετατροπής που να ξεκινούν από ένα δεδομένο λ-όρο καθιστά τη διαδικασία της αποτίμησης στη γενική περίπτωση μη ντετερμινιστική. Μια ντετερμινιστική μέθοδος η οποία αποφασίζει σε κάθε βήμα της αποτίμησης ποιο redex θα μετατραπεί ονομάζεται *στρατηγική αναγωγής* (reduction strategy). Μια από τις απλούστερες και σημαντικότερες, όπως θα δούμε στη συνέχεια, στρατηγικές αναγωγής είναι η ακόλουθη.

Ορισμός 9.18 *Η στρατηγική κατά την οποία επιλέγεται για μετατροπή κάθε φορά το αριστερότερο redex ενός όρου, δηλαδή αυτό του οποίου το σύμβολο λ βρίσκεται όσο το δυνατόν πιο αριστερά, ονομάζεται* στρατηγική της αναγωγής κανονικής σειράς *(normal order reduction strategy).*

9.6 Ιδιότητες του λογισμού

Όπως αναφέραμε ήδη πολλές φορές, ο λ-λογισμός σχεδιάστηκε με σκοπό να αποτελέσει μια γενική θεωρία συναρτήσεων. Σε μια τέτοια θεωρία, θα περίμενε κανείς ότι η αποτίμηση ενός όρου M μπορεί να δώσει το πολύ ένα τελικό αποτέλεσμα, δηλαδή το πολύ μια κανονική μορφή. Επομένως, λαμβάνοντας υπόψη τη σχέση ισοδυναμίας μεταξύ κανονικών μορφών, θα περίμενε κανείς ότι αν η αποτίμηση του όρου M μπορεί να οδηγήσει σε δύο κανονικές μορφές N_1 και N_2, τότε αυτές θα πρέπει να είναι ισοδύναμες, δηλαδή να ισχύει $N_1 =_\alpha N_2$. Αυτή η ιδέα της μοναδικότητας των κανονικών μορφών εκφράζεται ακριβώς από την πρόταση 9.3. Η απόδειξή της στηρίζεται στο ακόλουθο γενικότερο θεώρημα, το οποίο βεβαιώνει ότι αν η διαδικασία της αποτίμησης ενός όρου μπορεί να ακολουθήσει δυο διαφορετικούς δρόμους, τότε αυτοί υποχρεωτικά κάπου συγκλίνουν.

Θεώρημα 9.1 (Church-Rosser) *Έστω όροι* $M, N_1, N_2 \in \Lambda$ *τέτοιοι ώστε* $M \twoheadrightarrow N_1$ *και* $M \twoheadrightarrow N_2$. *Τότε υπάρχει όρος* $N \in \Lambda$ *τέτοιος ώστε* $N_1 \twoheadrightarrow N$ *και* $N_2 \twoheadrightarrow N$. *Σε μορφή διαγράμματος:*



Η ιδιότητα που περιγράφεται στο θεώρημα 9.1 ονομάζεται *ιδιότητα Church-Rosser* (Church-Rosser property), από τα ονόματα των ερευνητών που την απέδειξαν, ή *ιδιότητα της συμβολής* (confluence property).[6] Η απόδειξη του θεωρήματος είναι αρκετά δύσκολη και ξεφεύγει από το σκοπό αυτού του βιβλίου.

Από το θεώρημα 9.1 προκύπτουν εύκολα οι ακόλουθες προτάσεις ως συμπεράσματα. Σύμφωνα με την πρώτη, η απόδειξη της οποίας παραλείπεται, δύο ίσοι όροι μπορούν να αναχθούν στον ίδιο όρο, ο οποίος όμως δεν είναι απαραίτητα κανονική μορφή. Σύμφωνα με τη δεύτερη, η κανονική μορφή ενός όρου (αν υπάρχει) είναι μοναδική.

Πρόταση 9.2 *Αν* $M_1 = M_2$, *τότε υπάρχει ένας όρος* N *τέτοιος ώστε* $M_1 \twoheadrightarrow N$ *και* $M_2 \twoheadrightarrow N$.

Πρόταση 9.3 *Κάθε όρος* M *έχει το πολύ μία κανονική μορφή, δεδομένης της ισοδυναμίας που ορίζεται από τη σχέση συμφωνίας* $=_\alpha$.

Απόδειξη Έστω ότι ο όρος M έχει δύο κανονικές μορφές N_1 και N_2. Τότε ισχύει $M \twoheadrightarrow N_1$ και $M \twoheadrightarrow N_2$ και άρα $N_1 = N_2$. Από την πρόταση 9.2 προκύπτει ότι υπάρχει όρος P τέτοιος ώστε $N_1 \twoheadrightarrow P$ και $N_2 \twoheadrightarrow P$. Όμως, από την πρόταση 9.1 συνεπάγεται ότι $N_1 =_\alpha P$ και $N_2 =_\alpha P$ και άρα $N_1 =_\alpha N_2$. □

Όπως είδαμε στο παράδειγμα 9.13 της ενότητας 9.5, η αποτίμηση ενός κανονικοποιήσιμου λ-όρου δεν οδηγεί απαραίτητα στην κανονική του μορφή. Από την πρόταση 9.3 εξασφαλίζεται ότι αν για έναν κανονικοποιήσιμο όρο δύο στρατηγικές αποτίμησης οδηγούν σε δύο κανονικές μορφές, τότε οι τελευταίες θα είναι ισοδύναμες. Δεν εξασφαλίζεται όμως

[6] Τα δύο αυτά ονόματα μπορούν να χρησιμοποιηθούν για οποιαδήποτε σχέση μεταξύ λ-όρων η οποία αποδίδεται από ένα παρόμοιο διάγραμμα σε σχήμα ρόμβου. Το θεώρημα 9.1 αναφέρεται στη σχέση $\twoheadrightarrow$, μπορεί όμως να αποδειχθεί ότι την ιδιότητα της συμβολής έχει επίσης η σχέση $=$.

ότι κάθε στρατηγική αποτίμησης θα τερματίζει. Το επόμενο θεώρημα αναδεικνύει μια στρατηγική αποτίμησης που εγγυάται την εύρεση της κανονικής μορφής κάθε κανονικοποιήσιμου όρου.

Θεώρημα 9.2 (Κανονικοποίηση – Normalization) *Αν ο όρος M έχει κανονική μορφή, τότε η στρατηγική της αναγωγής κανονικής σειράς οδηγεί σε αυτήν.*

Το παρακάτω θεώρημα καθιστά δυνατή την αναπαράσταση στο λ-λογισμό αναδρομικών συναρτήσεων. Με αυτό το θέμα δε θα ασχοληθούμε εκτενέστερα στις σημειώσεις αυτές.

Θεώρημα 9.3 (Σταθερό σημείο – Fixed point)

(i) *Για κάθε όρο $F \in \Lambda$ υπάρχει όρος $X \in \Lambda$ τέτοιος ώστε να ισχύει $F\,X = X$.*[7]

(ii) *Υπάρχει ένας* τελεστής σταθερού σημείου, *δηλαδή ένας όρος $\mathsf{Y} \in \Lambda$ τέτοιος ώστε για κάθε $F \in \Lambda$ να ισχύει $F\,(\mathsf{Y}\,F) = \mathsf{Y}\,F$.*

Απόδειξη

(i) Έστω $W \;\equiv\; \lambda x.\,F\,(x\,x)$ και $X \;\equiv\; W\,W$. Τότε ισχύει:

$$X \;\equiv\; W\,W \;\equiv\; (\lambda x.\,F\,(x\,x))\,W \;\to_\beta\; F\,(W\,W) \;\equiv\; F\,X$$

(ii) Έστω ο κλειστός όρος $\mathsf{Y} \in \Lambda$ που ορίζεται ως

$$\mathsf{Y} \;\equiv\; \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))$$

και έστω όρος $F \in \Lambda$. Χρησιμοποιώντας την απόδειξη του (i) έχουμε

$$\mathsf{Y}\,F \;\equiv\; (\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,F \;\to_\beta\; (\lambda x.\,F\,(x\,x))\,(\lambda x.\,F\,(x\,x)) \;\equiv\; X$$

Καθώς $X = F\,X$, από τη συμβατότητα της σχέσης $=$ εύκολα προκύπτει $\mathsf{Y}\,F = F\,(\mathsf{Y}\,F)$.

□

[7] Στη μαθηματική ορολογία, ο όρος X ονομάζεται *σταθερό σημείο* της συνάρτησης F.

9.7 Η εκφραστική δύναμη του λάμβδα λογισμού

Ο λ-λογισμός εκ πρώτης όψεως μοιάζει με μια πολύ πρωτόγονη γλώσσα, η οποία δεν μπορεί να εκφράσει πολλές έννοιες. Η πραγματικότητα όμως είναι εντελώς διαφορετική. Ο λ-λογισμός μπορεί να περιγράψει τις περισσότερες έννοιες που χρησιμοποιούνται στις σύγχρονες γλώσσες προγραμματισμού. Στη συνέχεια θα εξετάσουμε πώς γνωστές έννοιες από τα μαθηματικά, τη λογική και τον προγραμματισμό μπορούν να αναπαρασταθούν με τη μορφή λ-όρων.

9.7.1 Λογικές τιμές

Μπορεί κανείς σχετικά εύκολα και με πολλούς διαφορετικούς τρόπους να κωδικοποιήσει στο λ-λογισμό τις λογικές τιμές ***true*** και ***false***. Η συνηθέστερη κωδικοποίηση είναι η ακόλουθη:

Ορισμός 9.19

$$\begin{aligned} \textbf{\textit{true}} &\equiv \lambda x.\, \lambda y.\, x \\ \textbf{\textit{false}} &\equiv \lambda x.\, \lambda y.\, y \end{aligned}$$

Εύκολα μπορεί κανείς να κωδικοποιήσει τις λογικές πράξεις πάνω σε λ-όρους. Για παράδειγμα, η λογική άρνηση ***not*** μπορεί να κωδικοποιηθεί με τον ακόλουθο τρόπο.

Ορισμός 9.20 $\textbf{\textit{not}} \equiv \lambda z.\, z\, \textbf{\textit{false}}\, \textbf{\textit{true}}$

(όπου τα ***true*** και ***false*** είναι οι όροι που ορίστηκαν παραπάνω). Εύκολα διαπιστώνει κανείς ότι το ***not*** διαθέτει τις απαιτούμενες ιδιότητες.

Θεώρημα 9.4

(i) $\textbf{\textit{not}}\ \textbf{\textit{true}} = \textbf{\textit{false}}$

(ii) $\textbf{\textit{not}}\ \textbf{\textit{false}} = \textbf{\textit{true}}$

Απόδειξη Θα δείξουμε μόνο το πρώτο (η απόδειξη του δεύτερου γίνεται παρόμοια).

$$\begin{aligned} \textbf{\textit{not}}\ \textbf{\textit{true}} &\equiv (\lambda z.\, z\, \textbf{\textit{false}}\, \textbf{\textit{true}})\, \textbf{\textit{true}} \\ &\rightarrow_\beta \textbf{\textit{true}}\, \textbf{\textit{false}}\, \textbf{\textit{true}} \\ &\equiv (\lambda x.\, \lambda y.\, x)\, \textbf{\textit{false}}\, \textbf{\textit{true}} \\ &\twoheadrightarrow_\beta \textbf{\textit{false}} \end{aligned}$$

□

Μπορούμε επίσης να ορίσουμε το λ-όρο ***cond*** που κωδικοποιεί τη δομή ***if-then-else***. Θα χρησιμοποιούμε κατά παράβαση αυτή τη δομή για να διευκολύνουμε τη γραφή των όρων που κωδικοποιούν συνθήκες.

Ορισμός 9.21

$$\begin{array}{lcl} \textit{\textbf{cond}} & \equiv & \lambda z.\, \lambda x.\, \lambda y.\, z\, x\, y \\ \textit{\textbf{if}}\ B\ \textit{\textbf{then}}\ N\ \textit{\textbf{else}}\ M & \equiv & \textit{\textbf{cond}}\, B\, N\, M \end{array}$$

Αποδεικνύεται εύκολα ότι η παραπάνω δομή πληροί τις απαιτούμενες ιδιότητες.

Θεώρημα 9.5 *Για κάθε $N, M \in \Lambda$ ισχύουν τα παρακάτω:*

(i) ***if true then*** N ***else*** $M = N$

(ii) ***if false then*** N ***else*** $M = M$

Απόδειξη Θα δείξουμε και πάλι μόνο το πρώτο (η απόδειξη του δεύτερου γίνεται παρόμοια).

$$\begin{array}{lcl} \textit{\textbf{if true then}}\ N\ \textit{\textbf{else}}\ M & \equiv & \textit{\textbf{cond true}}\, N\, M \\ & \equiv & (\lambda z.\, \lambda x.\, \lambda y.\, z\, x\, y)\, \textit{\textbf{true}}\, N\, M \\ & \twoheadrightarrow_\beta & \textit{\textbf{true}}\, N\, M \\ & \equiv & (\lambda x.\, \lambda y.\, x)\, N\, M \\ & \twoheadrightarrow_\beta & N \end{array}$$

□

9.7.2 Διατεταγμένα ζεύγη

Στο λ-λογισμό μπορεί επίσης κανείς να κωδικοποιήσει διατεταγμένα ζεύγη όρων (που με τη σειρά τους μπορούν να κωδικοποιούν άλλα μαθηματικά αντικείμενα). Μία τέτοια δυνατή κωδικοποίηση είναι μέσω του όρου ***pair*** που ορίζεται παρακάτω. Κατά παράβαση, θα χρησιμοποιούμε το συμβολισμό $\langle N, M\rangle$ για να διευκολύνουμε τη γραφή όρων που κωδικοποιούν διατεταγμένα ζεύγη.

Ορισμός 9.22

$$\begin{array}{lcl} \textit{\textbf{pair}} & \equiv & \lambda x.\, \lambda y.\, \lambda z.\, z\, x\, y \\ \langle N, M\rangle & \equiv & \textit{\textbf{pair}}\, N\, M \end{array}$$

Οι πράξεις ***fst*** και ***snd*** που επιστρέφουν το πρώτο και το δεύτερο στοιχείο ενός διατεταγμένου ζεύγους μπορούν να κωδικοποιηθούν ως εξής:

Ορισμός 9.23

$$\begin{array}{lcl} \textit{\textbf{fst}} & \equiv & \lambda z.\, z\, \textit{\textbf{true}} \\ \textit{\textbf{snd}} & \equiv & \lambda z.\, z\, \textit{\textbf{false}} \end{array}$$

Εύκολα μπορεί κανείς να δείξει ότι οι πράξεις αυτές πληρούν τις απαιτούμενες ιδιότητες και άρα η δομή $\langle N, M \rangle$ μπορεί να χρησιμεύσει ως διατεταγμένο ζεύγος.

Θεώρημα 9.6 *Για κάθε $N, M \in \Lambda$ ισχύουν τα εξής:*

(i) $\boldsymbol{fst}\, \langle N, M \rangle = N$

(ii) $\boldsymbol{snd}\, \langle N, M \rangle = M$

Απόδειξη Θα δείξουμε μόνο το πρώτο (η απόδειξη του δεύτερου γίνεται παρόμοια).

$$
\begin{aligned}
\boldsymbol{fst}\, \langle N, M \rangle &\equiv (\lambda z.\, z\, \boldsymbol{true})\, \langle N, M \rangle \\
&\rightarrow_\beta \langle N, M \rangle\, \boldsymbol{true} \\
&\equiv \boldsymbol{pair}\, N\, M\, \boldsymbol{true} \\
&\equiv (\lambda x.\, \lambda y.\, \lambda z.\, z\, x\, y)\, N\, M\, \boldsymbol{true} \\
&\twoheadrightarrow_\beta \boldsymbol{true}\, N\, M \\
&\equiv (\lambda x.\, \lambda y.\, x)\, N\, M \\
&\twoheadrightarrow_\beta N
\end{aligned}
$$

□

9.7.3 Φυσικοί αριθμοί

Έχουν προταθεί πολλές διαφορετικές κωδικοποιήσεις των φυσικών αριθμών στο λ-λογισμό. Η πρώτη και πιο γνωστή από αυτές είναι τα αριθμοειδή του Church, που περιγράφονται παρακάτω.

Ορισμός 9.24 *Έστω φυσικός αριθμός $n \in \mathbb{N}$ και όροι $F, A \in \Lambda$. Ο όρος $F^n(A) \in \Lambda$ ορίζεται επαγωγικά ως*

$$
\begin{aligned}
F^0(A) &\equiv A \\
F^{n+1}(A) &\equiv F\, (F^n(A))
\end{aligned}
$$

Ο ορισμός του $F^n(A)$ θα μπορούσε ισοδύναμα να δοθεί με τη μορφή $F^{n+1}(A) \equiv F^n(F\, A)$. Με επαγωγή μπορεί κανείς να αποδείξει εύκολα ότι $F^n(F\, A) \equiv F\, (F^n(A))$ και ότι $F^n(F^m(A)) \equiv F^{n+m}(A)$.

Ορισμός 9.25 (Αριθμοειδή του Church – Church numerals) *Για κάθε φυσικό αριθμό $n \in \mathbb{N}$ ορίζεται ένας όρος $\boldsymbol{c}_n \in \Lambda$ ως*

$$
\boldsymbol{c}_n \equiv \lambda f.\, \lambda x.\, f^n(x)
$$

Το αριθμοειδές που αντιστοιχεί στον αριθμό 0 είναι το $\boldsymbol{c}_0 \equiv \lambda f.\, \lambda x.\, x$, στον αριθμό 1 αντιστοιχεί το $\boldsymbol{c}_1 \equiv \lambda f.\, \lambda x.\, f\, x$, στον αριθμό 2 το $\boldsymbol{c}_2 \equiv \lambda f.\, \lambda x.\, f\,(f\,x)$, κ.ο.κ. Όπως εύκολα διαπιστώνεται από τον ορισμό τους, όλα τα αριθμοειδή είναι σε β-κανονική μορφή (μάλιστα όλα εκτός του $\boldsymbol{c}_1$ είναι και σε η-κανονική μορφή). Προφανώς, ούτε όλοι οι λ-όροι είναι αριθμοειδή, ούτε και μπορούν όλοι να αναχθούν σε αριθμοειδή.

Μπορεί κανείς τώρα να ορίσει την κωδικοποίηση πάνω στα αριθμοειδή μερικών βασικών πράξεων των φυσικών αριθμών: της συνάρτησης που επιστρέφει τον επόμενο φυσικό αριθμό και των πράξεων της πρόσθεσης, του πολλαπλασιασμού και της ύψωσης σε δύναμη.

Ορισμός 9.26

$$\begin{array}{lcl} \boldsymbol{succ} & \equiv & \lambda n.\, \lambda f.\, \lambda x.\, n\, f\,(f\,x) \\ \boldsymbol{A}_{+} & \equiv & \lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, n\, f\,(m\, f\, x) \\ \boldsymbol{A}_{*} & \equiv & \lambda n.\, \lambda m.\, \lambda f.\, n\,(m\, f) \\ \boldsymbol{A}_{\exp} & \equiv & \lambda n.\, \lambda m.\, m\, n \end{array}$$

Το θεώρημα 9.7 αποδεικνύει την ορθότητα της κωδικοποίησης των πράξεων του παραπάνω ορισμού. Για την απόδειξή του είναι χρήσιμο το παρακάτω λήμμα.

Λήμμα 9.1 *Για κάθε $n, m \in \mathbb{N}$ και για κάθε $x, y \in \Lambda$ ισχύουν τα εξής:*

(i) $\boldsymbol{c}_n\, f\,(\boldsymbol{c}_m\, f\, x) = \boldsymbol{c}_{n+m}\, f\, x$

(ii) $(\boldsymbol{c}_n\, x)^m(y) = x^{nm}(y)$

(iii) *Αν* $m > 0$, *τότε* $(\boldsymbol{c}_n)^m(x) = \boldsymbol{c}_{n^m}\, x$

Απόδειξη

(i)

$$\begin{array}{rcl} \boldsymbol{c}_n\, f\,(\boldsymbol{c}_m\, f\, x) & \equiv & (\lambda f.\, \lambda x.\, f^n(x))\, f\,((\lambda f.\, \lambda x.\, f^m(x)))\, f\, x) \\ & \twoheadrightarrow_\beta & (\lambda f.\, \lambda x.\, f^n(x))\, f\,(f^m(x)) \\ & \twoheadrightarrow_\beta & f^n(f^m(x)) \\ & \equiv & f^{n+m}(x) \\ & \twoheadleftarrow_\beta & (\lambda f.\, \lambda x.\, f^{n+m}(x))\, f\, x \\ & \equiv & \boldsymbol{c}_{n+m}\, f\, x \end{array}$$

(ii) Με επαγωγή στο m. Για $m = 0$ προφανώς ισχύει $y = y$. Έστω ότι ισχύει για τυχαίο m. Τότε:

$$\begin{array}{rcll} (\boldsymbol{c}_n\, x)^{m+1}(y) & \equiv & \boldsymbol{c}_n\, x\,((\boldsymbol{c}_n\, x)^m(y)) & \\ & = & \boldsymbol{c}_n\, x\,(x^{nm}(y)) & \langle\ \textit{επαγωγική υπόθεση}\ \rangle \\ & \equiv & (\lambda f.\, \lambda x.\, f^n(x))\, x\,(x^{nm}(y)) & \\ & \twoheadrightarrow_\beta & x^n(x^{nm}(y)) & \\ & \equiv & x^{n+nm}(y) & \\ & \equiv & x^{n(m+1)}(y) & \end{array}$$

(iii) Με επαγωγή στο m. Για $m = 1$ προφανώς ισχύει $\boldsymbol{c}_n\, x = \boldsymbol{c}_n\, x$. Έστω ότι ισχύει για τυχαίο $m > 0$. Τότε:

$$
\begin{array}{rcll}
(\boldsymbol{c}_n)^{m+1}(x) & \equiv & \boldsymbol{c}_n\,((\boldsymbol{c}_n)^m(x)) & \\
& = & \boldsymbol{c}_n\,(\boldsymbol{c}_{n^m}\, x) & \langle \text{ επαγωγική υπόθεση } \rangle \\
& \equiv & (\lambda f.\, \lambda x.\, f^n(x))\,(\boldsymbol{c}_{n^m}\, x) & \\
& \to_\beta & \lambda y.\, (\boldsymbol{c}_{n^m}\, x)^n(y) & \\
& = & \lambda y.\, x^{n^m n}(y) & \langle \text{ λήμμα 9.1 (ii) } \rangle \\
& \equiv & \lambda y.\, x^{n^{m+1}}(y) & \\
& \leftarrow_\beta & (\lambda f.\, \lambda x.\, f^{n^{m+1}}(x))\, x & \\
& \equiv & \boldsymbol{c}_{n^{m+1}}\, x &
\end{array}
$$

□

Θεώρημα 9.7 *Για κάθε $n, m \in \mathbb{N}$ ισχύουν τα εξής:*

(i) $\boldsymbol{succ}\, \boldsymbol{c}_n = \boldsymbol{c}_{n+1}$

(ii) $\boldsymbol{A}_+\, \boldsymbol{c}_n\, \boldsymbol{c}_m = \boldsymbol{c}_{n+m}$

(iii) $\boldsymbol{A}_*\, \boldsymbol{c}_n\, \boldsymbol{c}_m = \boldsymbol{c}_{nm}$

(iv) *Αν $m > 0$ τότε $\boldsymbol{A}_{\exp}\, \boldsymbol{c}_n\, \boldsymbol{c}_m = \boldsymbol{c}_{n^m}$*

Απόδειξη

(i)

$$
\begin{array}{rcl}
\boldsymbol{succ}\, \boldsymbol{c}_n & \equiv & (\lambda n.\, \lambda f.\, \lambda x.\, n\, f\, (f\, x))\, (\lambda f.\, \lambda x.\, f^n(x)) \\
& \to_\beta & \lambda f.\, \lambda x.\, (\lambda f.\, \lambda x.\, f^n(x))\, f\, (f\, x) \\
& \twoheadrightarrow_\beta & \lambda f.\, \lambda x.\, f^n(f\, x) \\
& \equiv & \boldsymbol{c}_{n+1}
\end{array}
$$

(ii)

$$
\begin{array}{rcll}
\boldsymbol{A}_+\, \boldsymbol{c}_n\, \boldsymbol{c}_m & \equiv & (\lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, n\, f\, (m\, f\, x))\, \boldsymbol{c}_n\, \boldsymbol{c}_m & \\
& \twoheadrightarrow_\beta & \lambda f.\, \lambda x.\, \boldsymbol{c}_n\, f\, (\boldsymbol{c}_m\, f\, x) & \\
& = & \lambda f.\, \lambda x.\, \boldsymbol{c}_{n+m}\, f\, x & \langle \text{ λήμμα 9.1 (i) } \rangle \\
& \twoheadrightarrow_\eta & \boldsymbol{c}_{n+m} &
\end{array}
$$

(iii)

$$
\begin{array}{rcll}
\boldsymbol{A}_*\, \boldsymbol{c}_n\, \boldsymbol{c}_m & \equiv & (\lambda n.\, \lambda m.\, \lambda f.\, n\, (m\, f))\, \boldsymbol{c}_n\, \boldsymbol{c}_m & \\
& \twoheadrightarrow_\beta & \lambda f.\, \boldsymbol{c}_n\, (\boldsymbol{c}_m\, f) & \\
& \equiv & \lambda f.\, (\lambda f.\, \lambda x.\, f^n(x))\, (\boldsymbol{c}_m\, f) & \\
& \to_\beta & \lambda f.\, \lambda x.\, (\boldsymbol{c}_m\, f)^n(x) & \\
& = & \lambda f.\, \lambda x.\, f^{mn}(x) & \langle \text{ λήμμα 9.1 (ii) } \rangle \\
& \equiv & \boldsymbol{c}_{nm} &
\end{array}
$$

$$
\begin{aligned}
\text{(iv)} \quad \boldsymbol{A}_{\exp}\, \boldsymbol{c}_n\, \boldsymbol{c}_m &\equiv (\lambda n.\, \lambda m.\, m\, n)\, \boldsymbol{c}_n\, \boldsymbol{c}_m \\
&\twoheadrightarrow_\beta \boldsymbol{c}_m\, \boldsymbol{c}_n \\
&\equiv (\lambda f.\, \lambda x.\, f^m(x))\, \boldsymbol{c}_n \\
&\rightarrow_\beta \lambda x.\, (\boldsymbol{c}_n)^m(x) \\
&= \lambda x.\, \boldsymbol{c}_{n^m}\, x && \langle \text{ λήμμα 9.1 (iii) } \rangle \\
&\rightarrow_\eta \boldsymbol{c}_{n^m}
\end{aligned}
$$

□

9.7.4 Λάμβδα λογισμός και πέρασμα παραμέτρων

Η σχέση ομοιότητας του λ-λογισμού με τις γλώσσες προγραμματισμού δεν περιορίζεται στην εκφραστική του ικανότητα ως προγραμματιστικού μοντέλου. Οι στρατηγικές αναγωγής λ-όρων είναι πολύ στενά συνδεδεμένες με τις μεθόδους περάσματος παραμέτρων των γλωσσών προγραμματισμού.

Στην αναλογία μεταξύ λ-λογισμού και γλωσσών προγραμματισμού:

- οι λ-όροι αντιστοιχούν σε εκφράσεις ή εντολές,
- η αφαίρεση και η εφαρμογή αντιστοιχούν στον ορισμό και την κλήση συναρτήσεων ή διαδικασιών, και
- η διαδικασία της αναγωγής αντιστοιχεί στην αποτίμηση εκφράσεων ή την εκτέλεση εντολών.

Σε ορισμένες γλώσσες προγραμματισμού, η αποτίμηση μιας έκφρασης πολλές φορές σταματά πριν προκύψει ένα πλήρως αποτιμημένο αποτέλεσμα. Το είδος αυτό της αποτίμησης ονομάζεται *οκνηρή αποτίμηση* (lazy evaluation) και υποστηρίζεται κυρίως από γλώσσες συναρτησιακού προγραμματισμού, όπως η Haskell και η Miranda. Η οκνηρή αποτίμηση μιας έκφρασης τερματίζεται όταν το αποτέλεσμα που προκύπτει είναι μια *τιμή* (value).[8] Η ακριβής μορφή των τιμών διαφέρει από γλώσσα σε γλώσσα, γενικά όμως οι αριθμητικές και λογικές σταθερές καθώς και οι συναρτησιακές αφαιρέσεις είναι τιμές. Κατ' αναλογία, στο λ-λογισμό μόνο οι αφαιρέσεις μπορούν να θεωρηθούν τιμές.[9]

Η στρατηγική της αναγωγής κανονικής σειράς σχετίζεται στενά με την οκνηρή αποτίμηση. Η μετατροπή κάθε φορά του αριστερότερου β-redex σημαίνει ότι δεν μπορεί να έχει προηγηθεί μετατροπή μέσα στον όρο, πάνω στον οποίο εφαρμόζεται η αφαίρεση που

[8] Στην αγγλική ορολογία, οι τιμές ονομάζονται επίσης *canonical forms*, όμως ο ελληνικός όρος *κανονική μορφή* έχει ήδη χρησιμοποιηθεί για να αποδώσει τον αγγλικό όρο *normal form*. Για το λόγο αυτό, η χρήση του όρου canonical form αποφεύγεται.

[9] Αν κανείς περιοριστεί στο σύνολο Λ^0 των κλειστών λ-όρων, οι (κλειστές) κανονικές μορφές είναι υποχρεωτικά αφαιρέσεις και άρα κάθε κανονική μορφή είναι τιμή. Το αντίστροφο όμως δεν είναι αληθές.

αντιστοιχεί σε αυτό το β-redex. Κατ' αναλογία, στην οκνηρή αποτίμηση μια συνάρτηση καλείται χωρίς να έχει προηγηθεί η αποτίμηση των παραμέτρων της. Στη γλώσσα προγραμματισμού Algol 60, η συμπεριφορά αυτή επιτυγχάνεται με τη μέθοδο του περάσματος παραμέτρων *κατ' όνομα* (call by name). Η στρατηγική της αναγωγής κανονικής σειράς οδηγεί τελικά την αναγωγή του όρου $(\lambda x.\, \lambda y.\, y)\,\Omega$ στην τιμή $\lambda y.\, y$. Στο ίδιο αποτέλεσμα καταλήγουμε στη γλώσσα Algol 60, αν ορίσουμε τις συναρτήσεις

```
integer procedure f (x);
   integer x;
begin
   f := 42
end;

integer procedure g;
begin
   while true do
      g := 7
end;
```

και στη συνέχεια αποτιμήσουμε την έκφραση `f(g)`. Το αποτέλεσμα είναι η τιμή 42 γιατί κατά την αποτίμηση δε χρειάζεται να γίνει η κλήση στη συνάρτηση `g`, που δεν τερματίζεται.

Αντίθετα, η αποτίμηση της έκφρασης `f(g)` στο παραπάνω παράδειγμα δε θα τερματιζόταν σε μια γλώσσα προγραμματισμού όπως η Algol 68, η Pascal και η C. Στις γλώσσες αυτές το πέρασμα των παραμέτρων γίνεται *κατ' αξία* (by value) και, κατά συνέπεια, η τιμή της παραμέτρου `g` πρέπει να αποτιμηθεί πριν κληθεί η συνάρτηση `f`. Η κλήση της `g` οδηγεί προφανώς σε μη τερματισμό. Αυτού του είδους η αποτίμηση κατά την οποία οι παράμετροι αποτιμώνται πριν γίνει η κλήση στη συνάρτηση ονομάζεται *πρόθυμη αποτίμηση* (eager evaluation).

Στο λ-λογισμό, το ανάλογο της πρόθυμης αποτίμησης και του περάσματος παραμέτρων κατ' αξία είναι μια στρατηγική αποτίμησης που κατά την αποτίμηση του όρου $(\lambda x.\, \lambda y.\, y)\,\Omega$ θα επιχειρούσε πρώτα να αποτιμήσει τον όρο Ω. Τέτοιου είδους στρατηγικές προκύπτουν υποχρεωτικά αν στον κανόνα της β-μετατροπής

$$(\lambda x.\, M)\, N \;\; \rightarrow_\beta \;\; M[x := N]$$

προσθέσουμε τον περιορισμό ότι ο όρος N πρέπει να είναι τιμή.

Ασκήσεις

9.1 Να αποδειχθεί ότι οι σχέσεις $\twoheadrightarrow_\alpha$ και $=_\alpha$ είναι ίσες, δηλαδή για κάθε $M, N \in \Lambda$ ισχύει $M \twoheadrightarrow_\alpha N \;\Leftrightarrow\; M =_\alpha N$.

9.2 Ο κλειστός όρος $\Theta \equiv (\lambda x.\, \lambda y.\, y\,(x\,x\,y))\,(\lambda x.\, \lambda y.\, y\,(x\,x\,y))$ είναι γνωστός ως *τελεστής σταθερού σημείου του Turing*. Να αποδειχθεί ότι ο όρος αυτός διαθέτει την ιδιότητα $\Theta\, F = F\,(\Theta\, F)$ και να σχολιασθεί κατά πόσο αυτός υπερτερεί έναντι του τελεστή Y.

9.3 Να οριστούν οι λογικές πράξεις ***and***, ***or*** και ***xor*** πάνω στην κωδικοποίηση των λογικών τιμών που περιγράφεται στην ενότητα 9.7.1 και να αποδειχθεί ότι πληρούν τις απαιτούμενες ιδιότητες.

9.4 Να αποδειχθεί ότι και ο όρος $\lambda x.\, \lambda y \lambda z y\,(x\,y\,z) \in \Lambda$ κωδικοποιεί τη συνάρτηση που επιστρέφει τον επόμενο φυσικό αριθμό, πάνω στα αριθμοειδή του Church.

9.5 Να αποδειχθεί ότι ο όρος

$$\textbf{\textit{iszero}} \;\equiv\; \lambda x.\, x\,(\lambda x.\, \lambda z.\, z\,(\lambda x.\, \lambda y.\, y)\,x)\,(\lambda x.\, x)\,(\lambda x.\, \lambda y.\, x)$$

κωδικοποιεί τη συνάρτηση που, αν εφαρμοστεί στο αριθμοειδές του Church $\boldsymbol{c}_n$, επιστρέφει ***true*** αν $n = 0$ και ***false*** αν $n > 0$.

9.6 Να αποδειχθεί ότι ο όρος (που οφείλεται στον J. Velmans)

$$\textbf{\textit{pred}} \;\equiv\; \lambda x.\, \lambda y.\, \lambda z.\, x\,(\lambda p.\, \lambda q.\, q\,(p\,y))\,(\lambda y.\, z)\,(\lambda x.\, x)$$

κωδικοποιεί τη συνάρτηση που επιστρέφει τον προηγούμενο φυσικό αριθμό, πάνω στα αριθμοειδή του Church. Να μελετηθεί επίσης τί προκύπτει αν αυτός ο όρος εφαρμοστεί στο $\boldsymbol{c}_0$.

9.7 Να οριστεί ένας τρόπος κωδικοποίησης των διατεταγμένων n-άδων στο λ-λογισμό. Για κάθε $n \in \mathbb{N}$ και για κάθε $M_0, M_1, \ldots, M_{n-1} \in \Lambda$, ο όρος

$$\textbf{\textit{tuple}}\; \boldsymbol{c}_n\, M_0\, M_1\, \ldots\, M_{n-1}$$

πρέπει να αποτελεί κωδικοποίηση της διατεταγμένης n-άδας $\langle M_0, M_1, \ldots, M_{n-1}\rangle$. Επιπλέον, για κάθε $i \in \mathbb{N}$ τέτοιο ώστε $i < n$, ο όρος ***prj*** $\boldsymbol{c}_i$ πρέπει να κωδικοποιεί την πράξη της προβολής του i-οστού στοιχείου της n-άδας. Αφού οριστούν κατάλληλα οι όροι $\textbf{\textit{tuple}}, \textbf{\textit{prj}} \in \Lambda$ να αποδειχθεί ότι, με τους παραπάνω περιορισμούς, ικανοποιείται η σχέση

$$\textbf{\textit{prj}}\; \boldsymbol{c}_i\, \langle M_0,\, M_1\, \ldots\, M_{n-1}\rangle \;=\; M_i$$

9.8 Στη γλώσσα προγραμματισμού της αρεσκείας σας, υλοποιήστε ένα διερμηνέα για το λ-λογισμό.

Βιβλιογραφία

[Bare84] H.P. Barendregt, *The lambda calculus: its syntax and semantics*, Studies in Logic and the Foundations of Mathematics, North Holland, 1984, Revised edition.

[Bare90] H.P. Barendregt, "Functional programming and lambda calculus", in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, pp. 321–364, North Holland, 1990.

[Bare92] H.P. Barendregt, "Lambda calculi with types", in S. Abramsky, D.M. Gabbay and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 2, Oxford Science Publications, 1992.

[Chur32] A. Church, "A set of postulates for the foundations of logic", *Annals of Mathematics*, vol. 33, pp. 346–366, 1932, Part I.

[Chur33] A. Church, "A set of postulates for the foundations of logic", *Annals of Mathematics*, vol. 34, pp. 839–864, 1933, Part II.

[Chur40] A. Church, "A formulation of a simple theory of types", *Journal of Symbolic Logic*, vol. 5, pp. 56–58, 1940.

[Chur41] A. Church, *The calculi of lambda-conversion*, vol. 6 of *Annals of Mathematical Studies*, Princeton University Press, Princeton, NJ, 1941.

[Curr34] H.B. Curry, "Functionality in combinatory logic", in *Proc. Nat. Acad. Science USA*, vol. 20, pp. 584–590, 1934.

[Curr58] H.B. Curry and R. Feys, *Combinatory logic*, vol. I of *Studies in Logic and the Foundations of Mathematics*, North Holland, 1958.

[Curr72] H.B. Curry, J.R. Hindley and J.P. Seldin, *Combinatory logic*, vol. II of *Studies in Logic and the Foundations of Mathematics*, North Holland, 1972.

[Hind86] J.R. Hindley and J.P. Seldin, *Introduction to combinators and the λ-calculus*, vol. 1 of *London Mathematical Society Student Texts*, Cambridge University Press, 1986.

[Klee35] S.C. Kleene and J.B. Rosser, "The inconsistency of certain formal logics", *Annals of Mathematics*, vol. 36, pp. 630–636, 1935.

[Turi37] A. Turing, "Computability and λ-definability", *Journal of Symbolic Logic*, vol. 2, pp. 153–163, 1937.