Ε. ΖΑΧΟΣ

Ν. ΠΑΠΑΣΠΥΡΟΥ

Προγραμματισμός Ηλεκτρονικών Υπολογιστών

Σημειώσεις

ΑΘΗΝΑ 2013

ΠΡΟΛΟΓΟΣ

Στις σημειώσεις αυτές παρουσιάζεται περιληπτικά η ύλη που καλύπτεται στο μάθημα "Προγραμματισμός Ηλεκτρονικών Υπολογιστών", που διδάσκεται στους σπουδαστές του 1ου εξαμήνου της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Ε.Μ.Π.

Προσοχή: οι σημειώσεις δεν υποκαθιστούν την παρακολούθηση του μαθήματος.

Σκοπός του μαθήματος αυτού είναι η εισαγωγή στην επιστήμη των υπολογιστών, την πληροφορική. Περιλαμβάνει συζήτηση των εννοιών: αλγόριθμος, δομή δεδομένων, πρόγραμμα, γλώσσα προγραμματισμού. Η εισαγωγή γίνεται με βάση τη γλώσσα προγραμματισμού Pazcal: ένα εκπαιδευτικό υποσύνολο της C, σχεδιασμένο ειδικά για τις ανάγκες του μαθήματος.

Υπάρχουν διάφορα στάδια απασχόλησης του μηχανικού πληροφορικής: προδιαγραφές, σχεδίαση αλγορίθμων, κωδικοποίηση διαδικασιών και προγράμματος, επαλήθευση, τεκμηρίωση και συντήρηση.

Για εμπέδωση των θεωρητικών γνώσεων, οι σπουδαστές υλοποιούν διάφορες μικρές εφαρμογές στο εργαστήριο, όπου χρησιμοποιούν στην αρχή του εξαμήνου τη γλώσσα Pazcal και προς το τέλος τη C. Εβδομαδιαίως γίνονται 3 ώρες παράδοσης και τουλάχιστον 2 ώρες εργαστηριακές.

Οι σημειώσεις αυτές είναι μετεξέλιξη των σημειώσεων του Στάθη Ζάχου για τον ίδιο σκοπό, με τη γλώσσα Pascal, που χρονολογούνται από τις αρχές της δεκαετίας του 1990. Θέλουμε να ευχαριστήσουμε τους παρακάτω συναδέλφους, μεταδιδακτορικούς, μεταπτυχιακούς και προπτυχιακούς σπουδαστές, που βοήθησαν στην προετοιμασία παλαιότερων εκδόσεων αυτών των σημειώσεων: Άρη Παγουρτζή, Σωτήρη Μπούτση, Κατερίνα Ποτίκα, Παναγιώτη Χείλαρη, Στρατή Ιωαννίδη, Βαγγέλη Τζανή, Γιώργο Ανδρουλιδάκη, Γιώργο Ασημενό και Γιώργο Βακαλόπουλο.

Τέλος, οι σημειώσεις αυτές πρόκειται να επεκταθούν και να βελτιωθούν στο μέλλον. Μπορείτε να ανατρέχετε στην τελευταία τους μορφή που είναι διαθέσιμη από τη σελίδα του μαθήματος: http://courses.softlab.ntua.gr/progintro/.

Στάθης Ζάχος και Νίκος Παπασπύρου, 2013.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός του μαθήματος

1. Εισαγωγή στην Πληροφορική
2. Προγραμματισμός Η/Υ
3. Μεθοδολογία αλγοριθμικής επίλυσης προβλημάτων

Η Πληροφορική, ή επιστήμη και τεχνολογία υπολογιστών όπως επίσης ονομάζεται, είναι η συστηματική μελέτη αλγοριθμικών διαδικασιών που περιγράφουν και επεξεργάζονται πληροφορίες. Πιο συγκεκριμένα, η Πληροφορική περιλαμβάνει: θεωρία, ανάλυση, σχεδίαση, αποδοτικότητα, υλοποίηση και εφαρμογή αλγοριθμικών διαδικασιών. Το βασικό ερώτημα με το οποίο ασχολείται η Πληροφορική είναι: "Τι μπορεί να μηχανοποιηθεί; (και μάλιστα με αποδοτικό τρόπο)".

1.2 Ιστορική αναδρομή

Η Πληροφορική γεννήθηκε πριν περίπου 60 χρόνια από το πάντρεμα της θεωρίας αλγορίθμων, της μαθηματικής λογικής και του σχεδιασμού Η/Υ που έχουν αποθηκευμένα προγράμματα στη μνήμη τους. Σχετικά άτυπα (πρόχειρα) λέμε εδώ ότι **αλγόριθμος** είναι μια ακολουθία ενεργειών που περιγράφει έναν τρόπο επίλυσης ενός προβλήματος. Τα αντικείμενα πάνω στα οποία γίνονται οι ενέργειες του αλγορίθμου είναι κομμάτια πληροφορίας που ονομάζονται **δεδομένα** (data). **Πρόγραμμα** λέγεται μια συντακτικά ακριβής περιγραφή ενός αλγορίθμου, δοσμένη σε μια τυπική γλώσσα, τη γλώσσα προγραμματισμού.

Οι τυπικές γλώσσες (formal languages) έχουν αυστηρό συντακτικό ορισμό σε αντιδιαστολή με τις φυσικές γλώσσες.

Οι φυσικές γλώσσες, όπως π.χ. τα ελληνικά, αγγλικά, κ.λπ., δεν έχουν τόσο αυστηρούς **συντακτικούς** περιορισμούς, έχουν όμως πολύ μεγαλύτερη πυκνότητα και **σημασιολογικές** δυνατότητες (μπορούν π.χ. να εκφράσουν συναισθήματα).

Οι ρίζες της Πληροφορικής βρίσκονται βαθιά στα Μαθηματικά αλλά και στον προβληματισμό των Μηχανικών (Engineering). Από τα Μαθηματικά προέρχεται η αναλυτική μεθοδολογία

και απο τους Μηχανικούς η σχεδίαση (design). Η Πληροφορική άρα περιλαμβάνει θεωρία (μαθηματικό μέρος), πειραματικές μεθόδους και δημιουργία αφηρημένου μοντέλου (επιστημονική μεθοδολογία) και σχεδίαση για κατασκευή (κληρονομιά των Μηχανικών). Σε άλλους κλάδους αυτά είναι πιο ξεκάθαρα διαχωρισμένα, π.χ. παράβαλε με τη διαφορά Χημικών και Χημικών Μηχανικών. Στην Πληροφορική η **επιστημονική μεθοδολογία** και η **κατασκευαστική σχεδίαση** αποτελούν ολοκληρωμένα, αδιαχώριστα κομμάτια ενός και του αυτού κλάδου.

Για χιλιάδες χρόνια, το κύριο αντικείμενο των Μαθηματικών ήταν οι **υπολογισμοί**. Διάφορα μοντέλα φυσικών φαινομένων οδήγησαν σε εξισώσεις, που οι λύσεις τους περιγράφουν αυτά τα φαινόμενα. Συγχρόνως όμως και οι Μηχανικοί χρησιμοποιούσαν υπολογισμούς στη σχεδίαση συστημάτων. Ετσι ένα παράγωγο της αλληλεπίδρασης Μαθηματικών και Τεχνολογίας ήταν και διάφοροι υπολογιστικοί μηχανισμοί.

Τέτοια εργαλεία χρησιμοποιούσαν για την ναυσιπλοϊα ή γεωδεσία για εκατοντάδες χρόνια. Το 17ο αιώνα δύο σπουδαίοι επιστήμονες, ο Pascal και ο Leibniz, κατασκεύασαν υπολογιστικές αριθμομηχανές. Τη δεκαετία 1830-40 ο Babbage σχεδίασε την "αναλυτική μηχανή", που μπορούσε να υπολογίζει λογαρίθμους, τριγωνομετρικές συναρτήσεις και άλλες γενικές αριθμητικές συναρτήσεις. Τη δεκαετία 1920-30 ο Bush κατασκεύασε ηλεκτρονικό αναλογικό υπολογιστή για επίλυση γενικών συστημάτων διαφορικών εξισώσεων. Λίγο αργότερα, το ηλεκτρονικό flip-flop επέτρεψε το πέρασμα σε ψηφιακό υπολογιστή (Zuse).

Από την άλλη μεριά, η Μαθηματική Λογική ενδιαφέρεται για την συλλογιστική εγκυρότητα και για τυπικούς κανόνες. Από την εποχή του Αριστοτέλη και του Ευκλείδη, η Μαθηματική Λογική (αξιώματα, κανόνες, συλλογισμοί, αποδείξεις, θέωρήματα) ήταν αντικείμενο επιστημονικής έρευνας. Το 19ο αιώνα αναζητούσαν ένα καθολικό σύστημα λογικής με τις ιδιότητες της **συνέπειας** και της **πληρότητας**, δηλαδη ένα σύστημα με το οποίο θα μπορούσαμε να αποφασίσουμε μηχανικά εάν μια οποιαδήποτε δεδομένη μαθηματική πρόταση είναι αληθής ή ψευδής (Hilbert). Το 1931 ο Gödel απέδειξε το περίφημο **θεώρημα μη πληρότητας**. Έδειξε δηλαδή ότι δεν είναι δυνατόν να βρεθεί τέτοιο πλήρες και συνεπές σύστημα λογικών κανόνων. Το 1936 ο Turing επεξήγησε την έννοια "μια μαθηματική πρόταση δεν είναι ούτε αληθής ούτε ψευδής" ως εξής: η αληθοτιμή της δεν είναι δυνατόν να υπολογιστεί με μια θεωρητική καθολική υπολογιστική μηχανή (Μηχανή Turing).

Η σπουδαιότητα της εργασίας του Turing (που σήμερα θεωρείται ο πατέρας της Πληροφορικής) έγκειται στα εξής δύο αποτελέσματα:

1. Υπάρχουν καλώς ορισμένα (μαθηματικά) προβλήματα που όμως είναι αδύνατη (όχι απλώς άγνωστη) η λύση τους με οποιοδήποτε υπολογιστικό τρόπο.

2. Χρησιμοποιώντας κωδικοποίηση (με ακολουθίες συμβόλων ή φυσικούς αριθμούς) μπορούμε να παραστήσουμε δεδομένα ή προγράμματα και συνεπώς να τα αποθηκεύσουμε μέσα στη μνήμη μιας υπολογιστικής μηχανής. Αυτή είναι και η βασική διαφορά ενός υπολογιστή (έχει αποθηκευμένο πρόγραμμα) από μια υπολογιστική μηχανή (π.χ. ένα παλιομοδίτικο κομπιουτεράκι).

Σε αυτό, λοιπόν, το υπολογιστικό μοντέλο (που ονομάζεται μοντέλο von Neumann), το κωδικοποιημένο πρόγραμμα είναι αποθηκευμένο στη μνήμη σε μορφή που γίνεται κατανοητή από τη μηχανή (γλώσσα μηχανής). Ο προγραμματιστής όμως μπορεί να χρησιμοποιεί γλώσσα προγραμματισμού υψηλού επιπέδου (που είναι δηλαδή πλησιέστερα στην ανθρώπινη διατύπωση αλγοριθμικών λύσεων). Η μετάφραση από γλώσσα προγραμματισμού υψηλού επιπέδου

σε γλώσσα μηχανής γίνεται μηχανιστικά από ένα πρόγραμμα που λέγεται **μεταγλωττιστής** (compiler).

Η Πληροφορική πέρασε τα παιδικά της χρόνια τις δεκαετίες του 1940 και 1950 και έχει γίνει πια μια ολοκληρωμένη επιστήμη-τεχνολογία με ταχύτατη εξέλιξη στις προηγούμενες δεκαετίες και ευρύτατες προοπτικές για τον 21ο αιώνα.

1.3 Κλάδοι πληροφορικής

Ακολουθεί μια πρόχειρη απαρίθμηση διαφόρων κλάδων της Πληροφορικής. Κοινό χαρακτηριστικό όλων τους είναι στο περιεχόμενό τους: το θεωρητικό κομμάτι, η δημιουργία μοντέλων, και το σχεδιαστικό-κατασκευαστικό τους κομμάτι:

1. Αλγόριθμοι και Δομές Δεδομένων
2. Γλώσσες Προγραμματισμού
3. Αρχιτεκτονική Υπολογιστών και Δικτύων (hardware)
4. Αριθμητικοί και Συμβολικοί Υπολογισμοί
5. Λειτουργικά Συστήματα
6. Μεθοδολογία - Τεχνολογία Λογισμικού (software)
7. Βάσεις Δεδομένων και Διαχείριση Πληροφοριών
8. Τεχνητή Νοημοσύνη και Ρομποτική
9. Επικοινωνία ανθρώπου-υπολογιστή

Στο μάθημα "Εισαγωγή στην Επιστήμη των Υπολογιστών" του 4ου εξαμήνου θα επανέλθουμε για να συζητήσουμε περιληπτικά το περιεχόμενο αυτών των κλάδων.

1.4 Βασικές έννοιες

Ας στραφούμε όμως στους υπολογιστές και στον προγραμματισμό τους. Σχηματικά μπορούμε να πούμε:

$$\text{Υπολογιστής} = \text{Επεξεργαστής} + \text{Μνήμη} + \text{Συσκευές Εισόδου/Εξόδου}$$

όπου:

- ο **επεξεργαστής** (processor) εκτελεί εντολές που είναι αποθηκευμένες στη μνήμη και αλλάζει το περιεχόμενο της μνήμης,
- η **μνήμη** (memory) διατηρεί αποθηκευμένα δεδομένα και εντολές, και
- οι **συσκευές εισόδου/εξόδου** (input/output devices) καθιστούν δυνατή την επικοινωνία με ανθρώπους ή άλλους υπολογιστές.

Ιδιότητες του ηλεκτρονικού υπολογιστή:

- αυτόματο χωρίς εξυπνάδα

- μεγάλη ταχύτητα
- ακρίβεια στις πράξεις

Προγραμματισμός : Οργάνωση σειράς εντολών που υλοποιούν ένα συγκεκριμένο υπολογισμό.

Πρόγραμμα: Αλγόριθμος προς επίλυση κάποιου προβλήματος γραμμένος σε μια τυπική γλώσσα (με αυστηρούς συντακτικούς κανόνες) που λέγεται γλώσσα προγραμματισμού.

Προγραμματιστική Γλώσσα Μηχανής: Οι εντολές σχηματίζονται με δυαδικούς αριθμούς.

Π.χ.	0110110	11011011
	διεύθυνση στη μνήμη	καθαυτή εντολή

Γλώσσα χαμηλού επιπέδου (low level): Εντολές δυσνόητες για τους ανθρώπους (η χρήση τους δημιουργεί μεγάλα περιθώρια λάθους) και ευνόητες για μηχανές (μόνο όμως για τον συγκεκριμένο τύπο μηχανής για τον οποίο προορίζονται).

Γρήγορα δημιουργήθηκαν οι **μνημονικές γλώσσες** (assembly languages) που είναι πιο ευνόητες για τους ανθρώπους και μπορούν να μεταφραστούν εύκολα, μηχανιστικά, σε γλώσσα μηχανής. Η μετάφραση γίνεται από ένα πρόγραμμα, το συμβολομεταφραστή (assembler).

Π.χ.	L:	ADD	R1, R0
	συμβολική διεύθυνση ετικέτα (label)	πράξη (operation)	συμβολικές διευθύνσεις δεδομένων (variables)

Εξακολουθεί όμως να υπάρχει το πρόβλημα της δύσκολης μεταφοράς του προγράμματος σε μηχανή άλλου τύπου, δηλαδή διαφορετικής κατασκευής, (παρά το ότι με τις ετικέτες και τις μεταβλητές, που είναι σχετικές διευθύνσεις, επιτυγχάνεται η εύκολη μετακίνηση του προγράμματος μέσα στη μνήμη). Η αντιστοιχία με τις εντολές της γλώσσας μηχανής είναι μία προς μία, δηλαδή εξακολουθούμε να έχουμε πολλές λεπτομέρειες και επαναλήψεις, άρα μεγάλη πιθανότητα λαθών.

1.5 Γλώσσες υψηλού επιπέδου

Το επόμενο βήμα είναι οι γλώσσες υψηλού επιπέδου (high level): ευανάγνωστες, ανεξάρτητες από ένα συγκεκριμένο τύπο μηχανής και με μικρότερη πιθανότητα λάθους για τον προγραμματιστή. Π.χ.

```
(* σε Pascal *)
x := 3*y;
while x>1 do x := x/2
```

```
/* σε C */
x = 3*y;
while (x>1) x = x/2;
```

Η υλοποίηση (implementation) προγραμμάτων μιας γλώσσας υψηλού επιπέδου γίνεται είτε από πρόγραμμα-μεταγλωττιστή (compiler) ή από πρόγραμμα-διερμηνέα (interpreter) που εκτελεί αμέσως τις εντολές.

Ιστορικά πρώτη τέτοια γλώσσα ευρείας διάδοσης είναι η FORTRAN (FORmula TRANslator, 1957, Backus, IBM) με σκοπό την ευανάγνωστη γραφή μαθηματικών τύπων.

Μειονεκτήματα: όχι τελείως ανεξάρτητη από τον τύπο μηχανής. Είναι σχεδιασμένη μόνο για μαθηματικούς, φυσικούς, χημικούς κ.λπ. και οι πρώτες εκδόσεις της δεν υποστήριζαν δομημένο προγραμματισμό.

Πλεονεκτήματα: ευρεία διάδοση, έχουν γραφτεί και υπάρχουν σε βιβλιοθήκες πολλά προγράμματα σε FORTRAN.

Μερικές ενδιαφέρουσες ιστορικά γλώσσες, από τη FORTRAN μέχρι σήμερα:

- FORTRAN (1957, John Backus, IBM)
- LISP (1958, John McCarthy, MIT)
- Algol 60 (1960, IFIP, Rutishauser)
- COBOL (1960, CODASYL)
- BASIC (1964, Dartmuth College)
- PL/I (1964, IBM)
- Pascal (1971, Niklaus Wirth, ETH Zürich)
- Prolog (1972, Alain Colmerauer)
- C (1972, Brian Kernigham & Dennis Ritchie)
- ML (1973, Robin Milner, κ.α., Edinburgh)
- Modula-2 (1978, Niklaus Wirth, ETH Zürich)
- ADA (1980, Jean Ichbiah)
- Smalltalk 80 (1980, Xerox PARC)
- C++ (1983, Bjarne Stroustrup)
- Haskell (1990, Glasgow, UCL, Yale, Chalmers, Nottingham)
- Python (1991, Guido van Rossum)
- Ruby (1995, Yukihiro Matsumoto)
- Java (1995, James Gosling, Sun Microsystems)
- C# (2001, Microsoft)

Στο σχήμα 1.1 φαίνεται η εξέλιξη των πιο διαδεδομένων γλωσσών προγραμματισμού από τα τέλη της δεκαετίας του 1950 ως σήμερα.

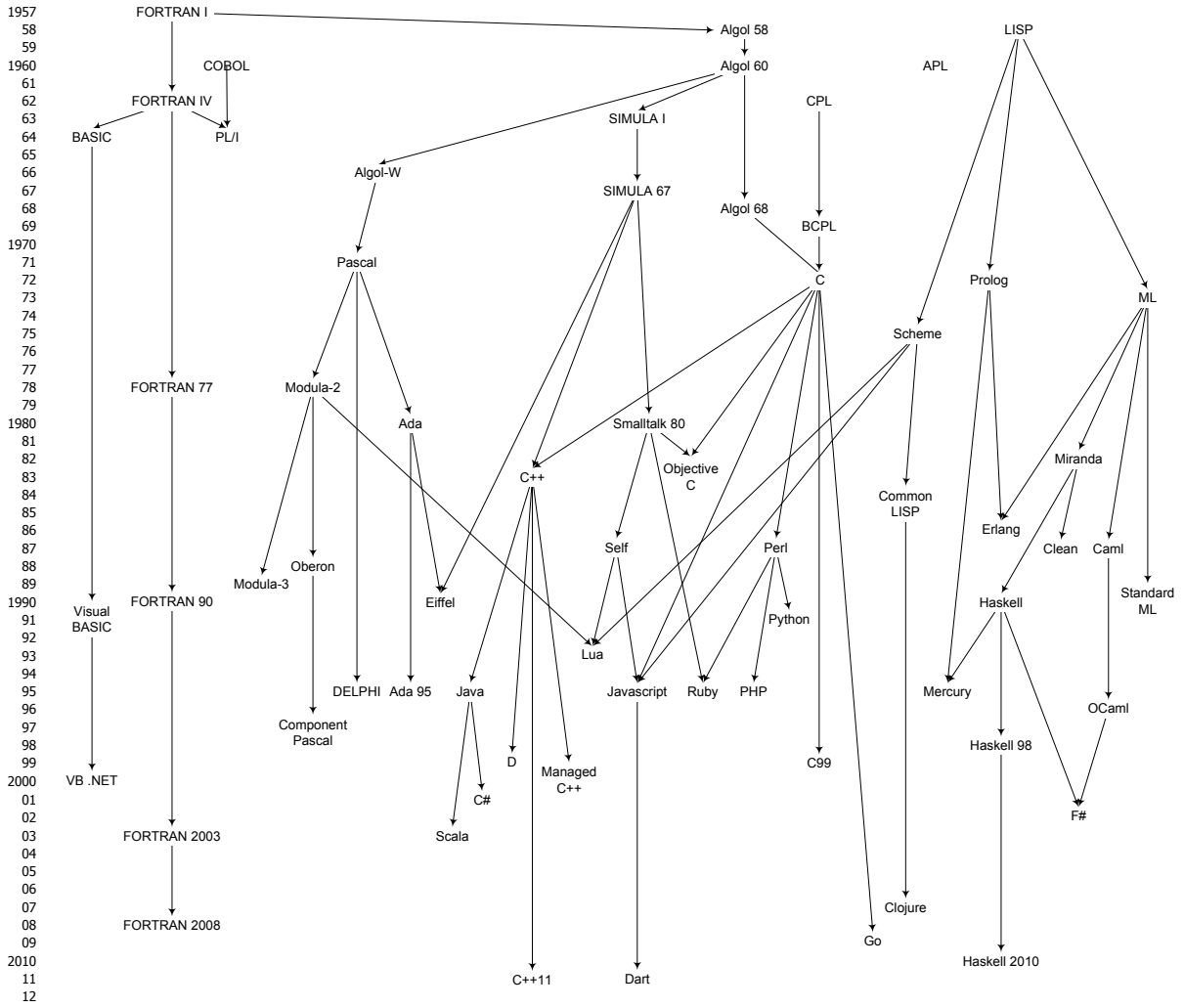
1.6 Hardware και software

Ας μιλήσουμε τώρα λίγο για τους Η/Υ αυτούς καθ' αυτούς. Ενα υπολογιστικό σύστημα αποτελείται βασικά από δύο μέρη:

- τον **υλικό εξοπλισμό** (hardware), και
- το **λογισμικό** (software), δηλαδή τα προγράμματα που επιτρέπουν και διευκολύνουν την χρήση των δυνατοτήτων του hardware.

Ο υλικός εξοπλισμός χωρίς λογισμικό είναι άχρηστος.

Πριν από το 1970 τα υπολογιστικά συστήματα ήταν μεγάλα (ονομαζόμενα mainframe), οργανωμένα σε υπολογιστικά κέντρα και χρησιμοποιούνταν από ένα πολύ μεγάλο αριθμό χρηστών κυρίως για τεχνολογικές-επιστημονικές και διοικητικές εφαρμογές. Το hardware ήταν πανάκριβο, το software όχι τόσο, γιατί γινόταν μόνο επαναληπτική χρήση λίγων βασικών απλών αλγορίθμων.



Σχήμα 1.1. Η εξέλιξη των πιο διαδεδομένων γλωσσών προγραμματισμού.

Οι τιμές του hardware άρχισαν και εξακολουθούν να πέφτουν, ενώ αντίθετα η τιμή του software συνεχίζει να ανεβαίνει λόγω πολύπλοκων αλγόριθμων, πολυσχιδών εφαρμογών και μεγαλυτέρων μισθών. Μετά το 1970 γενικεύτηκε η χρήση mini-υπολογιστικών συστημάτων (με 5-25 χρήστες). Μετά το 1980 έχουμε μεγάλη εξάπλωση των micro-υπολογιστών, συνεπώς και των personal computers (PC), δηλαδή συστημάτων για αποκλειστική χρήση από ένα άτομο. Από το 1990 και πέρα οι προσωπικοί υπολογιστές είναι οργανωμένοι σε δίκτυα υπολογιστών και επιτρέπουν την πρόσβαση σε πληθώρα πληροφοριών μέσω του Διαδικτύου (Internet). Σήμερα μηχανές με όλα τα χαρακτηριστικά ενός υπολογιστή συναντώνται και σε συσκευές διαφόρων μεγεθών με ποικιλία χρήσεων – π.χ. κινητά τηλέφωνα (smartphones) και ταμπλέτες (tablets) – ή σε ενσωματωμένα συστήματα (embedded systems) που επιτελούν πιο πολύπλοκες μηχανικές ή ηλεκτρονικές λειτουργίες. Οι εφαρμογές πλέον έχουν γενικευθεί σε πολλούς τομείς της ζωής μας. Από τον επιστήμονα-μηχανικό Πληροφορικής απαιτούνται πλέον γνώσεις ενός ολοκληρωμένου συστήματος (hardware-software).

Η έμφαση στο 1ο εξάμηνο είναι στην εξοικείωση με το software, ενώ στο 2ο εξάμηνο υπάρχουν δύο μαθήματα, ένα για software και ένα για hardware.

1.7 Ο πύργος της Βαβέλ και ο πόλεμος των γλωσσών

Από τη δεκαετία του 1950, χιλιάδες διαφορετικές γλώσσες προγραμματισμού έχουν σχεδιαστεί και υλοποιηθεί, οι περισσότερες από αυτές υψηλού επιπέδου. Πολλές σχεδιάζονται ακόμα κάθε χρόνο. Η συγκριτική μελέτη τους και η ταξινόμηση των χαρακτηριστικών τους είναι αντικείμενο του μαθήματος "Γλώσσες Προγραμματισμού" που διδάσκεται στο 6ο εξάμηνο.

Ποια είναι η καλύτερη γλώσσα προγραμματισμού; Γιατί, προφανώς, αυτή θα ήθελαν όλοι οι προγραμματιστές να χρησιμοποιούν και άρα αυτή θα έπρεπε να μάθουν οι πρωτοετείς σπουδαστές μας. Αυτή την ερώτηση την κάνουν οι περισσότεροι άνθρωποι που αρχίζουν να ασχολούνται με τον προγραμματισμό και συνήθως προκαλεί αρκετά έντονες συζητήσεις και ανταλλαγές απόψεων. Είναι όμως επίσης μία πολύ ασαφής ερώτηση. Όταν ασχοληθείτε με τον προγραμματισμό λίγο περισσότερο, ας πούμε, μέχρι να τελειώσετε τις σπουδές σας στο Ε.Μ.Π., σίγουρα θα έχετε μάθει ότι "η καλύτερη γλώσσα προγραμματισμού" δεν υπάρχει. Κάποιες γλώσσες είναι καλύτερες από κάποιες άλλες **για κάποιο συγκεκριμένο σκοπό**. Για την ακρίβεια, κάποιες γλώσσες χρησιμοποιούνται σχεδόν αποκλειστικά για κάποιους συγκεκριμένους σκοπούς και για κάποιους συγκεκριμένους σκοπούς χρησιμοποιούνται σχεδόν αποκλειστικά κάποιες γλώσσες.

Για παράδειγμα, η **C** είναι μία εξαιρετικά καλή γλώσσα για τον προγραμματισμό συστημάτων (systems programming). Αναπτύχθηκε κατά τη διάρκεια της περιόδου 1969-73 στα εργαστήρια AT&T Bell Labs ως μια γλώσσα υλοποίησης για το λειτουργικό σύστημα Unix. Πατέρας της είναι ο Dennis Ritchie, ο οποίος επίσης ανέπτυξε τον πρώτο μεταγλωττιστή στην περίοδο 1971-73. Η C είναι μία γλώσσα σχετικά **χαμηλού επιπέδου** (low-level), δηλαδή παρέχει στον προγραμματιστή τη δυνατότητα να αλληλεπιδρά σχετικά άμεσα με το hardware, και για το λόγο αυτό είναι κατάλληλη για τον προγραμματισμό συστημάτων. Οι μεταγλωττιστές της C έχουν τη φήμη ότι παράγουν προγράμματα που εκτελούνται γρήγορα (αν και σύντομα θα δούμε ότι τον πρώτο λόγο στο πόσο γρήγορα εκτελείται ένα πρόγραμμα δεν τον έχει η γλώσσα στην οποία γράφτηκε ή ο μεταγλωττιστής που την υλοποιεί αλλά ο αλγόριθμος που χρησιμοποιήθηκε από τον προγραμματιστή).

Η C σήμερα μάλλον δε θεωρείται η καλύτερη γλώσσα για αριθμητικούς ή/και επιστημονικούς υπολογισμούς (numerical/scientific computing), παρότι πολλοί τη χρησιμοποιούν για αυτό το σκοπό και υπάρχει πλήθος σχετικών προγραμμάτων και βιβλιοθηκών γραμμένων σε αυτήν. Η πρώτη γλώσσα που χρησιμοποιήθηκε ευρέως για τέτοιου είδους υπολογισμούς είδαμε ότι ήταν η FORTRAN — χρησιμοποιείται ακόμη σήμερα από αρκετούς επιστήμονες. Όμως, μία μεγάλη κοινότητα μαθηματικών και μηχανικών σήμερα προγραμματίζουν στις ειδικές γλώσσες εργαλείων όπως το MATLAB και το Mathematica. Αυτές οι γλώσσες είναι ιδανικές για αριθμητικούς και επιστημονικούς υπολογισμούς, δύσκολα όμως θα μπορούσαν να χρησιμοποιηθούν για κάτι άλλο, π.χ. τον προγραμματισμό συστημάτων ή βάσεων δεδομένων.

Στον αντίποδα (κατά τη γνώμη των συγγραφέων αυτών των σημειώσεων, τουλάχιστον, γιατί και αυτό το ζήτημα είναι υποκειμενικό και φυσικά υπάρχει αντίλογος), η C είναι μία πολύ κακή γλώσσα για να διδάξει κανείς προγραμματισμό. Ορισμένα χαρακτηριστικά της είναι τόσο χαμηλού επιπέδου που τείνουν να φέρουν το hardware του υπολογιστή στο επίκεντρο της προσοχής, αντί τους αλγορίθμους που υλοποιούν το software. Όταν μαθαίνεις προγραμματισμό, δε χρειάζεσαι δώδεκα διαφορετικούς τύπους ακέραιων αριθμών (μέσα στους οποίους μετράμε και τους χαρακτήρες και τις λογικές τιμές) και άλλους τρεις τύπους αριθμών κινητής υποδιαστολής (πραγματικών). Δε χρειάζεσαι μία δομή `for` τόσο ισχυρή που να μπορείς να γράψεις όλο τον αλγόριθμο της δυαδικής αναζήτησης (βλ. κεφ. 5) σε μία γραμμή προγράμματος. Δε χρειάζεται επίσης να πρέπει να καταλάβεις τι κάνει το `x = x++;` (δεν κάνει τίποτα γιατί είναι παράνομο,

όπως θα δούμε στο κεφάλαιο 9).

Η **Pascal** είναι σίγουρα μια από τις καλύτερες γλώσσες προγραμματισμού για την εκπαίδευση. Είναι γλώσσα γενικού σκοπού (general purpose), δηλαδή δεν αποσκοπεί σε κάποια συγκεκριμένη περιοχή εφαρμογών αλλά προσπαθεί να καλύπτει όσο γίνεται καλύτερα ένα μεγάλο εύρος. Υποστηρίζει την συστηματική, δομημένη, αλγοριθμική επίλυση προβλημάτων, είναι ευανάγνωστη άρα τα προγράμματα αυτά καθ' αυτά αποτελούν μια πρώτη τεκμηρίωση, έχει ξεκάθαρα δομικά στοιχεία που βοηθούν στη βαθμιαία ανάπτυξη προγραμμάτων, έχει πλεονασμούς και περιορισμούς που όμως βοηθούν στο να αποφευχθούν λάθη και να γίνεται επαλήθευση της ορθότητας και τέλος έχει σύντομο αυστηρό και απλό συντακτικό ορισμό. Η Pascal σχεδιάστηκε και υλοποιήθηκε από μια πολύ μικρή ομάδα ανθρώπων με επικεφαλής των Niklaus Wirth, μεταξύ 1968-1971, με βασική αρχή των περιορισμένων στόχων. Από την άλλη πλευρά, η Pascal και οι άμεσοι απόγονοί της χρησιμοποιούνται σήμερα ελάχιστα στη βιομηχανία λογισμικού.

1.8 Η γλώσσα Pazcal

Φτάσαμε λοιπόν στην Pazcal, τη γλώσσα που χρησιμοποιούμε στο μάθημα του 1ου εξαμήνου για τη διδασκαλία του προγραμματισμού. Πήραμε ένα "εκπαιδευτικό υποσύνολο" της C, αφήνοντας έξω ό,τι δε χρειάζεται να βλέπει κάποιος που τώρα μαθαίνει προγραμματισμό για πρώτη φορά. Προσθέσαμε μερικές επεκτάσεις που ακολουθούν την εκπαιδευτική φιλοσοφία της Pascal, χωρίς όμως να αλλοιώνουν το χαρακτήρα της αρχικής γλώσσας. Το αποτέλεσμα είναι η Pazcal, μια γλώσσα σχεδιασμένη έτσι ώστε να συνδυάζει τα καλά χαρακτηριστικά της Pascal και της C. Θα σας συνοδεύσει στο μεγαλύτερο μέρος του 1ου εξαμήνου. Αφού κάνετε τα πρώτα βήματα στον προγραμματισμό με αυτήν, θα σας είναι εύκολο να την αφήσετε πίσω σας και να προγραμματίζετε σε C.

Στη σχεδίαση της Pazcal κάναμε μερικές πολύ σημαντικές επιλογές. Αν τις έχετε στο μυαλό σας, θα καταλάβετε εύκολα μερικά πράγματα που διαφορετικά ίσως σας παραξενέψουν.

1. **Η βασική εξίσωση!**

   $$\text{Pazcal} = \text{C}^{\textcircled{\scriptsize ☺}} + \text{“}\texttt{pazcal.h}\text{”}$$

   όπου $\text{C}^{\textcircled{\scriptsize ☺}} \subset$ C είναι το "εκπαιδευτικό υποσύνολο" της ANSI C99 και "`pazcal.h`" ένα αρχείο επικεφαλίδων (header file) που ορίζει μερικές "εκπαιδευτικές" επεκτάσεις (ορισμούς τύπων, σταθερών και macros).

2. **Τι λείπει από τη C:** Η $\text{C}^{\textcircled{\scriptsize ☺}}$ δεν περιέχει όσα χαρακτηριστικά της C κρίναμε ως ανεπιθύμητα για τους σκοπούς ενός εισαγωγικού μαθήματος στον προγραμματισμό. Τα πιο σημαντικά από αυτά που λείπουν περιγράφονται στο Κεφάλαιο 9. Για μία πλήρη περιγραφή της ANSI C99 ανατρέξτε στη σχετική βιβλιογραφία.

3. **Τι έχει προστεθεί που δεν υπάρχει στη C:** Είναι πολύ εύκολο να ξεχωρίσει κανείς σε αυτές τις σημειώσεις τι είναι "κανονική C" και τι είναι επέκταση της Pazcal. Οι επεκτάσεις της Pazcal γράφονται πάντα με κεφαλαία γράμματα, π.χ. η "εντολή" **`WRITE`** (η αντίστοιχη συνάρτηση βιβλιοθήκης της C είναι η `printf`, με μικρά γράμματα). Αυτό εξηγεί γιατί, π.χ., στο επόμενο κεφάλαιο οι τύποι **`int`**, **`char`** και **`bool`** γράφονται με μικρά γράμματα ενώ ο τύπος **`REAL`** γράφεται με κεφαλαία: οι πρώτοι είναι τύποι της "κανονικής C" ενώ ο τελευταίος είναι τύπος της Pazcal (ο αντίστοιχος τύπος της C ονομάζεται **`double`**).

1.9 Ασκήσεις

Πληκτρολογήστε και εκτελέστε τα παρακάτω προγράμματα Pazcal:

```
PROGRAM hello1a ()
{
  WRITELN(”hello world”);
}
```

```
PROGRAM hello1b ()
{
  WRITESPLN(”hello”, ”world”);
}
```

```
PROGRAM hello1c ()
{
  WRITE(”hello ”); WRITELN(”world”);
}
```

```
PROGRAM hello1d ()
{
  WRITESP(”hello”, ”world”); WRITELN();
}
```

```
PROC hello ()
{
  WRITELN(”hello world”);
}

PROGRAM hello2 ()
{
  hello(); hello(); hello(); hello();
}
```

```
PROC hello ()
{
  WRITELN(”hello world”);
}

PROGRAM hello3 ()
{
  int i;
  FOR(i, 1 TO 20)
    hello();
}
```

```
const int n = 20;
int i;

PROC num_hello ()
{
  WRITESPLN(i, ”hello world”);
}

PROGRAM hello4 ()
{
  FOR(i, 1 TO n)
    num_hello();
}
```

```
PROC hello ()
{
  WRITELN(”hello world”);
}

PROGRAM hello5 ()
{
  int i, n;
  WRITESP(”Give number of greetings”,
          ”then press <enter>: ”);
  n = READ_INT();
  FOR(i, 1 TO n)
    hello();
}
```

```
PROC hello ()
{
  WRITELN(”hello world”);
}

PROGRAM hello6 ()
{
  int i, n;
  WRITESP(”Give number of greetings”,
          ”then press <enter>: ”);
  n = READ_INT();
  if (n < 0)
    WRITESPLN(”The number”, n,
              ”is negative”);
  else
    FOR(i, 1 TO n)
      hello();
}
```

Κεφάλαιο 2

Δομή προγράμματος

Η Pazcal είναι μια γλώσσα προγραμματισμού, δηλαδή μια τυπική γλώσσα (με αυστηρούς συντακτικούς κανόνες) με την οποία περιγράφουμε τον τρόπο επίλυσης ενός προβλήματος.

Η περιγραφή αυτή του αλγορίθμου (τρόπου επίλυσης κάποιου προβλήματος) και γενικά του τρόπου πραγματοποίησης κάποιας εργασίας απο τον υπολογιστή λέγεται πρόγραμμα.

Παράδειγμα:

```
PROGRAM example1()
```
επικεφαλίδα

```
{
    REAL r, a;     // Χρειάζονται δύο μεταβλητές: ακτίνα και εμβαδό
```
δηλώσεις

```
    WRITE("Give the radius: ");
    r = READ_REAL();                    // Είσοδος ακτίνας
    a = 3.1415926 * r * r;              // Υπολογισμός εμβαδού
    WRITELN("The area is: ", a);        // Εμφάνιση αποτελέσματος
```
εντολές

```
}
```
σώμα = block

Ένα απλό πρόγραμμα γραμμένο σε γλώσσα Pazcal, αποτελείται απο τρια βασικά ξεχωριστά μέρη, σα μια συνταγή μαγειρικής:

- *Επικεφαλίδα* (όνομα του φαγητού): Όνομα του προγράμματος.
- *Δηλώσεις* (υλικά που χρειάζονται): Αναγνωριστικά ονόματα μεταβλητών, διαδικασιών κ.ο. που θα χρησιμοποιηθούν (declarations).
- *Εντολές* (οδηγίες για μαγείρεμα): Τι θέλουμε να εκτελεστεί (statements).

Οι δηλώσεις και οι εντολές γράφονται μέσα σε άγκιστρα και σχηματίζουν από κοινού το "σώμα" του προγράμματος (block).

Παρατηρήσεις:

- Τα άγκιστρα χρησιμεύουν ως παρενθέσεις που υποδηλώνουν την αρχή και το τέλος του σώματος (block).

- Σχόλια (comments) που διαβάζονται αλλά δε μεταφράζονται (δηλαδή αγνοούνται) απο τον compiler υπάρχουν δύο ειδών. Το πρώτο από αυτά φαίνεται στο προηγούμενο παράδειγμα: ξεκινούν με `//` και εκτείνονται μέχρι το τέλος της γραμμής. Το δεύτερο είδος σχολίων μπορούν να εκτείνονται και σε περισσότερες γραμμές: ξεκινούν με `/*` και τελειώνουν με την πρώτη εμφάνιση του `*/` που ακολουθεί. Για παράδειγμα:

  `/*` Αυτό είναι ένα σχόλιο πολλών γραμμών.
  Συνεχίζεται και παρακάτω.
  Ακόμα όμως κι αν φαίνεται να περιέχει ίδια `/*` σχόλια μέσα του,
  τελειώνει την πρώτη φορά που εμφανίζεται αστεράκι και κάθετος,
  δηλαδή στην αρχή της επόμενης γραμμής...
  `*/`

- Οι λέξεις με **έντονη** γραφή ονομάζονται δεσμευμένες λέξεις (reserved words) ή λέξεις κλειδιά (keywords) και θεωρούνται ένα σύμβολο η κάθε μια (**`PROGRAM`**, **`REAL`**, **`WRITE`**, κ.λπ.). Συμπτωματικά στο προηγούμενο παράδειγμα όλες οι λέξεις κλειδιά ήταν γραμμένες με κεφαλαία γράμματα, θα δούμε όμως ότι υπάρχουν και λέξεις κλειδιά γραμμένες με πεζά γράμματα (π.χ. **`int`**) και ότι η διάκριση μεταξύ κεφαλαίων και πεζών γραμμάτων είναι σημαντική στην Pazcal.

- Οι μεταβλητές `r` και `a` είναι ονόματα περιοχών μνήμης. Σύμφωνα με τις δηλώσεις του προγράμματος, σε κάθε μία από αυτές τις μεταβλητές μπορεί να αποθηκευτεί ένας πραγματικός αριθμός (**`REAL`**). Οι μεταβλητές πρέπει να δηλώνονται πριν χρησιμοποιηθούν. Τα ονόματά τους (αναγνωριστικά — identifiers) τα επιλέγει ο χρήστης, δεν μπορούν όμως να είναι δεσμευμένες λέξεις.

- Η εντολή **`WRITE`**`("Give the radius: ")` εμφανίζει στην οθόνη τη συμβολοσειρά "`Give the radius: `" (προσοχή, υπάρχει ένα κενό διάστημα μετά την άνω-κάτω τελεία). Αυτό είναι ένα μήνυμα προς το χρήστη του προγράμματος, ώστε να ξέρει τι του ζητείται να κάνει στη συνέχεια.

- Η εντολή `r = READ_REAL();` διαβάζει από το πληκτρολόγιο έναν πραγματικό αριθμό (τον οποίο πρέπει να πληκτρολογήσει ο χρήστης) και τον αναθέτει (δηλαδή τον αποθηκεύει) στη μεταβλητή `r`.

- Η εντολή `a = 3.1415926 * r * r;` υπολογίζει το εμβαδό ενός κύκλου με ακτίνα `r` και το αποθηκεύει στη μεταβλητή `a`. Χρησιμοποιεί τον τύπο $\mathtt{a} = \pi \mathtt{r}^2$. Υπολογίζει πρώτα το δεξί μέλος χρησιμοποιώντας το περιεχόμενο της θέσης μνήμης που ονομάζεται `r` (δηλαδή την ακτίνα). Στη συνέχεια αποθηκεύει το αποτέλεσμα στη θέση μνήμης που ονομάζεται `a`.

- Η εντολή **`WRITELN`**`("The area is: ", a);` εμφανίζει στην οθόνη πρώτα τη συμβολοσειρά "`The area is: `" και στη συνέχεια, δίπλα της, το περιεχόμενο της μεταβλητής (=θέσης μνήμης) με όνομα `a`. Έτσι εμφανίζεται στην οθόνη η τιμή του εμβαδού του κύκλου.

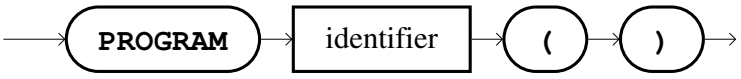
2.1 Σύνταξη του προγράμματος

Ας δούμε το κάθε μέρος ενός απλού προγράμματος ξεχωριστά:

Επικεφαλίδα (program header)

Είναι το πρώτο πράγμα που συναντάμε σε κάθε απλό πρόγραμμα και το πιο εύκολο. Στην επικεφαλίδα ορίζουμε το όνομα του προγράμματός μας: μια λέξη-αναγνωριστικό με την οποία θα βαφτίσουμε το πρόγραμμά μας. Η σύνταξη της επικεφαλίδας είναι:

program_header



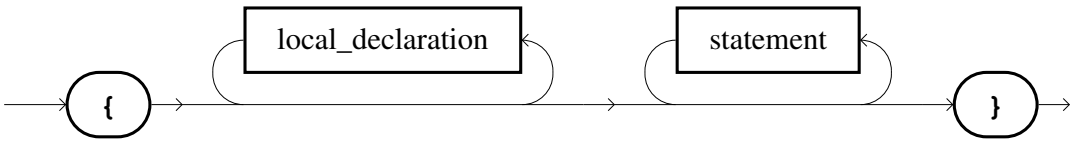
Ο συμβολισμός που χρησιμοποιούμε για να περιγράψουμε τη σύνταξη των τμημάτων που αποτελούν ένα πρόγραμμα ονομάζεται **συντακτικό διάγραμμα**. Αυτό που είναι μέσα σε οβάλ κουτί πρέπει να γραφτεί ακριβώς έτσι, ενώ αυτό που είναι σε ορθογώνιο θέλει περαιτέρω επεξήγηση. Τα βέλη ορίζουν τη σειρά με την οποία γράφονται τα μέρη του συντακτικού διαγράμματος.

Εδώ, στη θέση της λέξης identifier βάζουμε ένα αναγνωριστικό όνομα. (Δες παρακάτω για το τι επιτρέπεται να είναι ένα αναγνωριστικό όνομα). Παραδείγματα επικεφαλίδων με ορθά ονόματα είναι :

```
PROGRAM george()
PROGRAM programma1()
PROGRAM to_programma_mou()
```

Μετά την επικεφαλίδα ακολουθεί το σώμα (block) του προγράμματος, που με τη σειρά του αποτελείται από το τμήμα δηλώσεων και το τμήμα των εντολών, μέσα σε άγκιστρα. Οι δηλώσεις που βρίσκονται μέσα σε ένα block λέγονται **τοπικές** (local) γιατί ισχύουν μόνο μέσα στο εσωτερικό αυτού του block.

block



Προσέξτε τους δύο "κύκλους" που σχηματίζουν τα βέλη στο παραπάνω συντακτικό διάγραμμα: ένα block περιβάλλεται υποχρεωτικά από άγκιστρα και περιέχει πρώτα τις τοπικές δηλώσεις και μετά τις εντολές. Μπορούν να υπάρχουν οσεσδήποτε τοπικές δηλώσεις (μηδέν, μία ή περισσότερες), η μία μετά την άλλη, και στη συνέχεια οσεσδήποτε εντολές (μηδέν, μία ή περισσότερες).

Είναι δυνατόν ένα block που κάνει κάτι πολύ απλό να μη χρειάζεται να χρησιμοποιήσει μεταβλητές. Στην περίπτωση αυτή, όπως φαίνεται στο συντακτικό διάγραμμα μπορούμε να παραλείψουμε τελείως τις τοπικές δηλώσεις και να περάσουμε κατευθείαν στις εντολές. Επίσης,

είναι δυνατόν ένα block να μην έχει καμία εντολή, αν και στην περίπτωση αυτή το block μάλλον δε θα είναι και πολύ χρήσιμο.

Τοπικές δηλώσεις (local declarations)

Στο μέρος των τοπικών δηλώσεων ενός block ορίζουμε (δηλώνουμε) διάφορες μεταβλητές και σταθερές που θα χρησιμοποιήσουμε στη συνέχεια, μέσα στο block. Θα ξεκινήσουμε από τη δήλωση **μεταβλητών**.

Όπως ξέρουμε ένα κομμάτι του υπολογιστή είναι η μνήμη του. Στη μνήμη του ένας υπολογιστής αποθηκεύει διάφορα μεγέθη (δεδομένα) για να μπορεί να τα επεξεργάζεται. Σε ένα πρόγραμμα, τα μεγέθη που μας ενδιαφέρουν και που θέλουμε να κρατηθούν στη μνήμη του υπολογιστή για επεξεργασία, αποθηκεύονται στις μεταβλητές.

Θα φανταστούμε κάθε μεταβλητή σαν ένα "κουτάκι μνήμης" του υπολογιστή, το οποίο περιέχει δεδομένα. Ας πάρουμε για παράδειγμα το πρόγραμμα της αρχής του κεφαλαίου, που ζητάει από το χρήστη το μήκος της ακτίνας ενός κύκλου και υπολογίζει το εμβαδό του:

```
PROGRAM example1()
{
  REAL r, a;

  WRITE(”Give the radius: ”);
  r = READ_REAL();
  a = 3.1415926 * r * r;
  WRITELN(”The area is: ”, a);
}
```

Προσέξτε τα εξής:

- Χρειαστήκαμε δύο "κουτάκια μνήμης" (μεταβλητές) για να αποθηκεύσουμε την ακτίνα και το εμβαδό του κύκλου. Στα δύο αυτά κουτάκια δώσαμε τα ονόματα `r` και `a` αντίστοιχα.
- Η μεταβλητή `r` παίρνει την τιμή της με την εντολή `r = READ_REAL();`, η οποία ζητά από το χρήστη του προγράμματος να πληκτρολογήσει έναν πραγματικό αριθμό. Στη συνέχεια, η μεταβλητή `a` παίρνει την τιμή της με την εντολή `a = 3.1415926 * r * r;`. Για να υπολογιστεί η τιμή του `a`, ο υπολογιστής χρειάζεται να ανατρέξει στη μνήμη του, στη μεταβλητή `r`, για να θυμηθεί ποια τιμή έχει δώσει ο χρήστης.

Στο μέρος των τοπικών δηλώσεων του block, όπως είπαμε, πρέπει να ορίσουμε τις μεταβλητές που θα χρησιμοποιήσουμε. Ο ορισμός γίνεται με τη δήλωση:

```
REAL r, a;
```

Στην παραπάνω γραμμή, `r` και `a` είναι τα ονόματα των δύο μεταβλητών, δηλαδή τα αναγνωριστικά με τα οποία βαφτίζουμε τις θέσεις αποθήκευσης στη μνήμη. Όπως σε όλα τα αναγνωριστικά, επιτρέπονται γράμματα μικρά και κεφαλαία (που θεωρούνται διαφορετικά), τα ψηφία 0 ως 9 καθώς και η κάτω παύλα (χαρακτήρας υπογράμμισης — underscore). Όμως για αναγνωριστικά δεν πρέπει να χρησιμοποιούνται δεσμευμένες λέξεις. Τα αναγνωριστικά αρχίζουν πάντα με γράμμα.

Ο τύπος **REAL** προσδιορίζει το πεδίο δυνατών τιμών κάθε μίας από τις δύο μεταβλητές. Στην Pazcal χρειάζεται να δηλώνουμε για κάθε μεταβλητή τι τιμές θα αποθηκεύονται σε αυτήν, π.χ. αριθμοί ακέραιοι ή πραγματικοί, γράμματα και χαρακτήρες, κ.λπ.

Η Pazcal υποστηρίζει τους εξής απλούς τύπους:

Απλοί τύποι	Χρήση	Σχόλια
int	Για αποθήκευση ακεραίου αριθμού	Δεν μπορεί φυσικά να αποθηκευτεί οσοδήποτε μεγάλος αριθμός
REAL	Για αποθήκευση πραγματικού αριθμού	Δεν μπορεί φυσικά να αποθηκευτεί οποιοσδήποτε αριθμός
char	Για αποθήκευση χαρακτήρα	Δες πίνακα ASCII στο παράρτημα
bool	Για αποθήκευση λογικής τιμής	Μόνες δυνατές τιμές **true**, **false**

Προσέξτε ότι, από αυτούς, μόνο ο τύπος **REAL** γράφεται με κεφαλαία γράμματα.

Γενικά όταν θέλουμε να δηλώσουμε πολλές μεταβλητές και διαφορετικών τύπων, γράφουμε δηλώσεις όπως οι παρακάτω:

```
int i, j, k;
REAL x, y;
char ch;
bool passed;
int m;
```

Εδώ δηλώνουμε ότι θα χρησιμοποιήσουμε τρεις μεταβλητές `i`, `j` και `k` για ακεραίους (τύπου **int**), δυο μεταβλητές `x`, `y` τύπου **REAL**, μια μεταβλητή `ch` τύπου **char**, μια μεταβλητή `passed` τύπου **bool** και άλλη μια μεταβλητή `m` τύπου **int** (που θα μπορούσαμε, αν θέλαμε, να την είχαμε βάλει μαζί με τις `i`, `j` και `k`).

Ας πούμε λίγα λόγια για τον τύπο **bool** (συχνά το διαβάζουμε "Boolean", από το όνομα του άγγλου Μαθηματικού George Boole, θεμελιωτή της άλγεβρας των λογικών τιμών). Μεταβλητές αυτού του τύπου μπορούν να πάρουν μόνο δύο δυνατές τιμές: **true** (αληθής ή "ναι") και **false** (ψευδής ή "όχι").

Για παράδειγμα, έστω ότι φτιάχνουμε έναν κατάλογο με φοιτητές. Για κάθε φοιτητή μπορούμε να κρατήσουμε μια μεταβλητή τύπου **bool** με όνομα `passed` που θα μας λέει αν ο φοιτητής πέρασε το μάθημα "Προγραμματισμός Ηλεκτρονικών Υπολογιστών". Αν η `passed` έχει την τιμή **true** τότε ο φοιτητής πέρασε το μάθημα, αλλιώς (αν δηλαδή η `passed` έχει την τιμή **false**) τότε το χρωστάει.

Εντολές (statements)

Οι εντολές που βρίσκονται μέσα σε ένα block γράφονται με τη σειρά που θέλουμε να τις εκτελέσει ο υπολογιστής. Οι περισσότερες απλές εντολές στην Pazcal τελειώνουν με ένα ελληνικό ερωτηματικό `;` (γνωστό και ως semicolon), αυτό όμως είναι μέρος της σύνταξης των επιμέρους εντολών. Περισσότερα για τις εντολές της Pazcal θα δούμε αργότερα, στη συνέχεια αυτού του κεφαλαίου και στα επόμενα κεφάλαια.

Σε πολλά σημεία μέσα στο πρόγραμμα μπορούμε να προσθέσουμε σχόλια, δηλαδή επεξηγηματικές φράσεις, που δεν μεταφράζονται από τον compiler, και εμπλουτίζουν περιγραφικά τον κώδικα του προγράμματός μας. Τα σχόλια αυτά, όπως είπαμε, είναι δύο ειδών: είτε αρχίζουν

με τους χαρακτήρες `//` και εκτείνονται μέχρι το τέλος της γραμμής, είτε γράφονται μέσα σε καθέτους και αστερίσκους, π.χ. /* *Αυτό είναι ένα σχόλιο* */. Αν και η Pazcal επιτρέπει σχόλια σε διάφορα σημεία του προγράμματος, καλό είναι αυτά να μπαίνουν σε ξεχωριστές γραμμές (ή στο δεξί άκρο μιας γραμμής).

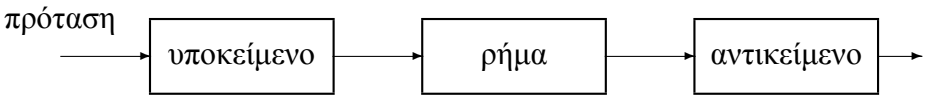
2.2 Τι σημαίνει ορθό πρόγραμμα

Ένα πρόγραμμα για να λειτουργεί σωστά, δηλαδή, για να δίνει τα αναμενόμενα αποτελέσματα πρέπει να είναι ορθό (correct). Διακρίνουμε τρία επίπεδα ορθότητας:

- **Συντακτική ορθότητα.**

  Ένα πρόγραμμα για να είναι συντακτικά ορθό πρέπει να υπακούει στους συντακτικούς κανόνες της γλώσσας προγραμματισμού στην οποία γράφεται. Όπως στη φυσική μας γλώσσα για να είναι μια πρόταση ολοκληρωμένη πρέπει να περιέχει υποκείμενο, ρήμα και αντικείμενο (π.χ. μη συντακτικά ορθή πρόταση: "εμένα παίρνει να τρέχεις", ενώ συντακτικά ορθή: "ο σκύλος τρώει λουκάνικο"), έτσι και σε μια γλώσσα προγραμματισμού το πρόγραμμα, τα ονόματα των μεταβλητών, τα υποπρογράμματα και οι εντολές που χρησιμοποιούμε σε αυτό πρέπει να υπακούνε σε ορισμένους συντακτικούς κανόνες, στους συντακτικούς κανόνες με τους οποίους έχει οριστεί η γλώσσα. Μάλιστα, οι κανόνες αυτοί είναι πολύ πιο αυστηροί από τους συντακτικούς κανόνες της φυσικής μας γλώσσας καθώς ο υπολογιστής δεν έχει τις ικανότητες του ανθρώπινου μυαλού που ακόμα και αν κάτι δεν είναι απόλυτα συντακτικά ορθό μπορεί να καταλάβει τί σημαίνει και να προχωρήσει παρακάτω. Ένας τρόπος που ορίζονται οι συντακτικοί κανόνες είναι με τα συντακτικά διαγράμματα (είδαμε μερικά παραδείγματα και θα δούμε περισσότερα στη συνέχεια).

  Τα συντακτικά διαγράμματα θα μπορούσαν να εφαρμοστούν και στις φυσικές γλώσσες. Για παράδειγμα, για την Ελληνική γλώσσα στην οποία συνήθως μια πρόταση αποτελείται απο υποκείμενο, ρήμα και αντικείμενο (ας υποθέσουμε αυστηρά με αυτή τη σειρά), θα μπορούσαμε να γράψουμε:



- **Νοηματική ορθότητα.**

  Στη φυσική μας γλώσσα μια πρόταση για να είναι ορθή δεν αρκεί να είναι συντακτικά ορθή, θα πρέπει να είναι και νοηματικά ορθή (π.χ. η πρόταση: "ο σκύλος τρώει το σπίτι" δεν είναι νοηματικά ορθή παρότι είναι συντακτικά ορθή). Έτσι και στα προγράμματά μας μπορεί παρόλο που έχουμε συντακτικά ορθές δηλώσεις και εντολές, αυτές να μη συνδυάζονται με σωστό τρόπο. Π.χ. δεν επιτρέπεται να προσθέσουμε σε έναν αριθμό ένα γράμμα της αλφαβήτου. Ένας άλλος βασικός κανόνας είναι πως ό,τι χρειαζόμαστε σε ένα πρόγραμμα πρέπει πρώτα να δηλωθεί.

- **Σημασιολογική ορθότητα.**

  Μια πρόταση μπορεί να είναι σημασιολογικά λανθασμένη παρότι είναι και συντακτικά και νοηματικά ορθή, π.χ. λέω "δώσε μου το μολύβι" ενώ στην πραγματικότητα χρειάζομαι

τη γόμμα. Ένα πρόγραμμα, αντίστοιχα, μπορεί να κάνει κάτι που εμείς δεν θέλουμε ή δεν περιμένουμε. Τέτοια λάθη είναι σημαντικά και όχι απλώς παραβλέψεις νοηματικές ή παραβάσεις συντακτικές. Συνεπώς, αυτά τα λάθη είναι λάθη κακού σχεδιασμού ή κακής υλοποιήσης του προγράμματος.

Στα προγράμματα Pazcal ο compiler μπορεί να ελέγχει για τα δύο πρώτα επίπεδα ορθότητας, για παράδειγμα:

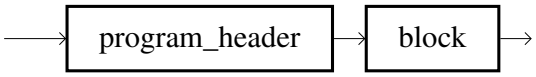
- δεν μας επιτρέπει να παραλείψουμε το όνομα του προγράμματος ή τα άγκιστρα που περικλείουν τις δηλώσεις και τις εντολές ενός block, και
- δεν μας επιτρέπει να προσθέσουμε σε έναν αριθμό ένα γράμμα της αλφαβήτου.

Όταν ο compiler διαπιστώσει τέτοιο λάθος βγάζει στην οθόνη ένα μήνυμα (**error message**) που περιλαμβάνει τη γραμμή του προγράμματος όπου βρίσκεται το λάθος και μια (πιθανή) περιγραφή του λάθους, χωρίς να ολοκληρώσει την μεταγλώττιση του προγράμματος. Πρέπει να εξαλειφθούν όλα τα λάθη που βρίσκει ο compiler από ένα πρόγραμμα πριν να ολοκληρωθεί η μεταγλώττισή του. Το τρίτο όμως επίπεδο ορθότητας πρέπει να το ελέγξουμε εμείς με δοκιμές (testing) ή/και με επαλήθευση (verification). Έτσι, τα σημασιολογικά λάθη είναι τα πιο δύσκολα να εντοπισθούν.

Παραδείγματα συντακτικών διαγραμμάτων.

Σε ένα συντακτικό διάγραμμα ό,τι βρίσκεται μέσα σε κύκλο ή σε οβάλ είναι σύμβολο της γλώσσας και εμφανίζεται στο κείμενο όπως έχει, ενώ ό,τι βρίσκεται σε ορθογώνιο είναι βοηθητική έννοια που χρειάζεται περαιτέρω εξήγηση, π.χ. με ένα άλλο συντακτικό διάγραμμα (που μπορεί να εμφανίζεται πιο πριν ή αργότερα).

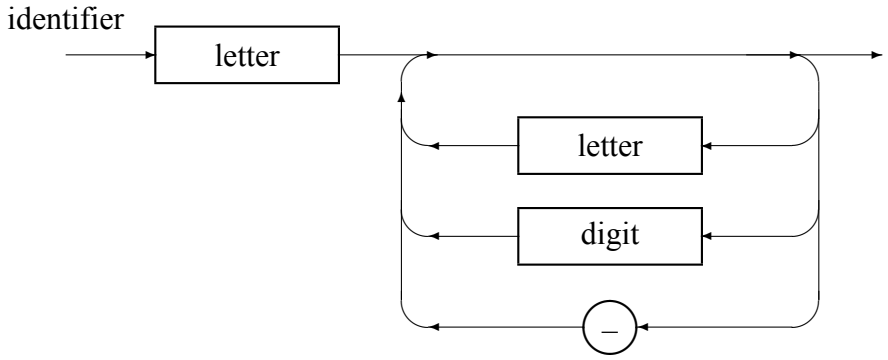
program



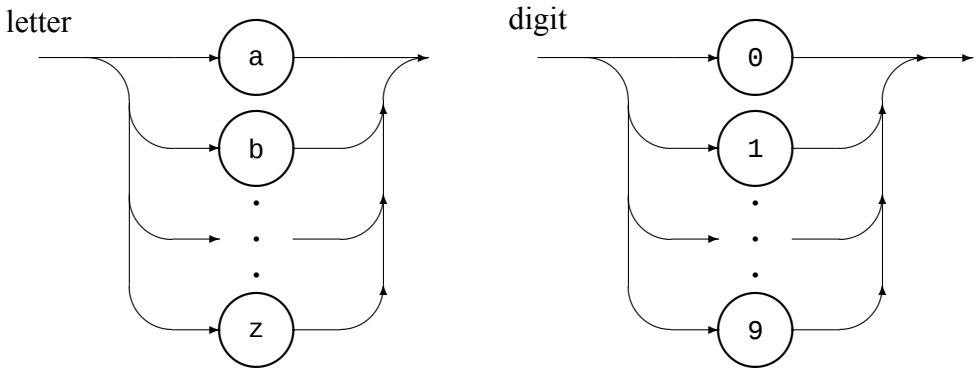
Όπως φαίνεται στο παραπάνω συντακτικό διάγραμμα, ένα πρόγραμμα ξεκινάει με την επικεφαλίδα του λέξη (program header) και ακολουθεί ένα block. Και τα δύο είναι μέσα σε οβάλ κουτιά: τα συντακτικά διαγράμματα που τα περιγράφουν έχουν δοθεί στη σελίδα 13.

Ας δούμε τώρα ένα λίγο πιο πολύπλοκο συντακτικό διάγραμμα, αυτό που αντιστοιχεί στα αναγνωριστικά της Pazcal. Όπως φαίνεται στο σχήμα 2.1, ένα αναγνωριστικό (identifier) πρέπει να ξεκινάει με γράμμα της αλφαβήτου και μπορεί να περιέχει γράμματα, ψηφία και underscore. Στο σχήμα 2.2 δίνονται τα σ.δ. για το **γράμμα** (letter), δηλαδή ένα απο τα γράμματα της αλφαβήτου, και το **ψηφίο** (digit), δηλαδή ένα από τα ψηφία από 0 έως 9. Πρέπει να τονίσουμε εδώ ότι η Pazcal είναι case-sensitive (όπως και η C αλλά αντίθετα από την Pascal), δηλαδή διακρίνει τα κεφαλαία από τα μικρά γράμματα. Έτσι, π.χ. `PROGRAM` και `program` θεωρούνται δύο διαφορετικές λέξεις από τον compiler.

Οι δεσμευμένες λέξεις δεν μπορούν να χρησιμοποιηθούν ως αναγνωριστικά (και αυτό δε φαίνεται στο συντακτικό διάγραμμα). Αναφέρουμε εδώ ενδεικτικά δεσμευμένες λέξεις: `PROGRAM`, `int`, `REAL`, `WRITE`, `while`, `do`, `return`. Ο κατάλογος όλων των δεσμευμένων λέξεων είναι αρκετά μεγαλύτερος (βλέπε παράρτημα).



Σχήμα 2.1. Αναγνωριστικό (identifier).



Σχήμα 2.2. Γράμμα (letter), ψηφίο (digit).

Παραδείγματα ορθών αναγνωριστικών:

```
X
bob
AbF
SKOYLIKOMYRMINGOTRYPA
A98
M6_7L
```

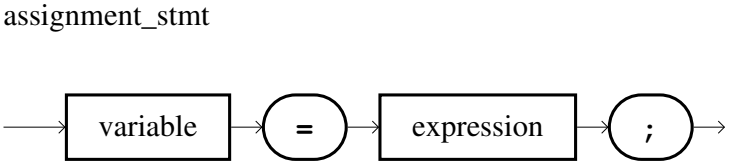
Παραδείγματα λανθασμένων αναγνωριστικών:

```
2ND
M.A,I
E+4
BA(3)
int
```

Ξανατονίζουμε: Στα συντακτικά διαγράμματα ό,τι βρίσκεται μέσα σε κύκλο ή οβάλ είναι σύμβολο της γλώσσας ενώ ό,τι είναι μέσα σε ορθογώνιο είναι βοηθητική έννοια που χρειάζεται περαιτέρω εξήγηση: π.χ. με ένα άλλο συντακτικό διάγραμμα. (Για τα πλήρη συντακτικά διαγράμματα της Pazcal δες το παράρτημα.)

2.3 Ανάθεση τιμής σε μεταβλητή

Οι **μεταβλητές** (variables) είναι ονόματα που δίνονται σε περιοχές της μνήμης του Η/Υ, οι οποίες μπορούν να αποθηκεύουν δεδομένα. Μια μεταβλητή μπορεί να έχει διάφορες τιμές ή και καμιά τιμή (διότι π.χ. δεν της έχουμε **αναθέσει** καμία τιμή ακόμα — στην περίπτωση αυτή θα είναι λάθος να προσπαθήσουμε να χρησιμοποιήσουμε την τιμή της). Για απλές μεταβλητές χρησιμοποιούμε αναγνωριστικά. Υπάρχουν όμως, όπως θα δούμε αργότερα, και πιο πολύπλοκες μεταβλητές. **Αναθέτουμε** ή **εκχωρούμε** (assign) τιμές στις μεταβλητές με την εντολή ανάθεσης (assignment statement), της οποίας το συντακτικό διάγραμμα δίδεται παρακάτω. Οι έννοιες variable και expression θα ορισθούν επακριβώς αργότερα, με συντακτικά διαγράμματα.



Στην παραπάνω σύνταξη, variable είναι η μεταβλητή στην οποία θέλουμε να δώσουμε τιμή, και expression είναι μια (μαθηματική) παράσταση (έκφραση). Η παράσταση expression αποτιμάται και η τιμή εκχωρείται στη μεταβλητή variable. Η παράσταση θα πρέπει να έχει μια τιμή που να είναι συμβατή με τον τύπο της μεταβλητής (να βρίσκεται, δηλαδή, μέσα στο πεδίο τιμών της): δεν μπορούμε π.χ. να εκχωρήσουμε πραγματικές τιμές σε ακέραιες μεταβλητές, ούτε να αναθέσουμε τιμές άλλες από τις **`true`** και **`false`** σε μεταβλητές τύπου **`bool`**.

Αλλά δε πρέπει να ξεχνάμε ότι για να αναθέσουμε σε μεταβλητή τιμή πρέπει πρώτα η μεταβλητή να έχει δηλωθεί. Για αυτό και η δήλωση μεταβλητών στην αρχή του κώδικα, στο τμήμα των δηλώσεων είναι απαραίτητη.

Παραδείγματα εντολών ανάθεσης:

```
x = 2;
pi = 3.1415926;
done = true;
d = 'b';
z1 = (-b + sqrt(b * b - 4 * a * c))/ (2 * a);
counter = counter + 1;
```

Στα πρώτα τέσσερα παραδείγματα, οι εκφράσεις είναι σταθερές παραστάσεις (constants). Στα δύο τελευταία, οι εκφράσεις είναι πιο πολύπλοκες, χρησιμοποιούν τιμές άλλων μεταβλητών, ακόμη και την τιμή αυτής της μεταβλητής στην οποία θα γίνει η ανάθεση. Προϋποτίθενται φυσικά δηλώσεις, π.χ.:

Δηλώσεις (declarations) μεταβλητών:

```
int x, counter;
REAL pi;
bool done;
char d;
REAL z1, a, b, c;
```

Στην εντολή ανάθεσης `z1 = (-b + sqrt(b * b - 4 * a * c))/ (2 * a);` που είδαμε πιο πάνω, η πραγματική μεταβλητή `z1` υπολογίζεται χρησιμοποιώντας τις τιμές των πραγματικών μεταβλητών `a`, `b` και `c`. Η συνάρτηση `sqrt(x)` υπολογίζει την τετραγωνική ρίζα του ορίσματός της `x`, επομένως η τιμή της `z1` είναι πιθανώς μία από τις ρίζες της δευτεροβάθμιας εξίσωσης $\mathtt{a}z^2 + \mathtt{b}z + \mathtt{c} = 0$.

Το πιο αξιοπρόσεκτο όμως από τα παραπάνω παραδείγματα είναι το τελευταίο, στο οποίο η έκφραση στο δεξιό μέλος της ανάθεσης χρησιμοποιεί την ίδια τη μεταβλητή που εκχωρείται στο αριστερό μέλος:

```
counter = counter + 1;
```

Ας υποθέσουμε ότι, πριν εκτελεστεί αυτή η εντολή ανάθεσης, η τιμή της μεταβλητής `counter` είναι 6. Πρώτα υπολογίζεται η έκφραση στο δεξιό μέλος, χρησιμοποιώντας την υπάρχουσα τιμή της μεταβλητής: το αποτέλεσμα είναι $6 + 1 = 7$. Στη συνέχεια, αυτό το αποτέλεσμα εκχωρείται στη μεταβλητή `counter`, η τιμή της οποίας αλλάζει και γίνεται 7 (η παλιά τιμή της προφανώς χάνεται και δεν είναι δυνατό να αποκατασταθεί, παρά μόνο με νέα ανάθεση). Η συμπεριφορά αυτής της εντολής μας λέει δύο πολύ σημαντικά πράγματα:

- Κάθε φορά που εκτελείται, η τιμή της `counter` αυξάνει κατά μία μονάδα, άρα οι εντολές αυτής της μορφής είναι πολύ χρήσιμες όταν θέλουμε να χρησιμοποιήσουμε κάποιο είδος μετρητή στο πρόγραμμά μας.
- Το σύμβολο "=" στην εντολή ανάθεσης δεν πρέπει να το βλέπουμε ως το σύμβολο της ισότητας στα μαθηματικά! Πραγματικά, η ισότητα $\mathtt{counter} = \mathtt{counter} + 1$ στα μαθηματικά είναι αδύνατο να αληθεύει οποιαδήποτε και αν είναι η τιμή της μεταβλητής `counter`. Πολλές γλώσσες προγραμματισμού χρησιμοποιούν διαφορετικό σύμβολο για την ανάθεση ακριβώς για να μη γίνεται αυτή η σύγχυση (π.χ. στην Pascal το σύμβολο της ανάθεσης είναι "`:=`").

Αρχικοποιήσεις μεταβλητών

Κάθε μεταβλητή που χρησιμοποιείται πρέπει να δηλώνεται. Όμως, η τιμή της δεν μπορεί να χρησιμοποιηθεί αν δεν προηγηθεί κάποια ανάθεση σε αυτή τη μεταβλητή (δηλαδή η μεταβλητή δεν μπορεί να εμφανίζεται, π.χ., στο δεξιό μέλος κάποιας ανάθεσης αν προηγουμένως δεν έχει εμφανιστεί στο αριστερό μέλος κάποιας άλλης ανάθεσης). Η πρώτη φορά που ανατίθεται τιμή σε μία μεταβλητή λέγεται συχνά **αρχικοποίηση** (initialization) της μεταβλητής.

Στην Pazcal, η αρχικοποίηση μίας μεταβλητής είναι δυνατό να γίνει συγχρόνως με τη δήλωσή της. Για παράδειγμα στο τμήμα τοπικών δηλώσεων ενός block μπορούμε να γράψουμε:

```
int x, counter = 6, z;
REAL pi = 3.1415926;
bool done = true;
```

με αποτέλεσμα όλες οι μεταβλητές εκτός από τις `x` και `z` να αρχικοποιηθούν στις τιμές που δίνονται κατά τη δήλωσή τους. Προφανώς, ο τύπος των εκφράσεων που αρχικοποιούν τις μεταβλητές πρέπει να συμφωνεί με τον τύπο των μεταβλητών, όπως στις αναθέσεις. Οι τιμές των

μεταβλητών x και z, που δεν αρχικοποιούνται, εξακολουθούν να μην μπορούν να χρησιμοποιηθούν πριν γίνει ανάθεση σε αυτές.

Αναπαράσταση δεδομένων και εύρος τιμών

Το εύρος τιμών που μπορεί να αποθηκευτεί για κάθε τύπο δεδομένων στην Pazcal εξαρτάται γενικά από το είδος υπολογιστή που χρησιμοποιούμε. Για παράδειγμα, για ακέραιους αριθμούς, οι τιμές του τύπου **`int`** είναι συνήθως στους σημερινούς υπολογιστές από -2147483648 (`INT_MIN` $= -2^{31}$) μέχρι 2147483647 (`INT_MAX` $= 2^{31} - 1$) και απαιτούν 32 bits (διφία) στη μνήμη. Βέβαια εκφράσαμε το εύρος των πιθανών τιμών των μεταβλητών με δεκαδικές τιμές, αλλά οι ακέραιοι αποθηκεύονται εσωτερικά (internally) ως δυαδικοί αριθμοί και το πλήθος των διαθέσιμων bits για την αποθήκευση κάθε τύπου (εν προκειμένω 32) είναι εκείνο που καθορίζει τις μέγιστες και τις ελάχιστες επιτρεπόμενες τιμές.

```
INT_MAX: 0111 ... 1
INT_MIN: 1000 ... 0
```

Η αναπαράσταση των ακέραιων αριθμών στη μνήμη του υπολογιστή γίνεται συνήθως σε "συμπλήρωμα ως προς 2". Το αριστερότερο (most significant) bit χρησιμοποιείται για το πρόσημο: όταν είναι `1` ο αριθμός είναι αρνητικός ενώ διαφορετικά είναι θετικός ή μηδέν. Περισσότερα στο μάθημα "Λογική Σχεδίαση" του 2ου εξαμήνου. Η αναπαράσταση των πραγματικών αριθμών (κινητής υποδιαστολής — floating point) είναι αρκετά πιο πολύπλοκη. Θα αναφερθούμε σε αυτήν στο κεφάλαιο 6.

2.4 Επικοινωνία με το χρήστη: είσοδος και έξοδος

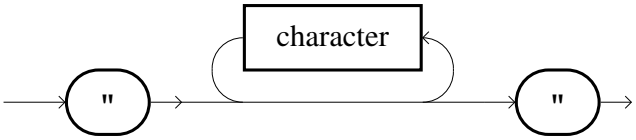
Για να είναι χρήσιμο ένα πρόγραμμα, που εκτελείται σε κάποιον υπολογιστή, πρέπει να αλληλεπιδρά με το περιβάλλον. Αυτό συνήθως γίνεται μέσω ειδικών συσκευών που ονομάζονται **συσκευές εισόδου/εξόδου** (input/output devices — I/O devices). Η οθόνη του υπολογιστή είναι μια συσκευή εξόδου και το πληκτρολόγιο μια συσκευή εισόδου. Τα περισσότερα προγράμματα αλληλεπιδρούν με το περιβάλλον διαβάζοντας δεδομένα από κάποια συσκευή εισόδου και παρουσιάζοντας τα αποτελέσματα της εκτέλεσής τους σε κάποια συσκευή εξόδου.

2.4.1 Έξοδος στην οθόνη

Σ' αυτό το σημείο θα εξηγήσουμε τις εντολές εξόδου **`WRITE`** και **`WRITELN`** που συναντήσαμε στο πρώτο παράδειγμα αυτού του κεφαλαίου, καθώς επίσης και των εντολών **`WRITESP`** και **`WRITESPLN`**. Πριν από όλα όμως πρέπει να αναλύσουμε την έννοια της συμβολοσειράς (string).

Με αυστηρό ορισμό οι συμβολοσειρές (strings) είναι πεπερασμένες ακολουθίες από χαρακτήρες του αλφαβήτου της γλώσσας Pazcal. Το αλφάβητο της γλώσσας θα το δούμε αργότερα. Το μόνο που χρειαζόμαστε σε αυτό το σημείο είναι ότι τα strings μπορούν να περιέχουν γράμματα, ψηφία, σύμβολα, σημεία στίξης και κενούς χαρακτήρες. Οι σταθερές συμβολοσειρές περικλείονται πάντοτε σε εισαγωγικά (quotes). Τα παραπάνω εξηγούνται πολύ απλά με το παρακάτω απλοποιημένο συντακτικό διάγραμμα:

string_constant



Μερικά παραδείγματα strings είναι τα εξής:

```
"Hello world!"
"Οι αριθμοί που μας ενδιαφέρουν είναι το 17 και το 42."
""
"  ()()()  :-)  ()()()  "
```

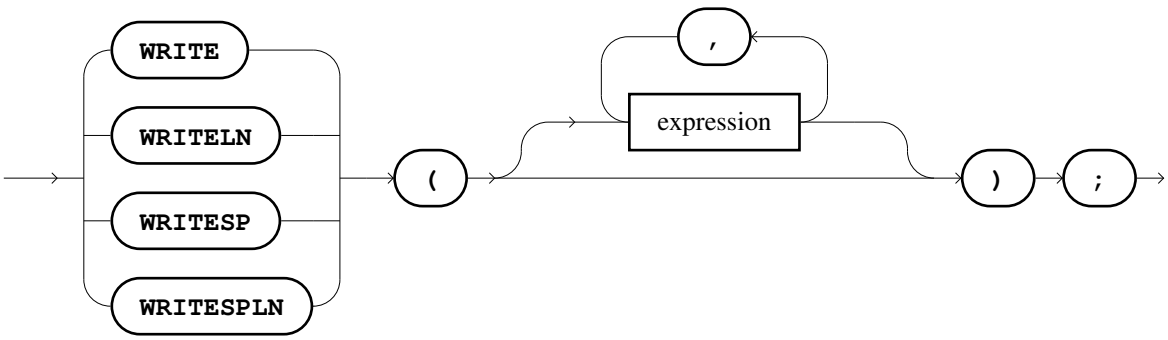
Ο αναγνώστης ας διατηρήσει στη μνήμη του ότι (όπως στο δεύτερο παράδειγμα) το string μπορεί να περιέχει και ψηφία ή (όπως στο τρίτο παράδειγμα) μπορεί να είναι κενό.

Στις προηγούμενες παραγράφους συναντήσαμε την έννοια της έκφρασης (expression) και είδαμε ότι αναφέρεται σε αριθμητικές ή λογικές παράστασεις που αποτελούνται από αριθμητικές ή λογικές σταθερές, μεταβλητές και μαθηματικούς τελεστες. Η τιμή μιας αριθμητικής παράστασης είναι ένας αριθμός ενώ μιας λογικής παράστασης είναι **`true`** ή **`false`**. Πριν προχωρήσουμε, θα προσθέσουμε ένα ακόμη είδος έκφρασης: τις συμβολοσειρές (strings). Οι τιμές τους είναι σταθερές συμβολοσειρές, όπως αυτές που είδαμε παραπάνω. Περισσότερα για τις συμβολοσειρές (π.χ. τι τύπου είναι, πώς κατασκευάζω μεταβλητές συμβολοσειρές, κ.λπ.) θα δούμε αρκετά αργότερα, για την ώρα όμως είναι σημαντικό να τις συγκαταλέγουμε στις εκφράσεις για να μπορούμε να εμφανίζουμε μηνύματα στην οθόνη.

Οι εντολές `WRITE` και `WRITELN`.

Κατά την εκτέλεση των περισσότερων προγράμματων, τα αποτελέσματα παρουσιάζονται στο χρήστη μέσω της οθόνης του υπολογιστή. Για το σκοπό αυτό χρησιμοποιούνται οι εντολές **`WRITE`**, **`WRITELN`**, **`WRITESP`** και **`WRITESPLN`**. Η σύνταξή τους περιγράφεται στο παρακάτω συντακτικό διάγραμμα:

write_stmt



Η χρήση όλων των εντολών της οικογένειας **`WRITE`** γίνεται γράφοντας μέσα σε παρενθέσεις τις εκφράσεις οι τιμές των οποίων που είναι επιθυμητό να παρουσιαστούν στην οθόνη και χωρίζοντας αυτές μεταξύ τους με κόμματα. Κατά την εκτέλεση υπολογίζονται πρώτα οι τιμές των

εκφράσεων και στη συνέχεια γίνεται η παρουσίαση. Αν η τιμή μίας έκφρασης είναι συμβολοσειρά, οι χαρακτήρες της εκτυπώνονται ακριβώς όπως είναι. Αν η τιμή είναι αριθμητική, τότε εκτυπώνεται σε δεκαδική αναπαράσταση (όπως δηλαδή έχουμε συνηθίσει να βλέπουμε τους αριθμούς). Αν η τιμή είναι λογική, τότε εκτυπώνεται "`true`" ή "`false`".

Για να γίνει κατανοητός ο τρόπος που γίνεται η παρουσίαση, ας φανταστούμε ότι υπάρχει στην οθόνη του υπολογιστή μια υποθετική γραφίδα, μέσω της οποίας γίνεται η παρουσίαση των αποτελεσμάτων. Η γραφίδα αυτή ονομάζεται και δρομέας ή δείκτης (cursor) και, κατά τη διάρκεια της εκτέλεσης του προγράμματος, συχνά εμφανίζεται ως ένα τετραγωνάκι που αναβοσβήνει. Η γραφίδα μπορεί σε κάθε σημείο της οθόνης να γράψει ένα χαρακτήρα και, κάθε φορά που γράφει ένα χαρακτήρα, προχωρά μια θέση δεξιότερα στην οθόνη. Αρχικά βρίσκεται στην πάνω αριστερή γωνία της οθόνης. Για κάθε μια έκφραση που δίνεται σε μια εντολή της οικογένειας **`WRITE`**, αφού πρώτα υπολογιστεί η τιμή της έκφρασης, η γραφίδα γράφει έναν-έναν τους χαρακτήρες που αντιστοιχούν σε αυτή την τιμή. Για παράδειγμα, αν η τιμή της έκφρασης είναι η συμβολοσειρά `"Hello world!"`, τότε η γραφίδα θα γράψει πρώτα το χαρακτήρα `H`, θα προχωρήσει μια θέση δεξιότερα, θα γράψει το χαρακτήρα `e`, θα προχωρήσει άλλη μια θέση δεξιότερα, κ.ο.κ. για όλους τους χαρακτήρες συμπεριλαμβανομένου και του κενού διαστήματος. Μόλις γράψει τον τελευταίο χαρακτήρα `!` (θαυμαστικό) θα προχωρήσει άλλη μία θέση δεξιότερα και θα σταματήσει.

Η μόνη διαφορά ανάμεσα στις εντολές **`WRITE`** και **`WRITELN`** είναι ότι η **`WRITELN`**, αφού παρουσιάσει τις τιμές όλων των εκφράσεων που βρίσκονται ανάμεσα στις παρενθέσεις, τοποθετεί τη γραφίδα στην αρχή της επόμενης γραμμής στην οθόνη. Αντίθετα, η **`WRITE`** αφήνει τη γραφίδα στο σημείο όπου έχει μεταφερθεί, μετά την παρουσίαση της τελευταίας τιμής.

Ας δούμε μερικά απλά παραδείγματα και, δεξιά, τι εμφανίζεται στην οθόνη.

```
• WRITELN("Hello world!");                          Hello world!
  WRITELN("Hell", "o wor", "ld!");                  Hello world!

• WRITE("Hel");                                     Hello world!
  WRITELN("lo w", "orld!");

• x = 6;                                            x = 6
  WRITE("x");                                       3*x-1 = 17
  WRITE(" = ");                                     x*(x+1) = 42
  WRITE(x);
  WRITELN();
  WRITELN("3*x-1 = ", 3*x-1);
  WRITELN("x*(x+1) = ", x*(x+1));
```

Προσέξτε ότι η εντολή **`WRITE`**`("x");` εμφανίζει στην οθόνη το χαρακτήρα "`x`" (γιατί εμφανίζονται τα περιεχόμενα της συμβολοσειράς `"x"` ακριβώς όπως είναι). Αντίθετα, η εντολή **`WRITE`**`(x);` υπολογίζει την τιμή της αριθμητικής έκφρασης `x` (δηλαδή βρίσκει από την αντίστοιχη θέση μνήμης την τιμή που έχει η μεταβλητή `x`) και εμφανίζει στην οθόνη το αποτέλεσμα: 6. Παρομοίως, προσέξτε τι εμφανίζουν στην οθόνη οι δύο τελευταίες γραμμές του παραδείγματος: πρώτα μία συμβολοσειρά που εμφανίζεται ως έχει και στη συνέχεια μία αριθμητική έκφραση, που υπολογίζεται και εμφανίζεται το αποτέλεσμά της.

Τέλος, πολλές φορές είναι επιθυμητό να εμφανιστούν στην οθόνη περισσότερες από μία τιμές αλλά να μην "κολλήσουν" μεταξύ τους. Οι εντολές **WRITESP** και **WRITESPLN** κάνουν ό,τι και οι **WRITE** και **WRITELN** αλλά, μεταξύ διαδοχικών τιμών που εμφανίζουν, προχωράνε τη γραφίδα μία θέση δεξιότερα. Κατ' αυτόν τον τρόπο, οι τιμές που εμφανίζονται διαχωρίζονται ανά δύο με ένα κενό διάστημα. Ας δούμε ένα ακόμα παράδειγμα και, δεξιά, τι εμφανίζεται στην οθόνη.

```
WRITELN(4, 2);                    42
WRITESPLN(4, 2);                  4 2
WRITE(6, 6);                      666
WRITELN(6);                       6 66
WRITESP(6, 6);
WRITESPLN(6);
```

Προσέξτε ότι οι **WRITE** και **WRITELN** ποτέ δε χωρίζουν τις τιμές που εμφανίζουν με κενά διαστήματα. Αντίθετα, οι **WRITESP** και **WRITESPLN** τις χωρίζουν, όταν είναι περισσότερες από μία. Μετά την τελευταία όμως τιμή σε μία **WRITESP** δεν προστίθεται κενό διάστημα, γι' αυτό και προκύπτει το `66` στο παραπάνω πρόγραμμα.

2.4.2 Μορφοποίηση εξόδου

Πολλές φορές θέλουμε η έξοδος ενός προγράμματος να έχει συγκεκριμένη μορφή. Για παράδειγμα, είναι πιθανό να θέλουμε οι αριθμοί που υπολογίζει το πρόγραμμά μας να εμφανίζονται με τέσσερα δεκαδικά ψηφία και σωστά στοιχισμένοι. Αυτό στην Pazcal μπορούμε να το πετύχουμε συνδυάζοντας την οικογένεια των εντολών **WRITE**, που είδαμε προηγουμένως, με τη λέξη-κλειδί `FORM`.

Στην απλούστερη μορφή της, η `FORM` εμφανίζει το πρώτο της όρισμα στη δεξιά άκρη ενός "κουτιού" που έχει πλάτος τόσους χαρακτήρες όσο είναι η τιμή του δεύτερου ορίσματος.

```
WRITELN("x=", FORM(42, 8));       x=␣␣␣␣42
WRITELN("y=", FORM(-324, 8));     y=␣␣-324
WRITELN("z=", FORM(17, 8));       z=␣␣␣␣17
WRITELN("w=", FORM(0, 8));        w=␣␣␣␣␣0
```

Στο παραπάνω πρόγραμμα, οι αριθμοί $42, -324, 17$ και 0 εμφανίζονται δεξιά στοιχισμένοι με πλάτος 8 χαρακτήρες. Στα αριστερά κάθε αριθμού συμπληρώνεται ο κατάλληλος αριθμός κενών διαστημάτων. Το ίδιο μπορεί να γίνει και για τη μορφοποίηση δεδομένων άλλου τύπου:

```
WRITELN(FORM("hello", 8));        ␣␣␣hello
WRITELN(FORM("what", 8));         ␣␣␣␣what
WRITELN(FORM("do", 8));           ␣␣␣␣␣␣do
WRITELN(FORM("you", 8));          ␣␣␣␣␣you
WRITELN(FORM("think?", 8));       ␣␣think?
```

Μία δεύτερη μορφή του `FORM` μας επιτρέπει να προσδιορίσουμε πόσα δεκαδικά ψηφία ενός πραγματικού αριθμού θέλουμε να εμφανιστούν.

```
WRITELN("Temperatures");                                  Temperatures
WRITELN("------------");                                  ---------------
WRITELN(FORM("Athens", 10), FORM(100.0 / 3, 6, 1));       ␣␣␣␣Athens␣␣33.3
WRITELN(FORM("Rome", 10), FORM(100.0 / 4, 6, 1));         ␣␣␣␣␣␣Rome␣␣25.0
WRITELN(FORM("Paris", 10), FORM(100.0 / 6, 6, 1));        ␣␣␣␣␣Paris␣␣16.7
WRITELN(FORM("London", 10), FORM(100.0 / 15, 6, 1));      ␣␣␣␣London␣␣␣6.7
WRITELN(FORM("Moscow", 10), FORM(100.0 / -24, 6, 1));     ␣␣␣␣Moscow␣␣-4.2
```

2.4.3 Είσοδος από το πληκτρολόγιο

Στα περισσότερα προγράμματα χρειάζεται ο χρήστης να εισάγει δεδομένα. Στο παράδειγμα της αρχής του κεφαλαίου, αυτό έγινε με τη συνάρτηση READ_REAL (θα μιλήσουμε εκτενώς για τις συναρτήσεις και τις διαδικασίες της Pazcal αργότερα). Η συνάρτηση αυτή μπορεί να χρησιμοποιηθεί σε οποιαδήποτε έκφραση και έχει ως αποτέλεσμα έναν πραγματικό αριθμό. Τον αριθμό αυτό τον ζητάει από το χρήστη, ο οποίος θα πρέπει να τον πληκτρολογήσει, σε δεκαδική μορφή. Ας δούμε τι συμβαίνει όταν εκτελείται το παράδειγμα της αρχής του κεφαλαίου. Αριστερά είναι ο κώδικας και δεξιά ό,τι βλέπει ο χρήστης στην οθόνη. Υπογραμμισμένα είναι αυτά που ο χρήστης πληκτρολογεί και το σύμβολο " ↵ " σημαίνει ότι ο χρήστης πατάει το πλήκτρο "Enter" (ή "Return").

```
PROGRAM example1()                                Give the radius: 10↵
{                                                 The area is: 314.159260
  REAL r, a;

  WRITE("Give the radius: ");
  r = READ_REAL();
  a = 3.1415926 * r * r;
  WRITELN("The area is: ", a);
}
```

Αρχικά, το πρόγραμμα εμφανίζει στη οθόνη το μήνυμα "`Give the radius: `" και στη συνέχεια περιμένει απόκριση από το χρήστη. Η αναμονή αυτή οφείλεται στη συνάρτηση READ_REAL. Η γραφίδα (cursor) αναβοσβήνει. Η συνάρτηση READ_REAL "περιμένει" τον χρήστη να εισαγάγει από το πληκτρολόγιο έναν πραγματικό αριθμό και μετά να πατήσει το πλήκτρο "Enter". Μόλις συμβεί αυτό, η συνάρτηση επιστρέφει ως αποτέλεσμα τον αριθμό που πληκτρολόγησε ο χρήστης, ο οποίος ανατίθεται στη μεταβλητή `r`. Στη συνέχεια γίνεται ο υπολογισμός του εμβαδού και η εμφάνιση της δεύτερης γραμμής της εξόδου.

Απροσδόκητη είσοδος — σφάλμα εκτέλεσης

Αν ο χρήστης αντί να πληκτρολογήσει τον πραγματικό αριθμό που έχει ζητηθεί πληκτρολογήσει κάτι άλλο, π.χ. γράμματα αντί ψηφίων, τότε θα συμβεί ένα **σφάλμα εκτέλεσης** (runtime error). Ο υπολογιστής θα εμφανίσει ένα κατάλληλο μήνυμα σφάλματος και θα σταματήσει την εκτέλεση του προγράμματος. Αν εξαιρέσουμε ότι το πρόγραμμα δεν τρέχει πια, τίποτα τραγικό δεν έχει συμβεί... Ο χρήστης μπορεί να το εκτελέσει ξανά και να εισαγάγει αυτή τη φορά έναν έγκυρο πραγματικό αριθμό.

Αν όμως ο χρήστης πληκτρολογήσει πρώτα μερικά κενά διαστήματα και στη συνέχεια δώσει τον αριθμού που του ζητείται, τότε δε συμβαίνει σφάλμα εκτέλεσης: η READ_REAL προσπερνάει

τα κενά διαστήματα στην αρχή και προχωράει στην ανάγνωση του αριθμού. Το ίδιο συμβαίνει και αν ο χρήστης πατήσει μερικές φορές το πλήκτρο "Enter" πριν δώσει τον αριθμό: και σε αυτήν την περίπτωση, η READ_REAL περιμένει υπομονετικά. Σφάλμα εκτέλεσης συμβαίνει όταν ο πρώτος χαρακτήρας μετά από οσαδήποτε κενά διαστήματα ή αλλαγές γραμμής (δηλαδή πατήματα του πλήκτρου "Enter") δεν ξεκινά κάποιον έγκυρο πραγματικό αριθμό.

Είσοδος ακέραιων αριθμών και χαρακτήρων

Κατ' αναλογία με τη συνάρτηση READ_REAL, οι συναρτήσεις READ_INT και getchar διαβάζουν ακέραιους αριθμούς και χαρακτήρες, αντίστοιχα. Για παράδειγμα, το παρακάτω πρόγραμμα διαβάζει δύο ακέραιους μεταξύ των οποίων παρεμβάλλεται ένας χαρακτήρας (το σύμβολο μιας πράξης) και υπολογίζει το αποτέλεσμα της πράξης αυτής. Η εντολή **switch** χρησιμοποιείται για να διακρίνει περιπτώσεις ανάλογα με το σύμβολο της πράξης που έχει εισαχθεί: θα παρουσιαστεί αναλυτικά στο επόμενο κεφάλαιο — για την ώρα, επικεντρωθείτε στην είσοδο των δεδομένων. Δεξιά φαίνονται τρεις διαδοχικές εκτελέσεις του προγράμματος:

```
PROGRAM operation()
{
  int first, second, result;
  char operator;

  first = READ_INT();
  operator = getchar();
  second = READ_INT();

  switch (operator) {
    case '+': result = first + second; break;
    case '-': result = first - second; break;
    case '*': result = first * second; break;
    case '/': result = first / second; break;
  }
  WRITELN("The result is: ", result);
}
```

```
8+9↵
The result is: 17

102-201↵
The result is: -99

6*7↵
The result is: 42
```

Ας εξετάσουμε το δεύτερο παράδειγμα. Ο χρήστης πληκτρολόγησε "102-201" και πάτησε το πλήκτρο "Enter". Η εντολή `first = READ_INT();` διάβασε έναν ακέραιο αριθμό (τον 102) και τον ανέθεσε στη μεταβλητή first. Στη συνέχεια, η εντολή `operator = getchar();` διάβασε ένα χαρακτήρα (τον "-") και τον ανέθεσε στη μεταβλητή operator. Τέλος, η εντολή `second = READ_INT();` διάβασε έναν ακέραιο αριθμό (τον 201) και τον ανέθεσε στη μεταβλητή second. Προσέξτε ότι ο χρήστης δε χρειάστηκε να πατήσει τρεις φορές το πλήκτρο "Enter", μία για κάθε συνάρτηση εισόδου που κλήθηκε. Αυτό θα εξηγηθεί αμέσως τώρα.

Ενδιάμεσος αποθηκευτικός χώρος — buffer

Πολλές φορές ένα πρόγραμμα που διαβάζει δεδομένα από το πληκτρολόγιο μάς φαίνεται ότι συμπεριφέρεται απρόσμενα. Για παράδειγμα, ενώ περιμένουμε ότι η εκτέλεση θα σταματήσει και το πρόγραμμα θα περιμένει το χρήστη να πληκτρολογήσει κάποια δεδομένα, αυτό δε συμβαίνει. Ή και το αντίθετο, η εκτέλεση σταματά και περιμένει νέα δεδομένα ενώ νομίζουμε ότι αυτό δε χρειάζεται. Ή, ακόμα χειρότερα, συμβαίνει κάποιο σφάλμα εκτέλεσης γιατί τα δεδομένα που διαβάζονται δεν είναι αυτά που θα περιμέναμε. Για να εξηγηθούν τέτοιου είδους

απρόσμενες συμπεριφορές πρέπει να γίνει κατανοητός ο τρόπος με τον οποίο εισάγονται στην πραγματικότητα τα δεδομένα από το πληκτρολόγιο στον υπολογιστή.

Μεταξύ του πληκτρολογίου και του προγράμματός μας, παρεμβάλλεται μία καταχώρηση σε έναν ενδιάμεσο (προσωρινό) αποθηκευτικό χώρο, που λέγεται buffer. Στην αρχή της εκτέλεσης του προγράμματος, ο buffer είναι άδειος. Όταν ο χρήστης πληκτρολογεί κάτι, αυτό αποθηκεύεται αυτόματα στο buffer, και όταν το πρόγραμμα χρειάζεται να διαβάσει δεδομένα από το πληκτρολόγιο, τα ζητά ουσιαστικά από το buffer. Η αποθήκευση των δεδομένων στον buffer γίνεται στην πραγματικότητα όταν ο χρήστης πατήσει το πλήκτρο "Enter" και όχι με κάθε χαρακτήρα που πληκτρολογεί (αυτό είναι βολικό για να μπορεί ο χρήστης να διορθώνει τυχόν λάθη που κάνει στα δεδομένα εισόδου, π.χ. σβήνοντας με το πλήκτρο "Backspace" ή "Delete".

Όταν σε ένα πρόγραμμα εκτελείται π.χ. η συνάρτηση `READ_INT` (παρόμοια συμπεριφορά έχει και η `READ_REAL`), το πρόγραμμα ζητά από το buffer δεδομένα για να τα διαβάσει, ερμηνεύοντας αυτά ως έναν ακέραιο αριθμό στο δεκαδικό σύστημα. Αν ο buffer είναι άδειος, το πρόγραμμα "παγώνει" και περιμένει από το χρήστη να πληκτρολογήσει δεδομένα. Μόλις ο χρήστης το κάνει και πατήσει το πλήκτρο "Enter", τα δεδομένα που πληκτρολόγησε αποθηκεύονται στον buffer (συμπεριλαμβανομένου και του "Enter", που είναι ο χαρακτήρας "end of line" ή "τέλος γραμμής" και συμβολίζεται με `'\n'` στην Pazcal— περισσότερα για τους ειδικούς χαρακτήρες σε επόμενο κεφάλαιο). Η συνάρτηση `READ_INT` προχωράει αμέσως μόλις στον buffer υπάρχουν δεδομένα προς ανάγνωση.

Στη συνέχεια, η `READ_INT` εξετάζει έναν-έναν τους χαρακτήρες που βρίσκει στον buffer. Αν είναι κενά διαστήματα ή χαρακτήρες αλλαγής γραμμής (εν γένει "λευκοί χαρακτήρες" — white space), τους προσπερνάει και συνεχίζει με τα υπόλοιπα περιεχόμενα του buffer, αν υπάρχουν. Αν στη διαδικασία αυτή ο buffer αδειάσει, τότε και πάλι το πρόγραμμα "παγώνει" έως ότου ο χρήστης πληκτρολογήσει νέα δεδομένα και ο buffer ξαναγεμίσει. Μόλις η `READ_INT` βρει ένα χαρακτήρα που δεν είναι "λευκός", αρχίζει την ανάγνωση του ακέραιου αριθμού. Αν ο χαρακτήρας που διαβάστηκε είναι ψηφίο, συνεχίζει να διαβάζει χαρακτήρες μέχρι να βρει κάτι που δεν είναι ψηφίο. Μόλις συμβεί αυτό, η `READ_INT` ερμηνεύει τα ψηφία που διάβασε ως έναν ακέραιο αριθμό στο δεκαδικό σύστημα, τον οποίο και επιστρέφει. Αντίθετα, αν ο πρώτος μη "λευκός" χαρακτήρας δεν είναι ψηφίο, τότε όπως έχουμε πει προκύπτει σφάλμα εκτέλεσης.

Από τα παραπάνω καταλαβαίνουμε ότι μετά την εκτέλεση της `READ_INT` ο buffer δεν είναι αναγκαστικά άδειος. (Για την ακρίβεια, μετά την εκτέλεσή της ο buffer δεν είναι ποτέ άδειος. Μόλις διαβαστούν τα ψηφία του αριθμού, ο χαρακτήρας αλλαγής γραμμής που βρίσκεται στο τέλος του buffer — αυτός που τοποθετήθηκε εκεί όταν ο χρήστης πάτησε "Enter" — και ίσως και άλλοι χαρακτήρες που προηγούνται αυτού δεν πρόκειται να διαβαστούν από τη `READ_INT`.) Τα δεδομένα που παραμένουν στον buffer είναι διαθέσιμα και μπορούν να χρησιμοποιηθούν σε επόμενες εκτελέσεις των συναρτήσεων εισόδου.

Αυτό ακριβώς έγινε στο πρόγραμμα `operation` που είδαμε πριν λίγο: στην πρώτη `READ_INT` ζητείται από το χρήστη να πληκτρολογήσει δεδομένα. Ο χρήστης πληκτρολογεί "`102-201`↵", όμως η πρώτη `READ_INT` διαβάζει μόνο το "`102`" και το μετατρέπει στον ακέραιο αριθμό 102 ο οποίος τελικά εκχωρείται στη μεταβλητή `first`. Μετά το τέλος της πρώτης `READ_INT`, τα δεδομένα που έχουν παραμείνει στον buffer είναι "`-201`↵". Στη συνέχεια, η συνάρτηση `getchar` διαβάζει ένα χαρακτήρα, τον πρώτο που βρίσκει στον buffer, δηλαδή το σύμβολο "`-`" (προσοχή: η `getchar` δεν προσπερνά τους "λευκούς" χαρακτήρες, όπως οι `READ_INT` και `READ_REAL`). Τώρα ο buffer περιέχει "`201`↵". Η δεύτερη `READ_INT` διαβάζει το "`201`" και το μετατρέπει στον ακέραιο αριθμό 201 ο οποίος εκχωρείται στη μεταβλητή `second`. Στο τέλος, στον buffer έχει μείνει μόνο ο χαρακτήρας αλλαγής γραμμής "↵". Προσέξτε ότι το πρόγραμμα

"πάγωσε" μόνο μία φορά: η `getchar` και η δεύτερη `READ_INT` βρήκαν δεδομένα στον buffer και προχώρησαν στην ανάγνωσή τους χωρίς να διακόψουν την εκτέλεση του προγράμματος.

Καθάρισμα του buffer

Ας δούμε τώρα ένα πρόγραμμα λίγο απλούστερο από το `operation`, που ζητά από το χρήστη δύο αριθμούς και τους αθροίζει. Μία πρώτη προσέγγιση είναι αυτή και, δεξιά, δύο σενάρια εκτέλεσής του:

```
PROGRAM addition1()                                    First: 8↵
{                                                      Second: 9↵
  int first, second;                                   Result: 17
  WRITE("First: ");   first = READ_INT();
  WRITE("Second: ");  second = READ_INT();             First: 222 444↵
  WRITELN("Result: ", first + second);                 Second: Result: 666
}
```

Το πρώτο σενάριο εκτέλεσης φαίνεται φυσιολογικό, αλλά τι συνέβη στο δεύτερο; Αν έχετε καταλάβει πώς λειτουργεί ο buffer και η `READ_INT` δε θα σας είναι δύσκολο να δείτε τι έγινε. Ο χρήστης, αν και είδε το μήνυμα "`First:`" δεν περιορίστηκε στο να πληκτρολογήσει μόνο τον πρώτο αριθμό: μετά από αυτόν έβαλε ένα κενό διάστημα και έγραψε και έναν δεύτερο αριθμό. Αυτά έμειναν στον buffer και η δεύτερη `READ_INT` δε χρειάστηκε να παγώσει το πρόγραμμα και να ζητήσει νέα δεδομένα. Άρα, αμέσως μετά το δεύτερο μήνυμα "`Second:`" εμφανίστηκε το αποτέλεσμα της πράξης.

Ο επιμελής αναγνώστης ίσως προβληματιστεί σε αυτό το σημείο από το γεγονός ότι υπάρχει ένα μόνο **`WRITELN`** στο πρόγραμμα, αντίθετα όμως εμφανίζονται τρεις γραμμές στο πρώτο σενάριο εκτέλεσης και δύο στο δεύτερο. Η γραφίδα (cursor) μετακινείται στην επόμενη γραμμή της οθόνης όχι μόνο όταν εκτελεστεί μία **`WRITELN`** αλλά και όταν ο χρήστης πατήσει το πλήκτρο "Enter". Αυτό εξηγεί γιατί στο πρώτο σενάριο εκτέλεσης τα "`First:`" και "`Second:`" εμφανίζονται σε μία γραμμή το καθένα, μαζί με τον αριθμό που πληκτρολογεί ο χρήστης, και γιατί στο δεύτερο σενάριο εκτέλεσης το "`Second:`" και το "`Result:`" εμφανίζονται στην ίδια γραμμή.

Μπορούμε να κάνουμε κάτι σε αυτό το πρόγραμμα, ώστε ο χρήστης να υποχρεώνεται να δώσει ένα μόνο αριθμό κάθε φορά; Γενικά δεν μπορούμε να υποχρεώσουμε το χρήστη να έχει τη συμπεριφορά που θέλουμε! Αυτό που μπορούμε όμως να κάνουμε είναι, αν είμαστε σίγουροι ότι ο buffer έχει δεδομένα που δεν τα χρειαζόμαστε, να τα καθαρίσουμε. Η διαδικασία `SKIP_LINE` κάνει ακριβώς αυτό:

```
PROGRAM addition2()                                    First: 8↵
{                                                      Second: 9↵
  int first, second;                                   Result: 17
  WRITE("First: ");   first = READ_INT();
  SKIP_LINE();                                         First: 222 444↵
  WRITE("Second: ");  second = READ_INT();             Second: 111↵
  WRITELN("Result: ", first + second);                 Result: 333
}
```

Βάζοντας την εντολή `SKIP_LINE();` μεταξύ των δύο `READ_INT`, τα δεδομένα που έχουν παραμείνει στο buffer μετά την πρώτη `READ_INT` καθαρίζουν και ο buffer αδειάζει και πάλι. Στο πρώτο σενάριο εκτέλεσης, αυτό δεν έχει καμία ουσιαστική σημασία (τα περιεχόμενα του

buffer που καθαρίζουν είναι “ ↵ ”). Στο δεύτερο όμως, αυτό έχει ως αποτέλεσμα να αγνοηθούν τα παραπανίσια δεδομένα “ `444` ↵ ” και η δεύτερη `READ_INT` να ζητήσει εκ νέου έναν ακέραιο αριθμό από το χρήστη. Το αποτέλεσμα που προκύπτει είναι $222 + 111 = 333$.

Προσέξτε ότι στο σημείο που χρησιμοποιήσαμε τη `SKIP_LINE` είμαστε βέβαιοι ότι ο buffer δεν είναι άδειος. Αν την καλέσουμε και ο buffer είναι άδειος, τότε το πρόγραμμα θα “παγώσει” και η `SKIP_LINE` θα περιμένει να πληκτρολογήσει ο χρήστης δεδομένα για να τα αγνοήσει όλα! Δυστυχώς δεν υπάρχει τρόπος να ελέγξουμε αν ο buffer είναι ή όχι άδειος.

2.4.4 Ανακατεύθυνση εισόδου και εξόδου

Υπό φυσικολογικές συνθήκες, η οικογένεια των εντολών εξόδου που είδαμε εμφανίζουν τα αποτελέσματα στην οθόνη και εκείνη των συναρτήσεων εισόδου διαβάζουν τα δεδομένα από το πληκτρολόγιο. Για να είμαστε πιο ακριβείς, οι εντολές εξόδου εμφανίζουν τα αποτελέσματα στην **τυπική συσκευή εξόδου** (standard output device — `stdout`) και οι συναρτήσεις εισόδου διαβάζουν τα δεδομένα από την **τυπική συσκευή εισόδου** (standard input device — `stdin`), όμως, υπό φυσιολογικές συνθήκες, οι δύο αυτές συσκευές αντιστοιχούν στην οθόνη και το πληκτρολόγιο.

Ο προγραμματιστής είναι δυνατόν να **ανακατευθύνει** (redirect) την είσοδο και την έξοδο του προγράμματος και να χρησιμοποιήσει **αρχεία κειμένου** (text files) αντί του πληκτρολογίου και της οθόνης. Αυτό γίνεται με τις εντολές `INPUT` και `OUTPUT`. Το παρακάτω πρόγραμμα διαβάζει την είσοδό του από το αρχείο “`file-to-read-from.txt`” και εμφανίζει τα αποτελέσματά του στο αρχείο “`file-to-write-to.txt`”.

```
PROGRAM redirection ()
{
  int n, i, sum = 0;

  INPUT(”file-to-read-from.txt”);
  OUTPUT(”file-to-write-to.txt”);

  n = READ_INT();
  FOR (i, 1 TO n)
    sum = sum + READ_INT();
  WRITELN(sum);
}
```

Όταν ανακατευθύνουμε την έξοδο με την εντολή `OUTPUT` πρέπει να είμαστε πολύ προσεκτικοί γιατί το αρχείο στο οποίο ανακατευθύνουμε την έξοδο, αν ήδη υπάρχει, χάνει τα προηγούμενα περιεχόμενά του.

2.5 Αριθμητικές και λογικές παραστάσεις

Μία παράσταση (έκφραση — expression) μπορεί γενικά να αποτελείται από:

- σταθερές (constants), π.χ. `42`, `3.1415926`

- μεταβλητές (variables), π.χ. `r`, `passed`
- συναρτήσεις (functions), π.χ. `sqrt`
- τελεστές (operators), π.χ. `+`, `-`, `*`
- βοηθητικά σύμβολα, π.χ. παρενθέσεις, κόμματα, κ.λπ.

Επίσης πρέπει να επισημάνουμε τους εξής δυο κανόνες:

- Μια παράσταση πρέπει να γράφεται κατά τέτοιον τρόπο ώστε να μην υπάρχει αμφιβολία για τη σημασία της. Αυτό διευκολύνεται με τη χρήση παρενθέσεων.
- Οι παρενθέσεις πρέπει να είναι "καλά ζυγισμένες" (balanced). Το πλήθος των δεξιών παρενθέσεων πρέπει να είναι ίσο με το πλήθος των αριστερών παρενθέσεων. Κάθε πρόθεμα της παράστασης πρέπει να περιέχει τουλάχιστον τόσες δεξιές παρενθέσεις όσες και αριστερές.

Τα δύο σημαντικότερα είδη παραστάσεων στη γλώσσα Pazcal είναι οι αριθμητικές και οι λογικές παραστάσεις. Οι πρώτες έχουν ως αποτέλεσμα τιμές των τύπων **`int`** ή **`REAL`**, δηλαδή ακέραιοι ή πραγματικοί αριθμοί, ενώ οι δεύτερες έχουν ως αποτέλεσμα τιμές του τύπου **`bool`**, δηλαδή **`true`** ή **`false`**.

2.5.1 Αριθμητικές πράξεις και παραστάσεις

Ας ξεκινήσουμε με τις αριθμητικές παραστάσεις. Στην Pazcal έχουν ως αποτέλεσμα τιμές που είναι ακέραιοι ή πραγματικοί αριθμοί. Ένα πρώτο πράγμα που πρέπει να γίνει σαφές είναι ότι, αν και στα μαθηματικά οι ακέραιοι αριθμοί είναι υποσύνολο των πραγματικών αριθμών, στις περισσότερες γλώσσες προγραμματισμού οι ακέραιες τιμές και οι πραγματικές τιμές είναι ουσιαστικά δύο διακριτά σύνολα. Δηλαδή είναι διαφορετικό το "`0`" από το "`0.0`", το "`1`" από το "`1.0`", το "`42`" από το "`42.0`", κ.ο.κ. Τα πρώτα είναι ακέραιες τιμές, τύπου **`int`**, ενώ τα δεύτερα είναι πραγματικές τιμές, τύπου **`REAL`**. Η βασική τους διαφορά βρίσκεται στο ότι ο υπολογιστής χρησιμοποιεί τελείως διαφορετική αναπαράσταση για τα μεν και για τα δε. Υπό προϋποθέσεις είναι δυνατό να μετατραπεί (αυτόματα) μία ακέραιη τιμή σε πραγματική, δηλαδή το "`42`" να γίνει "`42.0`", όπως θα δούμε παρακάτω. Το αντίστροφο στην Pazcal δε γίνεται ποτέ αυτόματα.

Στην Pazcal χρησιμοποιούνται πέντε ειδικά σύμβολα, τα οποία ανήκουν στο αλφάβητο της γλώσσας, που ονομάζονται αριθμητικοί τελεστές.

Τελεστής	Πράξη
`+`	Πρόσθεση
`-`	Αφαίρεση
`*`	Πολλαπλασιασμός
`/`	Διαίρεση (ακέραιη και πραγματική)
`%`	Ακέραιο υπόλοιπο

Οι αριθμητικοί τελεστές αντιστοιχούν στις βασικές αριθμητικές πράξεις που γνωρίζουμε από τα μαθηματικά. Μπορούν να εφαρμοστούν σε τελούμενα που είναι οποιουδήποτε αριθμητικού

τύπου, δηλαδή **int** ή **REAL**, εκτός από τον τελεστή % που μπορεί να εφαρμοστεί μόνο σε ακέραια τελούμενα, δηλαδή τύπου **int**.

Όταν και τα δύο τελούμενα είναι του ίδιου τύπου, τότε και το αποτέλεσμα είναι του ίδιου τύπου. Όταν όμως τα δύο τελούμενα είναι διαφορετικών τύπων, δηλαδή το ένα είναι **int** και το άλλο **REAL**, τότε πριν γίνει η πράξη αυτό που είναι **int** μετατρέπεται αυτόματα σε **REAL**. Ας δούμε μερικά παραδείγματα απλών αριθμητικών πράξεων και, δεξιά, το αποτέλεσμα που υπολογίζεται. Προσέξτε ποιες τιμές είναι ακέραιοι αριθμοί και ποιές πραγματικοί. Στην τελευταία πράξη γίνεται αυτόματη μετατροπή της ακέραιας τιμής `17` στην πραγματική `17.0`.

```
8 + 9                    17
6 * 7                    42
4.5 - 0.3                4.2
9.0 + 8.0                17.0
4.2 + 1                  5.1
2 * 1.3                  2.6
1.0 * 17                 17.0
```

Θα σταθούμε λίγο περισσότερο στην πράξη της διαίρεσης. Πολλές φορές στα μαθηματικά, αντιμετωπίζουμε τη διαίρεση μεταξύ ακεραίων αριθμών και τη διαίρεση μεταξύ πραγματικών αριθμών ως την ίδια πράξη. Για να το κάνουμε αυτό, όμως, πρέπει είτε να μετατρέπουμε πάντοτε τους ακέραιους αριθμούς σε πραγματικούς, είτε να δεχθούμε ότι κάποιες ακέραιες διαιρέσεις είναι αδύνατες, π.χ. το 42 δε διαιρείται με το 17. Η **ακέραια διαίρεση** (integer division), που από το Δημοτικό έχουμε μάθει ότι δίνει πηλίκο και υπόλοιπο, είναι μία πράξη ουσιαστικά διαφορετική από τη διαίρεση των πραγματικών αριθμών.

Στην Pazcal (όπως και στη C), ο τελεστής / χρησιμοποιείται για δύο σκοπούς: όταν εφαρμοστεί σε ακέραια τελούμενα υπολογίζει το πηλίκο της ακέραιας διαίρεσης, ενώ όταν εφαρμοστεί σε πραγματικά τελούμενα (ή σε ένα ακέραιο και ένα πραγματικό, λόγω της αυτόματης μετατροπής που είδαμε) υπολογίζει το αποτέλεσμα της πραγματικής διαίρεσης. Ο τελεστής % εφαρμόζεται μόνο σε ακέραια τελούμενα και υπολογίζει το υπόλοιπο της ακέραιας διαίρεσης. Ας δούμε μερικά παραδείγματα:

```
5 / 2                    2
5.0 / 2.0                2.5
4.2 / 17                 0.247059...
5 % 2                    1
-17 / 5                  -3
-17 % 5                  -2
17 / -5                  -3
17 % -5                  2
-17 / -5                 3
-17 % -5                 -2
3 / 0                    σφάλμα εκτέλεσης
42.0 / 0.0               σφάλμα εκτέλεσης
```

Δύο παρατηρήσεις:

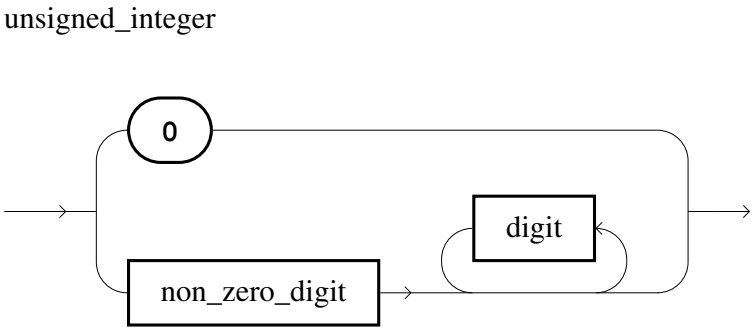
- Στην ακέραια διαίρεση, το πηλίκο προκύπτει παίρνοντας το αποτέλεσμα της πραγματικής διαίρεσης και αποκόπτοντας το κλασματικό μέρος (δηλαδή $-17/5 = -3.4 \rightsquigarrow -3$). Το

υπόλοιπο είναι τέτοιο ώστε πάντα να ισχύει: $(\texttt{a/b}) \times \texttt{b} + \texttt{a\%b} = \texttt{a}$. Ακριβώς έτσι ορίζεται και η ακέραια διαίρεση στη γλώσσα C. Προσέξτε ότι αυτό μπορεί να μην συμπίπτει με αυτό που περιμένατε από τα μαθηματικά, όταν κάποια από τα τελούμενα είναι αρνητικοί ακέραιοι αριθμοί.

- Η διαίρεση με το μηδέν (ακέραιο ή πραγματικό) απαγορεύεται και, αν γίνει, οδηγεί σε σφάλμα εκτέλεσης.

Τώρα θα ασχοληθούμε με τα μέρη που αποτελούν τις αριθμητικές παραστάσεις:

1. **Ακέραια σταθερά χωρίς πρόσημο**: όπου το digit είναι ένα από τα δεκαδικά ψηφία `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` και `9` και το non_zero_digit είναι ένα από τα ψηφία εκτός του `0`.

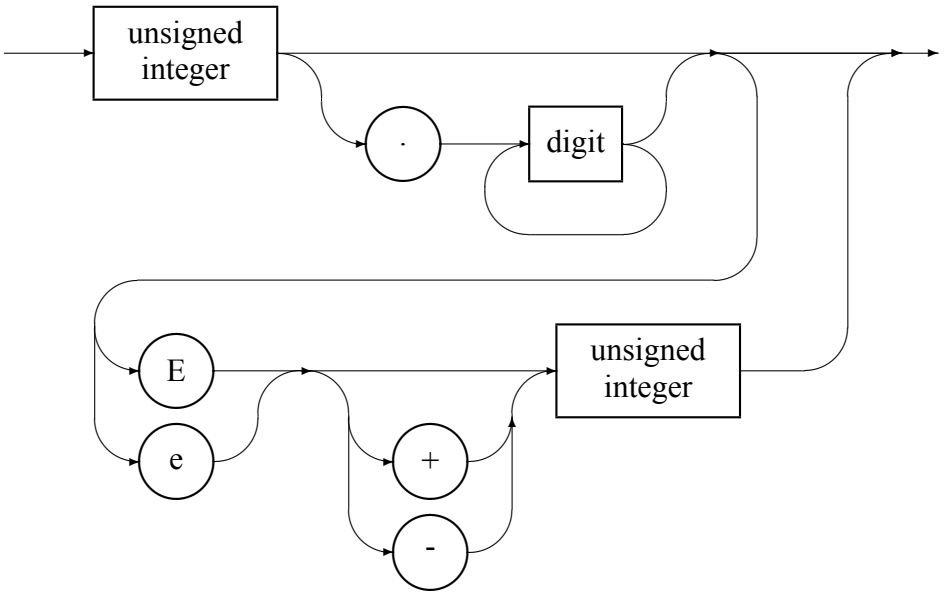


   Π.χ. `3789`, `24`, `0`, `100` αλλά όχι `007`: οι μη μηδενικές ακέραιες σταθερές δεν μπορούν να ξεκινούν με μηδέν (ο λόγος που υπάρχει αυτός ο περιορισμός θα φανεί στο κεφάλαιο 9).

2. **Αριθμός χωρίς πρόσημο**: Οι τύποι **`int`** και **`REAL`** περιγράφουν τους ακεραίους και πραγματικούς αριθμούς αντίστοιχα. Οι πραγματικοί αριθμοί μπορούν να γραφούν είτε με το συνηθισμένο τρόπο ή με τον εκθετικό. Στον συνηθισμένο τρόπο ο, πραγματικός αριθμός είναι ένας αριθμός που αποτελείται από έναν ακέραιο αριθμό (το ακέραιο μέρος), τη δεκαδική υποδιαστολή (που συμβολίζεται με μία τελεία) και ένα ή περισσότερα δεκαδικά ψηφία. Η εκθετική γραφή είναι εξυπηρετική όταν διαχειριζόμαστε πολύ μεγάλους αριθμούς ή πολύ μικρούς αριθμούς. Δυο παραδείγματα είναι: `1.234E2` και `23.456e-32`. Π.χ. αν θέλουμε να αναπαραστήσουμε την τιμή του φορτίου του ηλεκτρονίου η οποία είναι 1.602×10^{-19} τότε στην Pazcal θα γράψουμε: `1.602E-19`.

   Το "`E`" (κεφαλαίο ή μικρό) σημαίνει εκθέτη με βάση 10, δηλαδή το `1.602E-19` είναι 1.602 φορές το 10 στην -19η δύναμη. Ο εκθέτης είναι ακέραιος.

unsigned_number

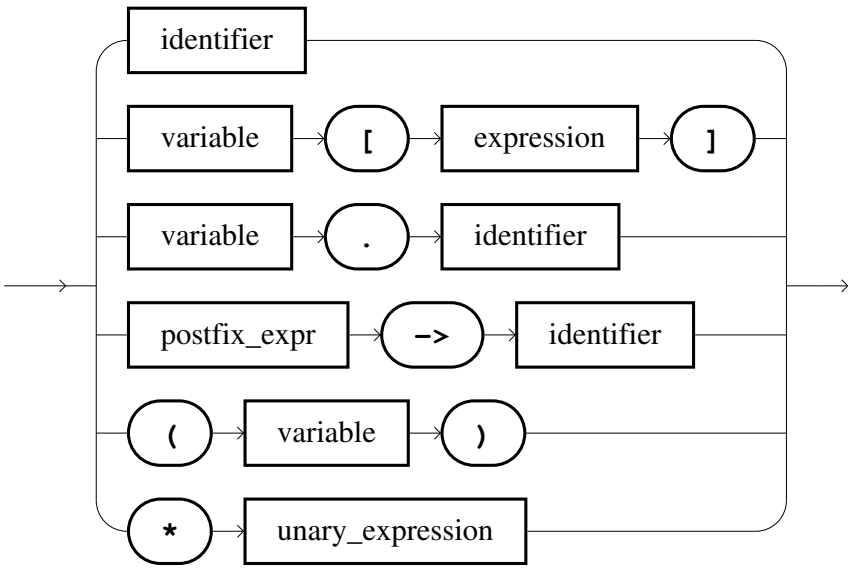


Μερικά παραδείγματα: `563.67E-6`, `25E3`, `6.9`, `42`, όχι όμως `.27`, `0.3E*2` ή `42.` (δηλαδή με υποδιαστολή αλλά χωρίς κανένα δεκαδικό ψηφίο μετά από αυτήν).

Προσέξτε ότι το "`E`" είναι μέρος της σύνταξης των αριθμητικών σταθερών και όχι αριθμητικός τελεστής. Δεν μπορεί να χρησιμοποιηθεί γενικά για την ύψωση του 10 σε κάποια δύναμη. Για παράδειγμα, αν οι μεταβλητές `x` καιη έχουν τις τιμές 4.2 και 7, αντίστοιχα, δεν μπορούμε να γράψουμε "`xEn`" για να υπολογίσουμε την τιμή 4.2×10^7. Ούτε "`4.2En`", ούτε "`xE7`".

3. **Μεταβλητή**: Την έννοια της μεταβλητής τη γνωρίσαμε από το πρώτο κιόλας παράδειγμα. Εδώ θα περιοριστούμε στο συντακτικό διάγραμμα, το οποίο μας δείχνει ότι υπάρχουν πολλά περισσότερα είδη μεταβλητών από αυτές που είδαμε μέχρι τώρα. Οι απλές μεταβλητές που έχουμε ήδη δει είναι ονόματα (αναγνωριστικά — identifiers).

variable



Αυτή τη στιγμή δε μας ενδιαφέρει να επικεντρωθούμε στη σημασία των άλλων ειδών μεταβλητών: θα τις γνωρίσουμε σε επόμενα κεφάλαια όταν συναντήσουμε τους πίνακες (arrays), τις δομές (structs) και τις ενώσεις (unions), και τους δείκτες (pointers). Ένα όμως πολύ ενδιαφέρον χαρακτηριστικό αυτού του συντακτικού διαγράμματος είναι ότι είναι **αναδρομικό** (recursive), δηλαδή αυτοαναφορικό: η σύνταξη της variable ορίζεται χρησιμοποιώντας την ίδια την έννοια variable! Η δεύτερη γραμμή, για παράδειγμα, λέει ότι μπορούμε να φτιάξουμε μία variable αν ξεκινήσουμε από μία (άλλη) variable και προσθέσουμε στο τέλος μία έκφραση μέσα σε αγκύλες. Η προτελευταία γραμμή λέει ότι μπορούμε να φτιάξουμε μία μεταβλητή βάζοντας μία (άλλη) μεταβλητή ανάμεσα σε παρενθέσεις. Με αυτόν τον τρόπο εξασφαλίζεται ότι οι παρενθέσεις θα είναι καλά ζυγισμένες!

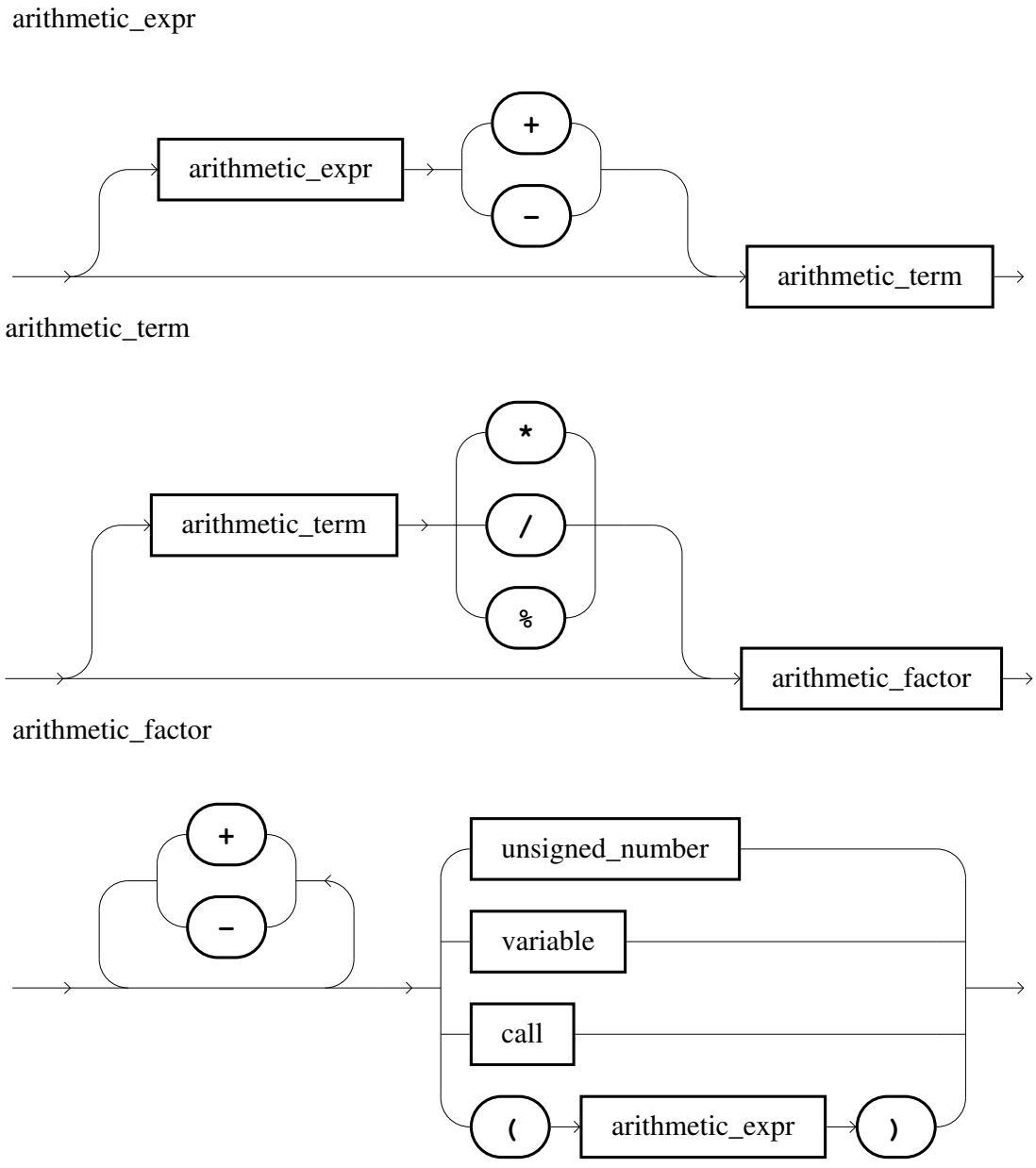
Παραδείγματα για μεταβλητές:

```
ub40
a[i]
dp[i+3][j-1]
myself.age
person[2].sex
a2a[7+din[j]]
x.a[7+z[8+i[3]]]
myself.birthdate.year
*(p+5)
```

Πολλές φορές είναι εύκολο σε ένα συντακτικό διάγραμμα να αντικαταστήσουμε την αναδρομή με βέλη που σχηματίζουν κύκλους. Άλλες φορές, όμως, αυτό είναι αδύνατο να γίνει! Προσπαθήστε αν θέλετε, για άσκηση, να φτιάξετε ένα μη αναδρομικό συντακτικό διάγραμμα για τη variable. Αν λείπει η προτελευταία γραμμή (αυτή με τις παρενθέσεις), γίνεται. Με αυτήν, όμως, είναι αδύνατο... Δεν είναι δυνατόν να φτιάξετε συντακτικό διάγραμμα που να περιγράφει τις καλά ζυγισμένες παρενθέσεις που να μην είναι αναδρομικό!

4. **Αριθμητική παράσταση**: Τα συντακτικά διαγράμματα που φαίνονται στο Σχήμα 2.3 ορίζουν την έννοια της αριθμητικής παράστασης (arithmetic expression). Είναι επίσης αναδρομικά, αλλά αυτή τη φορά είναι **έμμεσα αναδρομικά**, δηλαδή για τον ορισμό της αριθμητικής παράστασης (arithmetic_expr) απαιτείται η έννοια του αριθμητικού όρου (arithmetic_term), για τον ορισμό του αριθμητικού όρου απαιτείται η έννοια του αριθμητικού παράγοντα (arithmetic_factor) και για τον ορισμό του αριθμητικού παράγοντα απαιτείται η αριθμητική παράσταση. Ο λόγος που απαιτούνται όλα αυτά τα συντακτικά διαγράμματα είναι για να γίνει σαφής προτεραιότητα και η προσεταιριστικότητα των τελεστών.

   Με την έννοια της **προτεραιότητας** (precedence) μπορούμε να αναφερθούμε σε κάτι που γνωρίζουμε από τα μαθηματικά: σε μία αριθμητική παράσταση, ο πολλαπλασιασμός "δένει" πιο ισχυρά από την πρόσθεση. Αυτό σημαίνει ότι όταν βλέπουμε "`x+2*y`" καταλαβαίνουμε "`x+(2*y)`" και όχι "`(x+2)*y`". Ο τελεστής του πολλαπλασιασμού, λοιπόν, έχει μεγαλύτερη προτεραιότητα από εκείνον της πρόσθεσης. Για την ακρίβεια, οι τρεις "πολλαπλασιαστικοί" (multiplicative) τελεστές της Pazcal (`*`, `/` και `%`) έχουν **μεγαλύτερη** προτεραιότητα από τους δύο "προσθετικούς" (additive) τελεστές της Pazcal (`+` και `-`). Αυτό είναι εμφανές και στα συντακτικά διαγράμματα στο Σχήμα 2.3: μία αριθμητική παράσταση μπορεί να είναι ένα άθροισμα όρων, που καθένας μπορεί να είναι ένα γινόμενο παραγόντων.



Σχήμα 2.3. Αριθμητικές παραστάσεις.

Με την έννοια της **προσεταιριστικότητας** (associativity) μπορούμε επίσης να αναφερθούμε σε κάτι που γνωρίζουμε από τα μαθηματικά: σε μία ακολουθία προσθέσεων και αφαιρέσεων, οι πράξεις εκτελούνται από αριστερά προς τα δεξιά. Αυτό σημαίνει ότι όταν βλέπουμε "`x-2+y`" καταλαβαίνουμε "`(x-2)+y`" και όχι "`x-(2+y)`". Οι τελεστές της πρόσθεσης και της αφαίρεσης, λοιπόν, είναι **αριστερά προσεταιριστικοί**. Το ίδιο και οι "πολλαπλασιαστικοί" τελεστές. Και πάλι αυτό είναι εμφανές στα συντακτικά διαγράμματα στο Σχήμα 2.3: σε ένα άθροισμα, ο αριστερός προσθετέος είναι αριθμητική παράσταση (που μπορεί και πάλι να είναι άθροισμα) ενώ ο δεξιός είναι όρος (που μπορεί να είναι γινόμενο αλλά όχι άθροισμα, εκτός αν χρησιμοποιήσουμε παρενθέσεις).

Παραδείγματα αριθμητικών παραστάσεων:

```
x*2 + y*3
f(x, 3 + a[i].bc)
3 * a / 4 % n
-5/3 * bla + 7 * (-5/3 * blu) / f(7, x)
```

Το τρίτο παράδειγμα είναι ισοδύναμο με:

```
((3 * a) / 4) % n
```

Με βάση την προτεραιότητα και την προσεταιριστικότητα των τελεστών μπορούμε να ερμηνεύσουμε σωστά κάθε αριθμητική παράσταση. Η σειρά όμως με την οποία ο υπολογιστής θα εκτελέσει τις πράξεις για την αποτίμηση της παράστασης δεν ορίζεται πάντα πλήρως από τους κανόνες προτεραιότητας και προσεταιριστικότητας. Για παράδειγμα, στην πρώτη από τις παραπάνω παραστάσεις, ο υπολογιστής είναι γενικά ελεύθερος να υπολογίσει τα `x*2` και `y*3` με όποια σειρά θέλει (και στη συνέχεια να αθροίσει τα αποτελέσματα). Η σειρά που θα επιλέξει δεν επηρεάζει το αποτέλεσμα σε αυτό το παράδειγμα, αργότερα όμως θα δούμε ότι η σειρά αποτίμησης είναι σημαντική και δεν είναι σωστό (στην Pazcal και στη C) να θεωρούμε δεδομένο ότι οι πράξεις θα εκτελούνται από αριστερά προς τα δεξιά.

2.5.2 Λογικές πράξεις και παραστάσεις

Μέχρι τώρα έχουμε δει τις λογικές μεταβλητές, που μπορούν να έχουν μόνο δύο τιμές: **`true`** ή **`false`** (τύπου **`bool`**).

- Δήλωση μεταβλητής τύπου **`bool`**, π.χ. `bool state;`
- Εκχώρηση τιμής σε αυτήν, π.χ. `state = true;`

Σε αυτό το σημείο θα μιλήσουμε για λογικές παραστάσεις (logical ή boolean expressions), που σχηματίζονται χρησιμοποιώντας τελεστές σύγκρισης και λογικούς τελεστές.

Τελεστές σύγκρισης (comparison operators)

Τελεστής	Σύγκριση
==	ίσο
<	μικρότερο
>	μεγαλύτερο
<=	μικρότερο ή ίσο
>=	μεγαλύτερο ή ίσο
!=	διάφορο

Προσέξτε το συμβολισμό του τελεστή ελέγχου ισότητας: "==" (διπλό ίσον — το απλό ίσον χρησιμοποιείται όπως είδαμε για την εντολή εκχώρησης). Επίσης, προσέξτε το συμβολισμό για το διάφορο: "`!=`" (όχι ίσον, καθώς το "`!`" θα δούμε σε λίγο ότι χρησιμοποιείται για τη λογική άρνηση). Σε άλλες γλώσσες (π.χ. στην Pascal) χρησιμοποιείται ο τελεστής "<>" για το διάφορο.

Οι παραστάσεις που σχηματίζονται με τους τελεστές σύγκρισης εκφράζουν λογικές προτάσεις, αφού η τιμή μίας σύγκρισης μπορεί να είναι μόνο **`true`** ή **`false`**. Μπορούμε να συγκρίνουμε αριθμούς, χαρακτήρες ή ακόμα και τις λογικές τιμές (οπότε ισχύει **`false`** $<$ **`true`**). Το αποτέλεσμα των συγκρίσεων συνήθως το χρησιμοποιούμε σε μία εντολή **`if`**, **`while`** ή **`do`** ... **`while`**, μπορούμε όμως και να το εκχωρήσουμε σε μία λογική μεταβλητή:

```
state = 5 > 1;
recent = year >= 2010;
```

Η έκφραση `5 > 1` είναι αληθής (γιατί το 5 είναι μεγαλύτερο του 1), άρα η λογική μεταβλητή `state` θα λάβει την τιμή **`true`**. Στη δεύτερη εκχώρηση, η τιμή που θα πάρει η μεταβλητή `recent` εξαρτάται από την τιμή της ακέραιας μεταβλητής `year`.

Λογικοί τελεστές (logical operators)

Στην Pazcal υπάρχουν τρεις λογικοί τελεστές, που δέχονται τελούμενα τύπου **`bool`** και επιστρέφουν αποτέλεσμα τύπου **`bool`**. Για να καταλάβουμε τη σημασία τους, μας χρειάζονται οι **πίνακες αληθείας** (truth tables) των λογικών πράξεων:

- **Λογικό "και"** (`&&` — and): Η λογική παράσταση "`p && q`" είναι αληθής μόνο όταν και οι δυο λογικές παραστάσεις `p` και `q` είναι αληθείς.
- **Λογικό "ή"** (`||` — or): Η λογική παράσταση "`p || q`" είναι ψευδής μόνο όταν και οι δυο λογικές παραστάσεις `p` και `q` είναι ψευδείς.
- **Λογική άρνηση** (`!` — not): Η λογική παράσταση "`!p`" είναι αληθής μόνο αν η λογική παράσταση `p` είναι ψευδής.

p	q	p && q
false	**false**	**false**
false	**true**	**false**
true	**false**	**false**
true	**true**	**true**

p	q	p || q
false	**false**	**false**
false	**true**	**true**
true	**false**	**true**
true	**true**	**true**

p	!q
false	**true**
true	**false**

Σε αντίθεση με τις πράξεις `&&` και `||`, που είναι δυαδικές (binary) δηλαδή δέχονται δύο τελούμενα, η πράξη `!` δέχεται ένα τελούμενο (unary).

Προτεραιότητα και προσεταιριστικότητα στις λογικές παραστάσεις

Στο Σχήμα 2.4 ορίζεται η σύνταξη των λογικών παραστάσεων (logical expressions), που πάλι αποτελούνται από λογικούς όρους (logical terms) και λογικούς παράγοντες (logical factors). Η λογική άρνηση έχει μεγαλύτερη προτεραιότητα από όλους τους τελεστές σύγκρισης και τους δύο λογικούς τελεστές, επομένως είναι υποχρεωτικό να βάλουμε παρενθέσεις όταν γράφουμε, π.χ. "`!(a<b)`". Οι τελεστές σύγκρισης έχουν μεγαλύτερη προτεραιότητα από το λογικό "και", που με τη σειρά του έχει μεγαλύτερη προτεραιότητα από το λογικό "ή". Οι δύο λογικοί τελεστές είναι αριστερά προσεταιριστικοί. Οι τελεστές σύγκρισης δεν είναι προσεταιριστικοί, δηλαδή απαγορεύεται να γράψουμε π.χ. "`a < b < c`" (αυτό που εννοούμε συνήθως στα μαθηματικά με αυτή την παράσταση γράφεται σε Pazcal ως "`a < b && b < c`").

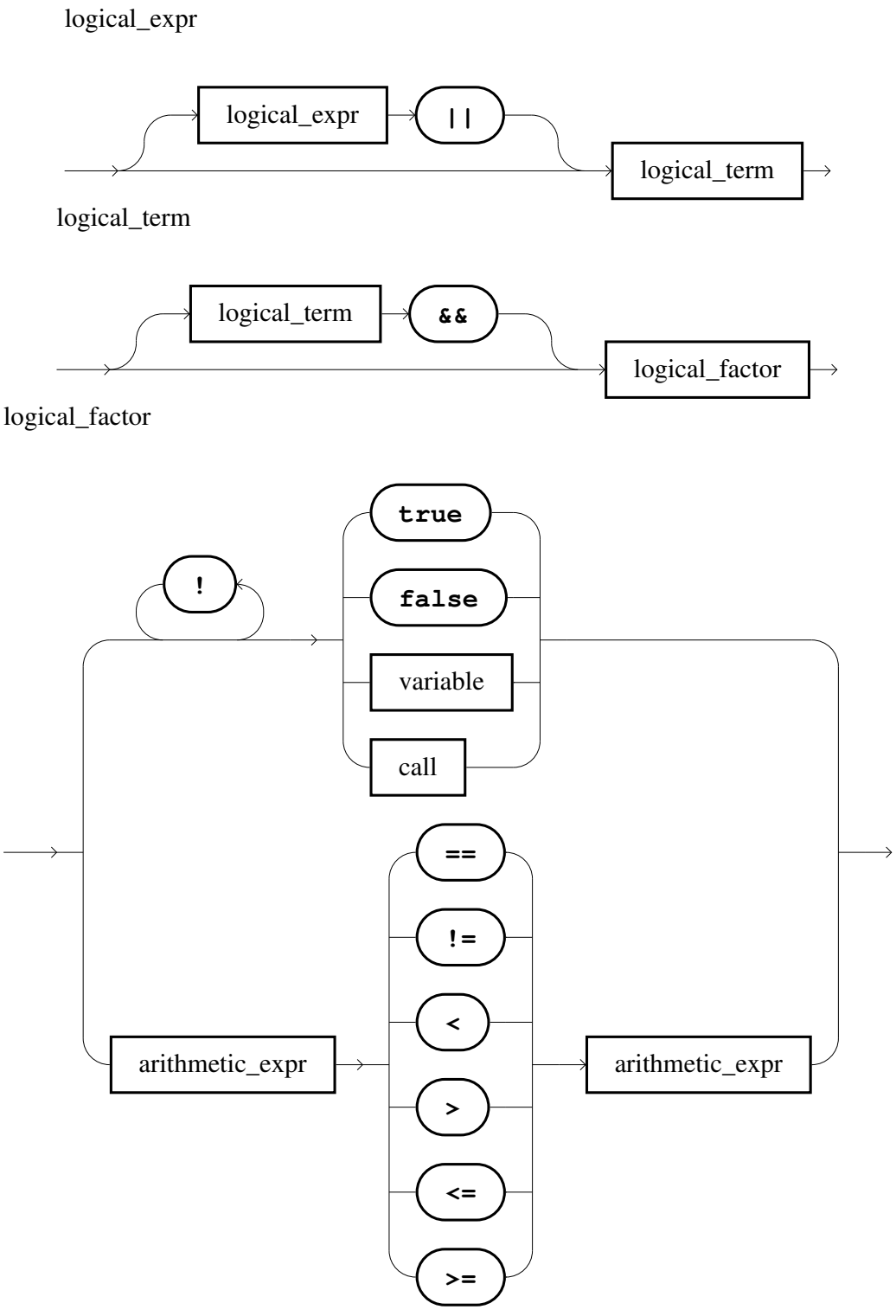
Παραδείγματα λογικών παραστάσεων:

```
x == y && z < a[i]
!done && b || 3/c == d+1
```

Το τελευταίο παράδειγμα είναι ισοδύναμο με:

```
((!done) && b) || ((3/c) == (d+1))
```

Οι λογικοί τελεστές "`&&`" και "`||`" στην Pazcal (όπως και στη C, αντίθετα όμως από την Pascal) έχουν μία ιδιαιτερότητα. Αν το αποτέλεσμα της πράξης είναι γνωστό από το πρώτο κιόλας τελούμενο (προσέξτε ότι **`false`** `&& x ==` **`false`** και **`true`** `|| x ==` **`true`** οποιαδήποτε και αν είναι η τιμή του `x`), τότε το δεύτερο τελούμενο **δεν αποτιμάται καθόλου**. Αυτό μας ενδιαφέρει, π.χ. στην περίπτωση του τελευταίου παραδείγματος παραπάνω. Αν η μεταβλητή `c` έχει την τιμή μηδέν, τότε περιμένουμε η αποτίμηση της παράστασης να οδηγήσει σε σφάλμα εκτέλεσης (διαίρεση με το μηδέν). Αυτό όμως δε θα συμβεί αν το αποτέλεσμα της λογικής παράστασης "`!done && b`" είναι **`true`**, γιατί τότε το δεύτερο τελούμενο του "`||`" δε θα υπολογιστεί (το τελικό αποτέλεσμα θα είναι προφανώς **`true`** σε κάθε περίπτωση). Ο τρόπος αυτός υπολογισμού των λογικών εκφράσεων ονομάζεται **βραχυκύκλωση** (short-circuit). Όλοι οι υπόλοιποι δυαδικοί τελεστές που έχουμε δει ως τώρα, σε αντίθεση με τους "`&&`" και "`||`", υπολογίζουν πάντα και τα δύο τους τελούμενα.



Σχήμα 2.4. Λογικές παραστάσεις.

2.6 Τα προγράμματα σε Pascal

Πρόγραμμα example1, σελ. 11

```
program example1 (input, output);
  var r, a : real;
begin
  write('Give the radius: ');
  read(r);
  a := 3.1415926 * r * r;
  writeln('The area is: ', a)
end.
```

Πρόγραμμα operation, σελ. 26

```
program operation (input, output);
  var first, second, result : integer;
      operator              : char;
begin
  read(first, operator, second);

  case operator of
    '+' : result := first + second;
    '-' : result := first - second;
    '*' : result := first * second;
    '/' : result := first div second
  end;
  writeln('The result is: ', result)
end.
```

Πρόγραμμα addition1, σελ. 28

```
program addition1 (input, output);
  var first, second : integer;
begin
  write('First: ');
  read(first);
  write('Second: ');
  read(second);
  writeln('Result: ', first + second)
end.
```

Πρόγραμμα addition2, σελ. 28

```
program addition2 (input, output);
  var first, second : integer;
begin
  write('First: ');
  readln(first);
```

```
  write('Second: ');
  read(second);
  writeln('Result: ', first + second)
end.
```

2.7 Τα προγράμματα σε C

Πρόγραμμα example1, σελ. 11

```
#include <stdio.h>

int main ()
{
  double r, a;

  printf("Give the radius: ");
  scanf("%lf", &r);
  a = 3.1415926 * r * r;
  printf("The area is: %lf\n", a);

  return 0;
}
```

Πρόγραμμα operation, σελ. 26

```
#include <stdio.h>

int main ()
{
  int first, second, result;
  char operator;

  scanf("%d%c%d", &first, &operator, &second);

  switch (operator) {
    case '+': result = first + second; break;
    case '-': result = first - second; break;
    case '*': result = first * second; break;
    case '/': result = first / second; break;
  }
  printf("The result is: %d\n", result);
  return 0;
}
```

Πρόγραμμα addition1, σελ. 28

```
#include <stdio.h>
```

```
int main ()
{
  int first, second;
  printf(”First: ”);
  scanf(”%d”, &first);
  printf(”Second: ”);
  scanf(”%d”, &second);
  printf(”Result: %d\n”, first + second);
  return 0;
}
```

Πρόγραμμα addition2, σελ. 28

```
#include <stdio.h>

int main ()
{
  int first, second;
  printf(”First: ”);
  scanf(”%d”, &first);
  while (getchar() != ’\n’);    // δεν υπάρχει αντίστοιχο της SKIP_LINE στη C
  printf(”Second: ”);
  scanf(”%d”, &second);
  printf(”Result: %d\n”, first + second);
  return 0;
}
```

Κεφάλαιο 3

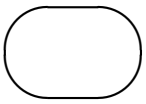
Δομές ελέγχου

Οι εντολές του προγράμματος εκτελούνται συνήθως από την αρχή μέχρι το τέλος, με την ίδια σειρά που εμφανίζονται στο κείμενο του προγράμματος. Αυτό είναι επιθυμητό μόνο σε πολύ απλά προγράμματα. Οι δομές ελέγχου τροποποιούν τη σειρά εκτέλεσης των εντολών του προγράμματος (δηλαδή τη **ροή ελέγχου** — control flow) κατά τη βούληση του προγραμματιστή.

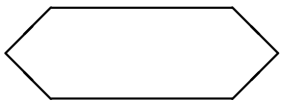
3.1 Λογικά διαγράμματα

Πολλοί χρησιμοποιούν **λογικά διαγράμματα ροής** (flow charts) για να περιγράψουν τη ροή ελέγχου ενός προγράμματος, δηλαδή τη σειρά με την οποία γίνονται οι διάφορες λειτουργίες του προγράμματος. Τα λογικά διαγράμματα δεν πρέπει να συγχέονται με τα συντακτικά διαγράμματα που είδαμε στο προηγούμενο κεφάλαιο: εκείνα περιγράφουν τη σύνταξη μίας γλώσσας, ενώ τα λογικά διαγράμματα περιγράφουν τη ροή εκτέλεσης των προγραμμάτων.

Τα βασικά σύμβολα που χρησιμοποιούνται στα λογικά διαγράμματα είναι:

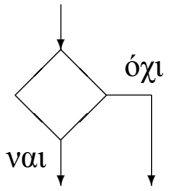


Το oval κουτί χρησιμοποιείται σε όλα τα λογικά διαγράμματα: μία φορά με τη λέξη ΑΡΧΗ, όταν συμβολίζει την αρχή του λογικού διαγράμματος, και τουλάχιστον άλλη μια με τη λέξη ΤΕΛΟΣ, όταν συμβολίζει το τέλος του.



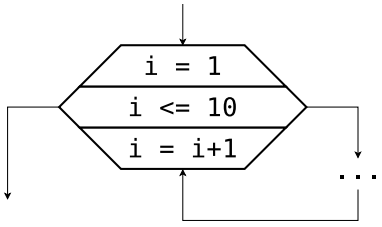
Στο εξάγωνο γράφονται συνοπτικά λειτουργίες ολόκληρες ή διαδικασίες, που δεν επεξηγούνται με λεπτομέρειες.

Στο ορθογώνιο παραλληλόγραμμο αναγράφονται απλές εντολές.



Στο ρόμβο γράφονται ερωτήσεις (π.χ. συγκρίσεις) όπου η απάντηση είναι ναι ή όχι. Χρησιμοποιείται για λογική διακλάδωση ανάλογα με την αληθοτιμή της συνθήκης.

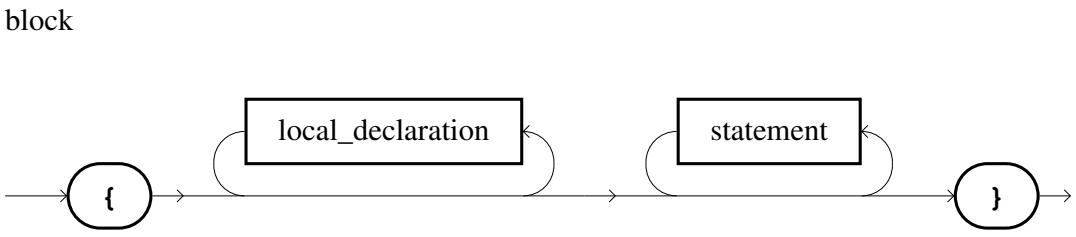
Το πλάγιο παραλληλόγραμμο συμβολίζει I/O λειτουργία: είτε την είσοδο δεδομένων στον υπολογιστή, όταν περιέχει τη λέξη ΔΙΑΒΑΣΕ, είτε την έξοδο των αποτελεσμάτων, όταν περιέχει τη λέξη ΤΥΠΩΣΕ. Σ' αυτό καταγράφονται ονόματα μεταβλητών στα οποία θα καταχωρηθούν τα δεδομένα, ή παραστάσεις τιμές των οποίων εμφανίζονται στην έξοδο. Επίσης, το πλάγιο παραλληλόγραμμο χρησιμοποιείται για να δηλωθεί ότι εμφανίζεται στην οθόνη (έξοδο γενικότερα) κάποιο μήνυμα επεξηγηματικό για τον χρήστη, είτε διαγνωστικό λάθους.



Χρησιμοποιείται για βρόχο FOR με μεταβλητή ελέγχου (εδώ `i`) που κυμαίνεται μεταξύ μίας αρχικής και μίας τελικής τιμής (εδώ `1` και `10`) και αυξάνεται ή μειώνεται με κάποιο βήμα (εδώ `+1`). Ο αριθμός των επαναλήψεων είναι πεπερασμένος και γνωστός τη στιγμή που αρχίζει να εκτελείται ο βρόχος.

3.2 Σύνθετη εντολή

Συχνά στα συντακτικά διαγράμματα (και στα αντίστοιχα λογικά διαγράμματα ροής) των δομών ελέγχου που θα συναντήσουμε σε αυτό το κεφάλαιο απαιτείται μια μόνο εντολή (statement). Αν ο προγραμματιστής θέλει σε εκείνο το σημείο να εκτελεστούν περισσότερες από μία εντολές, έχει τη δυνατότητα να χρησιμοποιήσει την **συνθέτη εντολή** (compound statement) που συνθέτει περισσότερες εντολές σε μία. Συντακτικά, η σύνθετη εντολή έχει τη μορφή του **block** που είδαμε στη σελίδα 13 και αντιγράφουμε εδώ.



Τα άγκιστρα στη σύνθετη εντολή είναι το αντίστοιχο των παρενθέσεων, στις αριθμητικές και τις λογικές παραστάσεις: περικλείουν περισσότερες εντολές που θέλουμε να συμπεριφέρονται ως μία, απλή εντολή. Στην αρχή ενός block, πριν τις εντολές που το αποτελούν, μπορούν να γραφούν δηλώσεις οι οποίες είναι **τοπικές** (local), δηλαδή ορατές μόνο στο εσωτερικό αυτού του block. (Η δήλωση τοπικών μεταβλητών στις σύνθετες εντολές είναι ιδιαίτερο χαρακτηριστικό της γλώσσας C και των απογόνων της. Σε γλώσσες σαν την Pascal τοπικές δηλώσεις γίνονται μόνο στα σώματα υποπρογραμμάτων.)

Σε μία σύνθετη εντολή, όπως είπαμε και στην αρχή του κεφαλαίου, οι εντολές εκτελούνται η μία μετά την άλλη, με τη σειρά που εμφανίζονται.

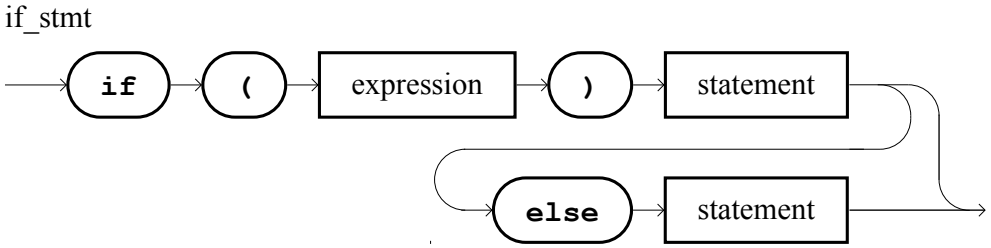
3.3 Απόφαση, έλεγχος συνθήκης

Για λήψη αποφάσεων και τον έλεγχο συνθηκών υπάρχουν δυο δομές ελέγχου: η εντολή **`if`** και η εντολή **`switch`**. Με χρήση αυτών, ο προγραμματιστής μπορεί να ορίσει ότι κάποιες εντολές θα εκτελούνται μόνο αν κάποια συνθήκη είναι αληθής.

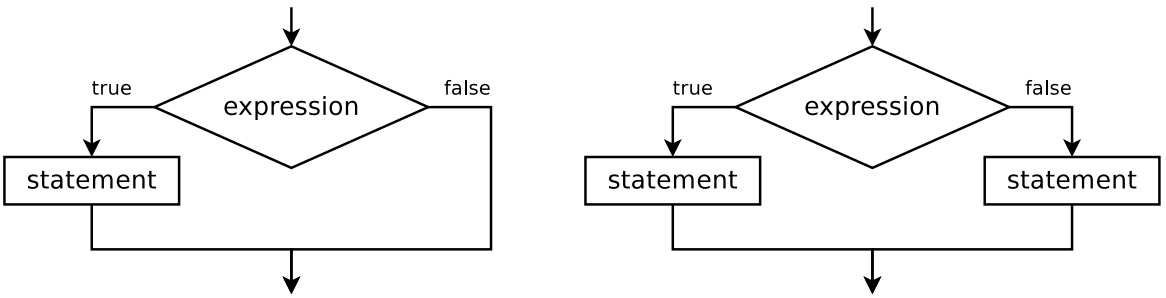
3.3.1 Η εντολή if

Η δυνατότητα επιλογών είναι βασική στην κατασκευή ενός προγράμματος. Πρέπει να είμαστε σε θέση να παίρνουμε αποφάσεις της μορφής: Αν ο λογαριασμός σου στην τράπεζα έχει τουλάχιστον x ευρώ, τότε έχεις δικαίωμα ανάληψης x ευρώ από το ΑΤΜ. Αυτό απαιτεί την ικανότητα του προγράμματος να συγκρίνει μεταβλητές, σταθερές και τιμές εκφράσεων, δηλαδή στην ουσία να αποτιμά λογικές παραστάσεις, και έπειτα να εκτελεί κάποιες εντολές ανάλογα με το αποτέλεσμα της αποτίμησης.

Το συντακτικό διάγραμμα της εντολής **`if`** είναι το ακόλουθο:



και τα αντίστοιχα λογικά διαγράμματα ροής, ανάλογα αν υπάρχει ή όχι το προαιρετικό σκέλος **`else`**, είναι τα ακόλουθα:



Η πιο απλή μορφή της **`if`** είναι π.χ. η εξής:

- **`if`** `(amount >= x) amount = amount - x;`
- **`if`** `(amount >= 1000000)` **`WRITELN`**`("Found a millionaire!");`

Στο πρώτο παράδειγμα, αν η συνθήκη "`amount >= x`" είναι αληθής τότε εκτελείται η εντολή "`amount = amount - x;`", ενώ αν η συνθήκη είναι ψευδής τότε η εντολή αυτή δεν εκτελείται. Στο δεύτερο παράδειγμα, αν η συνθήκη "`amount >= 1000000`" είναι αληθής τότε εμφανίζεται στην οθόνη το μήνυμα "`Found a millionaire!`".

Μετά από μία εντολή **`if`**, η εκτέλεση του προγράμματος συνεχίζεται με τις εντολές που έπονται.

Όπως προκύπτει από την παράγραφο 3.2, η εντολή που εκτελείται υπό συνθήκη μπορεί να είναι σύνθετη εντολή (compound statement). Ας δούμε μερικά παραδείγματα:

```
• if (x*x + y*y == z*z) {
    WRITESPLN(”Found a Pythagorian triple:”, x, y, z);
    s = (z-x)*(z-y)/2;
    WRITESPLN(”Did you know that”, s, ”is a perfect square?”);
  }
```

```
• if (a != 0) {
    REAL d = b*b-4*a*c;
    if (d >= 0) {
      REAL x = (-b+sqrt(d))/(2*a);
      WRITELN(”Found a solution: ”, x);
    }
  }
```

Αλλά μπορεί να είναι και η συνθήκη πιο σύνθετη όπως είδαμε στην παράγραφο 2.5.

```
• if (year >= 1901 && year <= 2000) WRITELN(”We’re in the 20th century!”);
```

```
• if (year % 4 == 0 AND year % 100 != 0 OR
      year % 400 == 0 AND year % 4000 != 0)
    WRITELN(”δίσεκτο έτος”);
```

Επίσης, όπως προκύπτει από το συντακτικό διάγραμμα, η εντολή **if** μπορεί να περιλαμβάνει προαιρετικά και ένα σκέλος **else**. Αυτό εκτελείται σε περίπτωση που η συνθήκη είναι ψευδής.

```
• if (grade >= 5) WRITELN(”You passed the exam.”);
  else            WRITELN(”I’m sorry, you failed.”);
```

```
• if (x % 2 == 0) WRITELN(”You have an even number: ”, x);
  else            WRITELN(”You have an odd number: ”, x);
```

```
• if (ch >= ’A’ && ch <= ’Z’) letter = ch;
  else                        WRITELN(”Not a capital letter”);
```

Προσέξτε ότι, στο δεύτερο παράδειγμα παραπάνω, η συνθήκη “`x % 2 == 0`” αληθεύει όταν ο ακέραιος αριθμός `x` είναι άρτιος. Εν γένει, η συνθήκη “`a % b == 0`” αληθεύει όταν ο αριθμός `a` διαιρείται (ακριβώς) από τον `b`.

Βεβαίως, στα δύο σκέλη της **if** μπορεί να είναι φωλιασμένες (nested) και άλλες εντολές **if**. Η περίπτωση που μία δεύτερη εντολή **if** εμφανίζεται στο σκέλος **else** είναι πολύ συνηθισμένη:

```
• if (x > y)      WRITELN("I win");
  else if (x < y) WRITELN("You win");
       else       WRITELN("We're tied");
```

Λόγω των ιδιοτήτων της ολικής διάταξης, το ίδιο αποτέλεσμα θα μπορούσαμε να έχουμε με τρία διαφορετικά συνεχόμενα **if**:

```
• if (x > y)  WRITELN("I win");
  if (x < y)  WRITELN("You win");
  if (x == y) WRITELN("We're tied");
```

Αυτό ίσως να φαίνεται ομορφότερο στην όψη, όμως, σε αυτήν την περίπτωση αποτιμούνται πάντα και οι τρεις συγκρίσεις, ενώ στην περίπτωση του πρώτου (φωλιασμένου) **if** αποτιμάται μόνο μία (στην καλύτερη περίπτωση) ή δύο (στη χειρότερη).

Πιο δύσκολη στην κατανόηση είναι η περίπτωση που υπάρχει φωλιασμένη εντολή **if** στο πρώτο σκέλος ενός άλλου **if**:

```
• if (x>0)
    if (y>0)       WRITELN("first quadrant");
    else if (y<0)  WRITELN("fourth quadrant");
    else           WRITELN("on the x-axis");
```

Σε αυτή την περίπτωση πρέπει να θυμόμαστε ότι ένα **else** αντιστοιχεί στο πλησιέστερο (προηγούμενο) **if** που δεν έχει ήδη αντιστοιχιστεί σε άλλο **else**. Δηλαδή, η παραπάνω εντολή είναι ισοδύναμη με την ακόλουθη (που είναι προτιμητέα γιατί δεν παρερμηνεύεται εύκολα):

```
• if (x>0) {
    if (y>0)      WRITELN("first quadrant");
    else if (y<0) WRITELN("fourth quadrant");
    else          WRITELN("on the x-axis");
  }
```

και διαφορετική από αυτήν (στην οποία τα μηνύματα που εμφανίζονται είναι εσφαλμένα):

```
• if (x>0) {
    if (y>0) WRITELN("first quadrant");
  }
  else if (y<0) WRITELN("fourth quadrant");  // should be third quadrant!
  else          WRITELN("on the x-axis");    // should be second quadrant!
```

Προσέξτε ότι έχουμε χρησιμοποιήσει τη στοίχιση έτσι ώστε να φαίνεται ευκολότερα ποιο **else** αντιστοιχεί σε ποιο **if**. Αυτό είναι πολύ καλή πρακτική γιατί κάνει τον κώδικα πολύ πιο ευανάγνωστο. Η στοίχιση όμως δεν αλλάζει σε τίποτα τον τρόπο με τον οποίο ο μεταγλωττιστής καταλαβαίνει το πρόγραμμά μας. Αν ο προγραμματιστής θέλει πραγματικά να γράψει την τελευταία από τις τρεις εκδοχές, τότε είναι υποχρεωμένος να χρησιμοποιήσει άγκιστρα (σύνθετη εντολή), για τον ίδιο λόγο που είναι υποχρεωμένος να χρησιμοποιήσει παρενθέσεις όταν θέλει να παρακάμψει την προτεραιότητα των τελεστών στις παραστάσεις.

3.3.2 Η εντολή switch

Στην προηγούμενη παράγραφο παρατηρήσαμε την εξής σύνταξη:

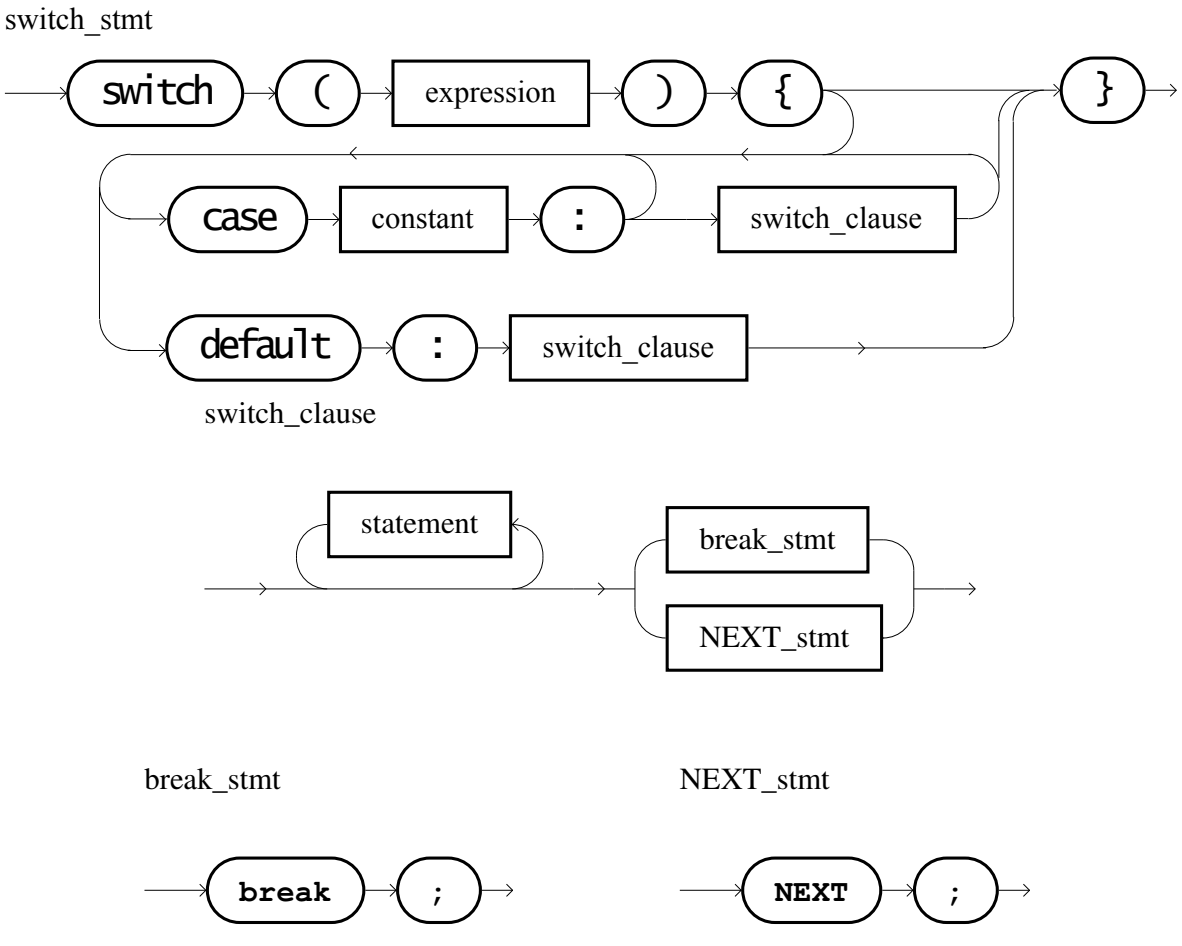
- **if** (*λογική παράσταση*) . . .
  else if (*λογική παράσταση*) . . .
  else . . .

  όπου τα αποσιωποιητικά αναφέρονται σε μια (απλή ή σύνθετη) εντολή. Παράδειγμα:

```
if (month == 1)       WRITELN("Ιανουάριος");
else if (month == 2)  WRITELN("Φεβρουάριος");
else if ...
...
else if (month == 12) WRITELN("Δεκέμβριος");
else                  WRITELN("άκυρος μήνας!");
```

Η προηγούμενη γραφή είναι πολύπλοκη και καλό είναι να αποφεύγεται, ιδιαίτερα όταν οι προς εξέταση λογικές εκφράσεις είναι πολλές. Επιπλέον, όσο μεγαλύτερη είναι η τιμή της μεταβλητής `month`, τόσο περισσότερες συγκρίσεις θα χρειαστούν μέχρι να βρεθεί η εντολή που θα εκτελεστεί.

Για όλους αυτούς τους λόγους όταν θέλουμε να γράψουμε κάτι σαν το παραπάνω χρησιμοποιούμε μια πιο σύντομη εντολή: την **switch**, η οποία συνοδεύεται από τις εντολές **break** και **NEXT**. Τα σχετικά συντακτικά διαγράμματα είναι:



Η εντολή **switch** δεν μπορεί να αντικαταστήσει όλες τις εντολές της μορφής **if** . . . **else if** . . . **else if** . . . προσφέρεται όμως για παραδείγματα όπως το παραπάνω με τους μήνες, το οποίο με τη βοήθειά της γράφεται ως εξής:

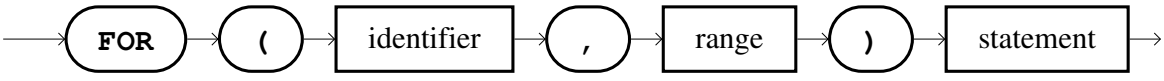
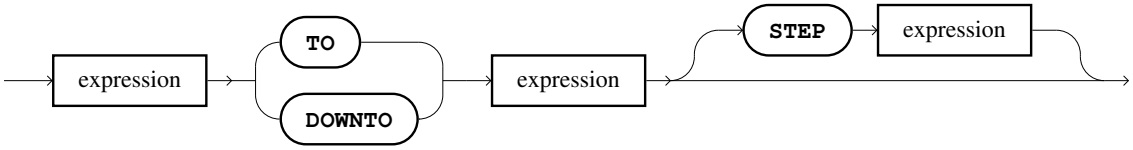
- ```
  switch (month) {
    case 1:  WRITELN(”Ιανουάριος”);    break;
    case 2:  WRITELN(”Φεβρόυάριος”);   break;
    case 3:  ...
    ...
    case 12: WRITELN(”Δεκέμβριος”);    break;
    default: WRITELN(”άκυρος μήνας!”); break;
  }
  ```

Κατά την εκτέλεση της **switch** αποτιμάται η έκφραση που δίνεται (εδώ η μεταβλητή month). Στη συνέχεια, ανάλογα με την τιμή της, η ροή ελέγχου μεταφέρεται στο αντίστοιχο **case**, αν υπάρχει. Προσέξτε ότι κάθε **case** ακολουθείται από ακριβώς μία σταθερή έκφραση. Αν δεν υπάρχει αντίστοιχο **case** αλλά υπάρχει σκέλος **default**, η ροή ελέγχου μεταφέρεται σε αυτό. Διαφορετικά, αν δηλαδή δεν υπάρχει ούτε σκέλος **default**, τότε η εκτέλεση συνεχίζεται με την εντολή που ακολουθεί τη **switch**. Η εντολή **break** στο τέλος (ή και ενδιάμεσα, όπως θα δούμε αργότερα) κάποιου από τα σκέλη του **switch** (switch_clause) προκαλεί το τέλος της εκτέλεσης της εντολής **switch** και μεταφέρει τη ροή ελέγχου στην εντολή που έπεται.

Άλλο παράδειγμα, στο οποίο φαίνεται ότι πριν από ένα σκέλος (switch_clause) μπορούν να υπάρχουν περισσότερα **case**:

- ```
  switch (month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      WRITELN(”31 days”); break;
    case 4: case 6: case 9: case 11:
      WRITELN(”30 days”); break;
    case 2:
      WRITELN(”28 or 29 days”); break;
  }
  ```

Η εντολή **NEXT** μπορεί να εμφανίζεται μόνο στο τέλος ενός σκέλους της εντολής **switch**. Σημαίνει ότι η ροή ελέγχου δεν πρέπει να βγεί από την εντολή **switch** στο τέλος αυτού του σκέλους (όπως θα συνέβαινε αν υπήρχε εκεί η εντολή **break**) αλλά να συνεχιστεί και στο αμέσως επόμενο σκέλος. Προσέξτε ότι δε γίνεται πλέον έλεγχος για την τιμή της έκφρασης του **switch**. Δείτε, για παράδειγμα, πώς αυτό μπορεί να χρησιμοποιηθεί για να μετρήσουμε τις μέρες που απομένουν μέχρι την πρωτοχρονιά. Υποθέτουμε ότι η σημερινή ημερομηνία δίνεται από τις μεταβλητές day και month και ότι το τρέχον έτος δεν είναι δίσεκτο.)

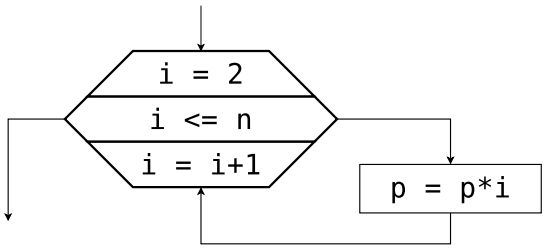
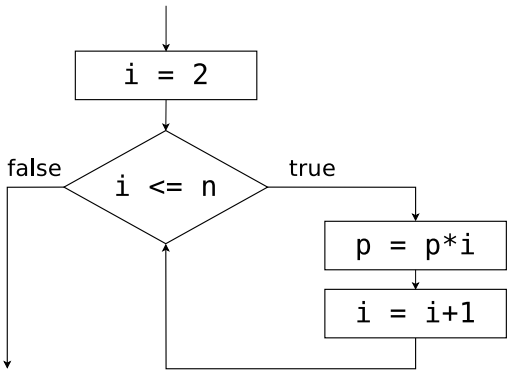
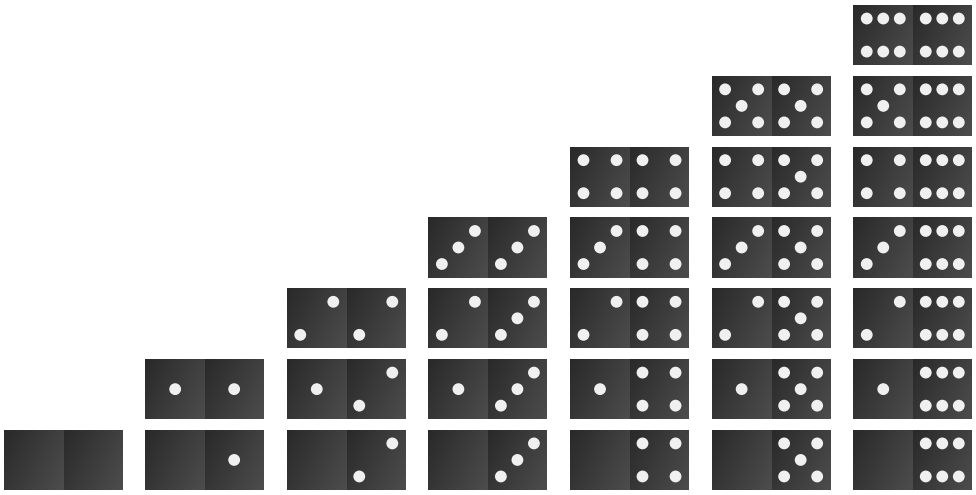
- ```
  remaining = 0;
  switch (month) {
    case 1:  remaining = remaining + 31; NEXT;
  ```
  ```

```
    case 2:  remaining = remaining + 28; NEXT;
    case 3:  remaining = remaining + 31; NEXT;
    case 4:  remaining = remaining + 30; NEXT;
    case 5:  remaining = remaining + 31; NEXT;
    ...
    case 11: remaining = remaining + 30; NEXT;
    case 12: remaining = remaining + 31; NEXT;
  }
  remaining = remaining - day + 1;
```

Στο τελευταίο σκέλος, είτε βάλουμε **NEXT** είτε **break** το αποτέλεσμα είναι το ίδιο, αφού η ροή ελέγχου σε αυτό το σημείο βγαίνει από το **switch**.

3.4 Οι επαναληπτικοί βρόχοι

Η πραγματική δύναμη του ηλεκτρονικού υπολογιστή είναι η επανάληψη. Στην ενότητα αυτή εξετάζονται οι δομές ελέγχου με τις οποίες σχηματίζονται **βρόχοι** (loops) στα προγράμματα, δηλαδή ακολουθίες εντολών που επαναλαμβάνεται η εκτέλεσή τους. Οι δομές αυτές στην Pazcal είναι οι εντολές **FOR**, **while** και **do** . . . **while**.

3.4.1 Εντολή FOR

Η εντολή **FOR** στην Pazcal χρησιμοποιείται όταν το πλήθος των επαναλήψεων που πρέπει να γίνουν είναι σταθερό και γνωστό τη στιγμή που αρχίζει η εκτέλεση του βρόχου. Το συντακτικό της διάγραμμα είναι το ακόλουθο.

FOR_stmt



range



Το αναγνωριστικό (identifier) ονομάζει τη **μεταβλητή ελέγχου** (control variable), δηλαδή μία απλή μεταβλητή — συνήθως ακέραιου τύπου — η τιμή της οποίας μεταβάλλεται κατά τη διάρκεια της εκτέλεσης του βρόχου. Ας ξεκινήσουμε με ένα απλό παράδειγμα:

```
PROGRAM counting ()
{
  int i;
  WRITELN(”Look: I can count!”);
  FOR (i, 1 TO 10) WRITELN(i);
}
```

```
Look: I can count!
1
2
3
4
5
6
7
8
9
10
```

Η τιμή της μεταβλητής ελέγχου `i` κυμαίνεται μέσα στην περιοχή (range) “`1 TO 10`”. Αρχικά παίρνει την τιμή 1 και διαδοχικά αυξάνεται κατά ένα, εως την τελική τιμή 10. Κάθε φορά, η εντολή “`WRITELN(i);`” στο σώμα του βρόχου εκτυπώνει την τρέχουσα τιμή της μεταβλητής `i`.

Ένα λίγο πιο σύνθετο παράδειγμα υπολογίζει και εκτυπώνει τις δυνάμεις του δύο μέχρι το 2^{10}. Εδώ, το σώμα του βρόχου είναι μία σύνθετη εντολή.

```
PROGRAM powers_of_two ()
{
  int i, p = 1;
  FOR (i, 0 TO 10) {
    WRITELN(2, ”^”, i, ” = ”, p);
    p = p * 2;
  }
}
```

```
2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
```

Προσέξτε ότι σε κάθε επανάληψη η τιμή της μεταβλητής `p` διπλασιάζεται, ξεκινώντας από την τιμή 1, με αποτέλεσμα κάθε φορά που φτάνει η εκτέλεση στην εντολή **`WRITELN`** να έχουμε $\mathtt{p} = 2^{\mathtt{i}}$. Μία τέτοια σχέση, που ισχύει σε κάθε επανάληψη του βρόχου, ονομάζεται **αναλλοίωτη του βρόχου** (loop invariant). Θα μιλήσουμε περισσότερο για αναλλοίωτες στο κεφάλαιο 8.

Πριν προχωρήσουμε σε πιο ενδιαφέροντα παραδείγματα βρόχων, ας αναφερθούμε λίγο στα είδη περιοχών (ranges) που υποστηρίζει η Pazcal, πάλι με παραδείγματα:

- `-10 TO 10`: αρχίζοντας από το -10, η τιμή της μεταβλητής ελέγχου θα αυξάνεται κατά ένα έως το 10. Το συνολικό πλήθος επαναλήψεων είναι $21 = 10 - (-10) + 1$. Γενικά σε μία περιοχή της μορφής “`lower TO upper`” όπου αρχικά ισχύει `lower` $\leq$ `upper` το πλήθος των επαναλήψεων ισούται με `upper` $-$ `lower` $+ 1$.

- `10 DOWNTO 3`: αρχίζοντας από το 10, η τιμή της μεταβλητής ελέγχου θα **μειώνεται** κατά ένα έως το 3. Το συνολικό πλήθος επαναλήψεων είναι $8 = 10 - 3 + 1$. Ομοίως με πρίν, για την περιοχή της μορφής “`upper DOWNTO lower`” όπου αρχικά ισχύει `lower` $\leq$ `upper` το πλήθος των επαναλήψεων ισούται με `upper` $-$ `lower` $+ 1$.

- `1 TO 20 STEP 3`: αρχίζοντας από το 1 η τιμή της μεταβλητής ελέγχου θα αυξάνεται κατά 3 έως το 20. Μόλις η τιμή της μεταβλητής υπερβεί το 20, η εκτέλεση του βρόχου διακό-

πτεται. Το συνολικό πλήθος επαναλήψεων αυτή τη φορά είναι 7 (οι τιμές της μεταβλητής ελέγχου θα είναι, διαδοχικά, $1, 4, 7, 10, 13, 16, 19$).

- `100 DOWNTO 50 STEP 5`: αρχίζοντας από το 100 η τιμή της μεταβλητής ελέγχου θα μειώνεται κατά 5 έως το 50. Το συνολικό πλήθος επαναλήψεων θα είναι 11 (οι τιμές της μεταβλητής ελέγχου θα είναι, διαδοχικά, $100, 95, 90, 85, 80, 75, 70, 65, 60, 55, 50$).

Βασική ιδιότητα του βρόχου **FOR** είναι ότι τα όρια της περιοχής και άρα και ο αριθμός των επαναλήψεων καθορίζεται πριν αρχίσουν οι επαναλήψεις:

1. Αποτιμάται η αρχική τιμή της μεταβλητής ελέγχου (η έκφραση πριν το **TO** ή το **DOWNTO**).
2. Αποτιμάται η τελική τιμή (η έκφραση μετά το **TO** ή το **DOWNTO**).
3. Αποτιμάται το βήμα (η έκφραση μετά το **STEP**): αν δεν υπάρχει χρησιμοποιείται η τιμή 1.

Παρατηρήσεις:

- Αν σε ένα βρόχο **FOR** οι ακραίες τιμές της περιοχής της μεταβλητής ελέγχου είναι ίσες, τότε το σώμα του βρόχου εκτελείται ακριβώς μία φορά.

- Αν σε ένα βρόχο **FOR** οι ακραίες τιμές είναι τέτοιες ώστε να μην ορίζουν περιοχή για τη μεταβλητή ελέγχου (π.χ. `5 TO 3` ή `1 DOWNTO 10`), τότε το σώμα του βρόχου δεν εκτελείται καμία φορά.

- Η μεταβλητή ελέγχου δεν είναι ορισμένη (δεν έχει τιμή) μετά το πέρας της εκτέλεσης του βρόχου.

- Δεν επιτρέπεται να αλλάζετε (π.χ. με ανάθεση) την τιμή της μεταβλητής ελέγχου μέσα στο σώμα του βρόχου.

- Ακόμα και εάν αλλάξουν, μέσα στο σώμα του βρόχου, οι τιμές μεταβλητών που εμπλέκονται στις παραστάσεις των ακραίων τιμών του βρόχου, αυτό δεν επιφέρεται μεταβολή στον αριθμό των επαναλήψεων γιατί όπως είπαμε, οι ακραίες τιμές προϋπολογίζονται.

Υπολογισμός παραγοντικού

Το παραγοντικό ενός φυσικού αριθμού n είναι το γινόμενο όλων των θετικών φυσικών αριθμών που δεν υπερβαίνουν το n. Δηλαδή:

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

Αρχική τιμή είναι το $0! = 1$.

Το πρόγραμμα που ακολουθεί ζητάει από το χρήστη να πληκτρολογήσει ένα φυσικό αριθμό n και εκτυπώνει την τιμή του $n!$. Δεξιά βλέπετε κάποια σενάρια εκτέλεσής του.

```
PROGRAM factorial ()
{
  int n, p, i;

  WRITE(”Give n: ”); n = READ_INT();

  p = 1;
  FOR (i, 2 TO n) p = p * i;
  WRITELN(n, ”! = ”, p);
}
```

```
Give n: 1↵
1! = 1

Give n: 4↵
4! = 24

Give n: 7↵
7! = 5040

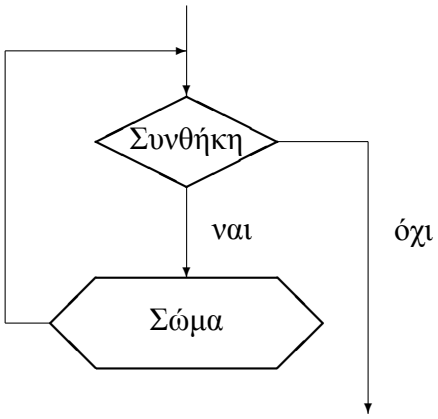
Give n: 12↵
12! = 479001600
```

Ο υπολογισμός γίνεται με το βρόχο “**FOR** (i, 2 **TO** n)”. Η αρχική τιμή της μεταβλητής p είναι 1. Σε κάθε επανάληψη του βρόχου, όμως, η τιμή αυτή πολλαπλασιάζεται με την τρέχουσα τιμή της μεταβλητής ελέγχου i. Επομένως, η **αναλλοίωτη** του βρόχου είναι ότι στο τέλος κάθε επανάληψης θα ισχύει η σχέση $\texttt{p} = \texttt{i}\,! = 1 \times 2 \times \ldots \times i$.

Το λογικό διάγραμμα που αντιστοιχεί στο τμήμα του παραπάνω προγράμματος όπου γίνεται ο υπολογισμός του παραγοντικού φαίνεται στο αριστερό σχήμα. Το λογικό διάγραμμα στο δεξιό σχήμα είναι ισοδύναμο και δείχνει αναλυτικά τη λειτουργία του βρόχου **FOR**: αρχικοποίηση, έλεγχος ορίου, αύξηση της μεταβλητής ελέγχου.





Αξιοπρόσεκτο είναι επίσης αυτό που παρατηρούμε στο παρακάτω σενάριο εκτέλεσης του προγράμματος υπολογισμού του παραγοντικού:

```
Give n: 17↵
17! = -288522240
```

Το αποτέλεσμα που εκτυπώνει το πρόγραμμα είναι προφανώς λάθος: η τιμή του $17!$ δεν μπορεί να είναι αρνητική. Αν κάνουμε τις πράξεις, θα δούμε ότι $17! = 355.687.428.096.000$. Όμως, ο αριθμός αυτός είναι υπερβολικά μεγάλος για να χωρέσει σε μία μεταβλητή τύπου **int** στην Pazcal. Θυμηθείτε ότι ο τύπος αυτός μπορεί να αναπαραστήσει ακέραιους αριθμούς μεταξύ των τιμών INT_MIN και INT_MAX και ότι στους σημερινούς υπολογιστές αρχιτεκτονικής 32 bit είναι συνήθως $\texttt{INT\_MIN} = -2^{31} = -2.147.483.648$ και $\texttt{INT\_MAX} = 2^{31} - 1 = 2.147.483.647$. Το γεγονός αυτό πρέπει να μας κάνει προσεκτικούς: όταν υπολογίζουμε αριθμητικά αποτελέσματα, πρέπει να φροντίζουμε ώστε το αποτέλεσμα κάθε πράξης που εκτελούμε να μπορεί να αναπαρασταθεί από τον τύπο δεδομένων που χρησιμοποιούμε!

Φωλιασμένοι βρόχοι: αρχίζω να βλέπω αστεράκια...

Στα παραδείγματα που ακολουθούν εξηγείται η λειτουργία **φωλιασμένων βρόχων** (nested loops). Στο πρώτο από αυτά βλέπουμε ότι υπάρχει ένα εξωτερικό "`FOR (i, 1 TO 5)`" και ένα εσωτερικό "`FOR (j, 1 TO 10)`". Οι δύο φωλιασμένοι βρόχοι έχουν φυσικά διαφορετικές μεταβλητές ελέγχου (`i` και `j`, εδώ).

```
PROGRAM star_rectangle ()
{
  int i, j;
  FOR (i, 1 TO 5) {
    FOR (j, 1 TO 10) WRITE("*");
    WRITELN();
  }
}
```

```
**********
**********
**********
**********
**********
```

Για κάθε μία επανάληψη του εξωτερικού βρόχου εκτελείται ολόκληρος ο εσωτερικός βρόχος και στη συνέχεια αλλάζουμε γραμμή. Κάθε επανάληψη του εσωτερικού βρόχου εκτυπώνει ένα αστεράκι. Άρα, σωστά βλέπουμε πέντε γραμμές (γιατί τόσες είναι οι επαναλήψεις του εξωτερικού βρόχου) που κάθε μία από αυτές έχει δέκα αστεράκια (γιατί τόσες είναι οι επαναλήψεις του εσωτερικού βρόχου).

Στο παρακάτω πρόγραμμα, αυτό που αλλάζει είναι η περιοχή του εσωτερικού βρόχου, που τώρα γίνεται "`FOR (j, 1 TO 2*i)`". Προσέξτε ότι τώρα το άνω όριο εξαρτάται από την τιμή της μεταβλητής ελέγχου του εξωτερικού βρόχου, άρα δε θα εκτελεστεί το ίδιο πλήθος επαναλήψεων του εσωτερικού βρόχου σε κάθε επανάληψη του εξωτερικού.

```
PROGRAM star_triangle ()
{
  int i, j;
  FOR (i, 1 TO 5) {
    FOR (j, 1 TO 2*i) WRITE("*");
    WRITELN();
  }
}
```

```
**
****
******
********
**********
```

Την πρώτη φορά (`i` $= 1$) θα γίνουν $2 * 1 = 2$ επαναλήψεις και η πρώτη γραμμή θα έχει δύο αστεράκια. Τη δεύτερη φορά (`i` $= 2$) θα γίνουν $2 \times 2 = 4$ επαναλήψεις, άρα τέσσερα αστεράκια, κ.ο.κ. μέχρι την τελευταία φορά (`i` $= 5$) που θα γίνουν $2 \times 5 = 10$ επαναλήψεις και θα εκτυπωθούν δέκα αστεράκια.

Ντόμινο: μια πρώτη γεύση της πολυπλοκότητας...

Μάλλον γνωρίζετε το επιτραπέζιο παιχνίδι **ντόμινο**. Στην παραδοσιακή του μορφή, το ντόμινο παίζεται με 28 ορθογώνια κομμάτια. Κάθε κομμάτι χωρίζεται σε δύο τετράγωνα και σε κάθε τετράγωνο εμφανίζεται ένας αριθμός από τελείες, από το μηδέν έως το έξι. Κάθε συνδυασμός δύο αριθμών εμφανίζεται ακριβώς μία φορά, δηλαδή υπάρχει ένα κομμάτι με τους αριθμούς $(6, 6)$, ένα κομμάτι με τους αριθμούς $(5, 6)$, ένα κομμάτι με τους αριθμούς $(2, 4)$, κ.λπ. Καθώς τα κομμάτια είναι συμμετρικά, δεν έχει σημασία η σειρά με την οποία αναφέρουμε τους αριθμούς ενός κομματιού: το $(2, 4)$ είναι το ίδιο με το $(4, 2)$ και, αυθαίρετα, όταν αναφερόμαστε σε κομμάτια θα προτιμάμε να γράφουμε πρώτα το μικρότερο από τους δύο αριθμούς. Το παραδοσιακό σετ των 28 κομματιών ντόμινο φαίνεται παρακάτω.



Εδώ δε θα ασχοληθούμε με το πώς παίζεται το ντόμινο αλλά μόνο με το να μετρήσουμε πόσα διαφορετικά κομμάτια υπάρχουν. Είπαμε ότι στο παραδοσιακό σετ υπάρχουν 28 κομμάτια. Όμως, δεν υπάρχει κάποιος ιδιαίτερος λόγος οι αριθμοί που γράφονται πάνω στα ντόμινο να σταματάνε στο 6. Ας φανταστούμε ένα σετ κομματιών ντόμινο στο οποίο οι αριθμοί είναι στο διάστημα από 0 έως n, για κάποιο φυσικό αριθμό $n > 0$. Το παρακάτω πρόγραμμα μετράει πόσα διαφορετικά κομμάτια υπάρχουν και, συγχρόνως, τα εμφανίζει στην οθόνη. Δεξιά φαίνεται ένα σχετικά μικρό σενάριο εκτέλεσής του ($n = 3$) και αμέσως μετά μερικά ακόμα σενάρια ($n = 6$, $n = 17$, $n = 42$) απ' όπου έχουν παραλειφθεί οι περισσότερες γραμμές με τα κομμάτια.

```
PROGRAM domino2 ()
{
  int n, count, i, j;

  WRITE("Give n: "); n = READ_INT();
  count = 0;
  FOR (i, 0 TO n)
    FOR (j, i TO n) {
      WRITESPLN(i, j);
      count = count + 1;
    }
  WRITESPLN("Total", count, "pieces.");
}
```

```
Give n: 3↵
0 0
0 1
0 2
0 3
1 1
1 2
1 3
2 2
2 3
3 3
Total 10 pieces.
```

```
Give n: 6↵
0 0
...
6 6
Total 28 pieces.
```

```
Give n: 17↵
0 0
...
17 17
Total 171 pieces.
```

```
Give n: 42↵
0 0
...
42 42
Total 946 pieces.
```

Οι μεταβλητές ελέγχου των δύο φωλιασμένων βρόχων αντιστοιχούν στους δύο αριθμούς που εμφανίζονται πάνω σε κάθε κομμάτι. Η πρώτη μεταβλητή (`i`) κυμαίνεται από 0 μέχρι n. Η δεύτερη όμως (`j`) κυμαίνεται από `i` μέχρι `n`, έτσι ώστε να ισχύει πάντοτε `i` $\leq$ `j`, σύμφωνα με τη σύμβαση που κάναμε ότι γράφουμε πάντα το μικρότερο αριθμό πρώτο.

Ας υποθέσουμε τώρα ότι μας ενδιαφέρει μόνο να μετρήσουμε τα κομμάτια και όχι να τα εμφανίσουμε στην οθόνη. Η προφανής λύση είναι να αφαιρέσουμε τη γραμμή "`WRITESPLN(i, j);`" από το παραπάνω πρόγραμμα. Μπορούμε όμως να χρησιμοποιήσουμε και το παρακάτω, διαφορετικό πρόγραμμα, που μετράει τα κομμάτια χωρίς να τα απαριθμεί τόσο αναλυτικά:

```
PROGRAM domino1 ()
{
  int n, count, i;

  WRITE(”Give n: ”); n = READ_INT();
  count = 0;
  FOR (i, 0 TO n) {
    WRITESPLN(”In”, i + 1, ”piece(s) the largest number is”, i);
    count = count + i + 1;
  }
  WRITESPLN(”Total”, count, ”pieces.”);
}
```

Η λειτουργία του προγράμματος βασίζεται στην ιδέα ότι για κάθε αριθμό i από 0 έως n υπάρχουν ακριβώς $i + 1$ κομμάτια στο σετ των κομματιών μας που έχουν αυτόν τον αριθμό ως μεγαλύτερο (δηλαδή στη δεξιά θέση, σύμφωνα με τη σύμβασή μας). Άρα, αρκεί να αθροίσουμε όλα αυτά τα $i + 1$ για όλα τα i από 0 έως n. Για να δούμε ένα σενάριο εκτέλεσης του προγράμματος:

```
Give n: 6↵
In 1 piece(s) the largest number is 0
In 2 piece(s) the largest number is 1
In 3 piece(s) the largest number is 2
In 4 piece(s) the largest number is 3
In 5 piece(s) the largest number is 4
In 6 piece(s) the largest number is 5
In 7 piece(s) the largest number is 6
Total 28 pieces.
```

Και τώρα είμαστε έτοιμοι για το επόμενο βήμα. Επιστρατεύοντας τα μαθηματικά μας, εύκολα βλέπουμε ότι αυτό που υπολογίσαμε με το προηγούμενο πρόγραμμα δεν είναι τίποτα άλλο από το:

$$\texttt{count} = \sum_{i=0}^{n}(i+1) = \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

Μπορούμε επομένως να γράψουμε το παρακάτω πρόγραμμα που μετράει τα κομμάτια χρησιμοποιώντας απευθείας αυτόν τον τύπο:

```
PROGRAM domino0 ()
{
  int n, count;

  WRITE(”Give n: ”); n = READ_INT();
  count = (n+2)*(n+1)/2;
  WRITESPLN(”Total”, count, ”pieces.”);
}
```

```
Give n: 6↵
Total 28 pieces.

Give n: 17↵
Total 171 pieces.

Give n: 42↵
Total 946 pieces.
```

Έχουμε λοιπόν τρία προγράμματα, τα `domino2`, `domino1`, και `domino0` που κάνουν ακριβώς το ίδιο πράγμα (αν εξαιρέσουμε τα ενδιάμεσα μηνύματα που εκτυπώνουν): υπολογίζουν το πλήθος των κομματιών σε ένα σετ ντόμινο για μία δοθείσα τιμή του n. Τα τρία αυτά προγράμματα υπολογίζουν **την ίδια τιμή** (`count`) αλλά με **τρεις διαφορετικούς τρόπους**:

$$\texttt{count} \;=\; \sum_{i=0}^{n}\sum_{j=i}^{n} 1 \;=\; \sum_{i=0}^{n}(i+1) \;=\; \frac{(n+1)(n+2)}{2}$$

`domino2` `domino1` `domino0`

Έχουμε λοιπόν τρία προγράμματα που κάνουν την ίδια δουλειά. Είναι αυτά ισοδύναμα; Από μία άποψη ναι, είναι ισοδύναμα, αφού υπολογίζουν την ίδια τιμή. Από μία άλλη άποψη όμως, υπάρχουν μεγάλες διαφορές ανάμεσα σε αυτά τα τρία προγράμματα. Για να τη δούμε, ας προσπαθήσουμε να τα εκτελέσουμε για διαφορετικές και σχετικά μεγάλες τιμές του n σε ένα δεδομένο σημερινό Η/Υ και ας μετρήσουμε το χρόνο που κάνουν για να εκτυπώσουν την απάντηση. (Θυμίζουμε ότι για μεγάλες τιμές του n μπορεί η απάντηση να μην είναι σωστή λόγω του περιορισμένου εύρους τιμών που μπορεί να αναπαραστήσει ο τύπος **`int`**, αυτό όμως δεν έχει σχέση με την ταχύτητα υπολογισμού των πράξεων.)

- Για $n = 20,000$, το `domino2` χρειάζεται γύρω στο 1 δευτερόλεπτο, ενώ τα άλλα δύο προγράμματα δίνουν την απάντηση αμέσως.

- Για $n = 200,000$, το `domino2` χρειάζεται γύρω στα 2 λεπτά, το `domino1` λίγα millisecond, ενώ το `domino0` δίνει την απάντηση αμέσως.

- Για $n = 2,000,000$, το `domino2` χρειάζεται γύρω στις 2,5 ώρες, το `domino1` μερικές δεκάδες millisecond, ενώ το `domino0` πάλι δίνει την απάντηση αμέσως.

- Για $n = 200,000,000$, το `domino2` θα χρειαζόταν περίπου 3 χρόνια, το `domino1` χρειάζεται περίπου 2 δευτερόλεπτα, ενώ το `domino0` εξακολουθεί να δίνει την απάντηση αμέσως.

Γιατί συμβαίνουν όλα αυτά; Ας προσπαθήσουμε να μετρήσουμε πόσες πράξεις χρειάζεται να κάνει κάθε πρόγραμμα, ως συνάρτηση του n, και για να απλοποιήσουμε τα πράγματα ας περιοριστούμε στις εμφανείς πράξεις (δηλαδή π.χ. στα σύμβολα `+` και `*` που φαίνονται στο πρόγραμμα), παρότι ο υπολογιστής θα χρειαστεί να κάνει και μη εμφανείς πράξεις (π.χ. για να αυξήσει τις μεταβλητές ελέγχου των βρόχων ή για να ελέγξει αν έχουμε υπερβεί τα όριά τους).

- Το `domino2` εκτελεί μία πρόσθεση για κάθε επανάληψη του εσωτερικού βρόχου. Δεδομένης της τιμής του i, ο εσωτερικός βρόχος επαναλαμβάνεται $n - i + 1$ φορές. Άρα, το συνολικό πλήθος πράξεων θα είναι:

$$\sum_{i=0}^{n}(n-i+1) = \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} = 0.5n^2 + 1.5n + 2$$

  (Φυσικά στο ίδιο συμπέρασμα θα είχαμε καταλήξει αν είχαμε σκεφτεί ότι θα χρειαστούν τόσες προσθέσεις μονάδων ώστε ξεκινώντας από το 0 να καταλήξει η τιμή της μεταβλητής `count` στη σωστή απάντηση.)

- Το `domino1` εκτελεί δύο προσθέσεις για κάθε επανάληψη του βρόχου. Άρα, το συνολικό πλήθος των πράξεων θα είναι:

$$\sum_{i=0}^{n} 2 = 2(n+1) = 2n + 2$$

- Το `domino0` εκτελεί πάντα τρεις πράξεις: δύο προσθέσεις και έναν πολλαπλασιασμό.

Για να συγκρίνουμε λοιπόν το χρόνο εκτέλεσης των τριών προγραμμάτων, αν υποθέσουμε ότι κάθε πράξη χρειάζεται κάποιο συγκεκριμένο χρόνο στον Η/Υ μας, πρέπει να συγκρίνουμε τις τρεις συναρτήσεις:

- $f_2(n) = 0.5n^2 + 1.5n + 2$,
- $f_1(n) = 2n + 2$ και
- $f_0(n) = 3$

για τις (σχετικά μεγάλες) τιμές του n που μας ενδιαφέρουν. Οι διαφορές στο χρόνο εκτέλεσης των τριών προγραμμάτων που παρατηρήσαμε εξηγούνται από το γεγονός ότι το n εμφανίζεται με διαφορετικό εκθέτη στις τρεις συναρτήσεις και, φυσικά, ο όρος με το μεγαλύτερο εκθέτη είναι αυτός που κυριαρχεί όταν το n μεγαλώνει. Συχνά, στη μελέτη της συμπεριφοράς εκτέλεσης των προγραμμάτων, κρατάμε μόνο τον όρο που κυριαρχεί και αγνοούμε τόσο τους υπόλοιπους όρους όσο και το σταθερό πολλαπλασιαστικό του συντελεστή. Λέμε λοιπόν τότε ότι:

- η f_2 είναι $O(n^2)$ ή τετραγωνική ως προς το n,
- η f_1 είναι $O(n)$ ή γραμμική ως προς το n, και
- η f_0 είναι $O(1)$ ή σταθερή ως προς το n.

Με αυτόν τον τρόπο χαρακτηρίζουμε τη **πολυπλοκότητα** (complexity) των τριών προγραμμάτων `domino2`, `domino1` και `domino0`, δηλαδή την ταχύτητα με την οποία υπολογίζουν την απάντηση: το πρώτο είναι $O(n^2)$, το δεύτερο $O(n)$ και το τρίτο $O(1)$. Περισσότερα για την έννοια της πολυπλοκότητας των αλγορίθμων και των προγραμμάτων θα δούμε στο κεφάλαιο 15, μέχρι τότε όμως θα χρησιμοποιήσουμε αρκετές φορές το συμβολισμό $O(\ldots)$ που μόλις μάθαμε.

Είναι λοιπόν ισοδύναμα τα τρία προγράμματα; Από μία άλλη άποψη όχι, δεν είναι, αφού έχουν τελείως διαφορετική πολυπλοκότητα. Είναι πολύ σημαντικό για κάθε υπολογιστικό πρόβλημα που μας δίνεται να μπορούμε να βρίσκουμε όχι οποιοδήποτε πρόγραμμα που να το λύνει, αλλά το πρόγραμμα με τη **μικρότερη δυνατή πολυπλοκότητα** (δηλαδή το γρηγορότερο δυνατό πρόγραμμα). Σε αυτό θα επιμείνουμε πολύ στο μάθημα του 1ου εξαμήνου!



Σχήμα 3.1. Διάγραμμα ροής για το `while` loop.

3.4.2 Εντολή while

Η εντολή **`while`** χρησιμοποιείται όταν το πλήθος των επαναλήψεων σε ένα βρόχο δεν είναι γνωστό πριν την εκτέλεσή του. Το συντακτικό διάγραμμά της είναι το εξής:

while_stmt

Η εκτέλεση του βρόχου **`while`** γίνεται ως εξής (βλ. το διάγραμμα ροής στο σχήμα 3.1):

1. Αποτιμάται η λογική έκφραση.
2. Εάν η λογική έκφραση είναι ψευδής, ο βρόχος **`while`** τερματίζεται και η εκτέλεση περνά στην επόμενη εντολή του προγράμματος.
3. Εάν η συνθήκη είναι αληθής, εκτελείται η εντολή που βρίσκεται στο σώμα του βρόχου.
4. Η εκτέλεση επαναλαμβάνεται από το πρώτο βήμα.

Ας ξεκινήσουμε πάλι με ένα απλό παράδειγμα. Θέλουμε να εκτυπώσουμε τις δυνάμεις του δύο μέχρι το 10.000.000. Θα μπορούσαμε να υπολογίσουμε πόσες είναι, πιθανώς με χαρτί και μολύβι, πριν αρχίσουμε να εκτελούμε το βρόχο και να χρησιμοποιήσουμε το **`FOR`** που είδαμε στην προηγούμενη παράγραφο. Είναι όμως ευκολότερο να αλλάξουμε το `powers_of_two` (σελ. 3.4.1) και να γράψουμε το παρακάτω πρόγραμμα:

```
PROGRAM powers_of_two_2 ()
{
  int i = 0, p = 1;

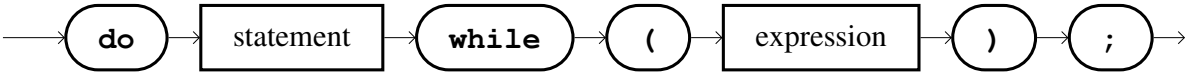
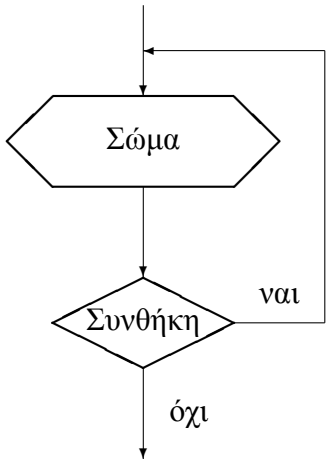
  while (p <= 10000000) {
    WRITELN(2, "^", i, " = ", p);
    p = p * 2; i = i + 1;
  }
}
```

```
2^0 = 2
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
2^11 = 2048
```

```
2^12 = 4096
2^13 = 8192
2^14 = 16384
2^15 = 32768
2^16 = 65536
2^17 = 131072
2^18 = 262144
2^19 = 524288
2^20 = 1048576
2^21 = 2097152
2^22 = 4194304
2^23 = 8388608
```

Τερματισμός: δε σταματούν όλα τα προγράμματα!

Μέχρι τώρα, πριν δούμε την εντολή **while**, για όλα τα προγράμματα που θα μπορούσαμε να γράψουμε είμαστε σίγουροι ότι σταματούν, αργά ή γρήγορα. Αν βάζαμε πολλούς φωλιασμένους βρόχους **FOR** ίσως χρειαζόταν να περιμένουμε πάρα πολύ μέχρι να τερματιστούν (λεπτά, ώρες, μέρες, μήνες, ολόκληρη τη ζωή μας, πολλές χιλιετίες, ίσως και περισσότερο από τη ζωή του σύμπαντος για κάποια προγράμματα), είμαστε σίγουροι όμως ότι κάποτε θα τελείωναν. Χρησιμοποιώντας μόνο την εντολή **FOR** της Pazcal, με την οποία είναι γνωστό εξ αρχής το πλήθος των επαναλήψεων, δεν είναι δυνατό να γράψουμε πρόγραμμα που να μην τερματίζεται.

Με το βρόχο **while** αλλάζουν όλα αυτά. Φανταστείτε το παρακάτω πρόγραμμα:

```
PROGRAM line_punishment ()
{
  while (true) WRITELN("I must not tell lies");
}
```

```
I must not tell lies
I must not tell lies
I must not tell lies
... συνεχίζεται για πάντα
```

Η συνθήκη **true** είναι προφανώς πάντοτε αληθής και άρα το σώμα του βρόχου θα εκτελείται επ' άπειρον. Αυτό θα έχει ως αποτέλεσμα να τυπώνονται συνέχεια γραμμές, μέχρι ο χρήστης να βαρεθεί να τις βλέπει και να πατήσει "`Ctrl+C`" ("`Ctrl+Break`" ή οποιονδήποτε άλλο συνδυασμό πλήκτρων κάνει το λειτουργικό σύστημα του υπολογιστή του να σταματήσει να εκτελεί ένα πρόγραμμα — αυτό θα το συμβολίζουμε στο εξής με Break).

Λιγότερο προφανές αλλά εξίσου άπειρο στην εκτέλεσή του είναι και το παρακάτω πρόγραμμα, που τρέχει για πάντα χωρίς να εμφανίζει τίποτα στην οθόνη.

```
PROGRAM another_infinite_loop ()
{
  int x = 17;
  while (x > 0) x = (x + 42) % 2012;
}
```

x
17
59
101
143
…

Για να βεβαιωθούμε ότι ο βρόχος αυτός δεν τερματίζεται ποτέ, θα χρησιμοποιήσουμε και πάλι την έννοια της **αναλοίωτης** (invariant), δηλαδή μία λογική πρόταση που αληθεύει σε κάθε επανάληψη του βρόχου. Προσέξτε ότι η τιμή της μεταβλητής `x` είναι πάντα θετικός και περιττός ακέραιος. Μπορούμε να βεβαιωθούμε για αυτό το γεγονός επαγωγικά:

- Κατ' αρχήν ισχύει, γιατί η τιμή 17 είναι θετικός περιττός ακέραιος.
- Αν η τιμή του `x` είναι θετικός περιττός ακέραιος, τότε το ίδιο ισχύει και για την τιμή της παράστασης "`(x + 42) % 2012`" που ανατίθεται στο `x` σε κάθε επανάληψη. Για αυτό είμαστε βέβαιοι επειδή: (α) θα είναι σίγουρα μεταξύ 0 και 2011, ως το υπόλοιπο της ακέραιας διαίρεσης ενός θετικού ακέραιου με το 2012, (β) θα είναι σίγουρα περιττός αριθμός, γιατί η τιμή "`x + 42`" είναι περιττός και παίρνουμε το υπόλοιπο της διαίρεσης με το 2012 που είναι άρτιο, και (γ) από τα (α) και (β) προκύπτει ότι δεν μπορεί να είναι μηδέν, άρα το ζητούμενο.

Από τα παραπάνω καταλαβαίνουμε ότι κάποια προγράμματα δεν τερματίζονται και αυτό φαίνεται αμέσως, όπως στο `line_punishment`. Κάποια άλλα δεν τερματίζονται και αυτό φαίνεται δυσκολότερα, όπως στο `another_infinite_loop`. Εν γένει, δεν είναι δυνατόν να αποφασίσουμε με αυτόματο τρόπο αν ένα πρόγραμμα γραμμένο σε μία γλώσσα σαν την Pazcal

τερματίζεται ή όχι, όταν τρέξει με κάποια δεδομένη είσοδο. Το πρόβλημα αυτό είναι γνωστό ως **πρόβλημα τερματισμού** (halting problem) και, όπως είπαμε, αποδεικνύεται ότι γενικά δεν είναι επιλύσιμο — ισοδύναμη ορολογία: δεν είναι αποκρίσιμο (decidable).

Πρώτοι αριθμοί

Οι βρόχοι **while** μπορούν επίσης να είναι **φωλιασμένοι** (nested), δηλαδή να τοποθετηθούν μέσα σε άλλους βρόχους. Στο παράδειγμα που ακολουθεί, ένας βρόχος **while** βρίσκεται φωλιασμένος μέσα σε ένα βρόχο **FOR**.

Το πρόγραμμα εμφανίζει στην οθόνη τους πρώτους αριθμούς μέχρι το 1000. Θυμίζουμε ότι ένας φυσικός αριθμός λέγεται **πρώτος** (prime) αν έχει ακριβώς δύο διαιρέτες (τη μονάδα και τον εαυτό του). Προσέξτε ότι, βάσει αυτού του ορισμού, το 1 δεν είναι πρώτος αριθμός.

```
PROGRAM primes ()
{
  int p, t;
  WRITELN(2);
  FOR (p, 3 TO 1000 STEP 2) {
    t = 3;
    while (p % t != 0) t = t+2;
    if (p == t) WRITELN(p);
  }
}
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
...
```

```
...
911
919
929
937
941
947
953
967
971
977
983
991
997
```

Το πρόγραμμα ξεκινάει εμφανίζοντας τον αριθμό 2 (το μοναδικό άρτιο πρώτο αριθμό) και στη συνέχεια εξετάζει μόνο τους περιττούς αριθμούς — "`FOR(p, 3 TO 1000 STEP 2)`". Για κάθε "υποψήφιο πρώτο" αριθμό `p`, προσπαθεί να τον διαιρέσει με όλους τους περιττούς αριθμούς `t`, ξεκινώντας από τον "υποψήφιο διαιρέτη" $\texttt{t} = 3$, μέχρι να βρει κάποιον `t` που να τον διαιρεί — "`while (p % t != 0)`". Είναι βέβαιο ότι θα βρεθεί τέτοιος `t`: στη χειρότερη περίπτωση ο βρόχος **while** θα σταματήσει όταν γίνει $\texttt{t} = \texttt{p}$. Αν συμβεί αυτό, τότε δε βρέθηκε κανένας διαιρέτης πλην του ίδιου του `p`, άρα ο `p` είναι πρώτος αριθμός και πρέπει να εκτυπωθεί. Διαφορετικά, αν βρέθηκε κάποιος $\texttt{t} < \texttt{p}$ που να διαιρεί το `p`, τότε ο `p` δεν είναι πρώτος αριθμός.

Ο πίνακας κάτω αριστερά δείχνει τις τιμές των μεταβλητών `p` και `t` κατά τη διάρκεια της εκτέλεσης του προγράμματος `primes`. Ένας τέτοιος πίνακας ονομάζεται **trace table**. Όταν εκτελούμε ένα πρόγραμμα **με το χέρι**, κατασκευάζουμε το trace table και γράφουμε τι εμφανίζεται στην οθόνη (output).

Trace Table:

p	t
3	3
5	3
	5
7	3
	5
	7

p	t
9	3
11	3
	5
	7
	9
	11

p	t
13	3
	5
	7
	9
	11
	13

p	t
15	3
17	3
	5
...	...
997	997
999	3

Output:

```
2
3
5
7
11
...
997
```

Μέγιστος κοινός διαιρέτης

Ο **μέγιστος κοινός διαιρέτης** (greatest common divisor, gcd) δύο ακέραιων αριθμών είναι ο μεγαλύτερος ακέραιος που διαιρεί και τους δύο. Έστω λοιπόν ότι δίνονται δύο θετικοί ακέραιοι a και b και αναζητούμε το μέγιστο κοινό διαιρέτη τους.

Μία πρώτη ιδέα είναι να παραγοντοποιήσουμε κάθε έναν αριθμό σε γινόμενο πρώτων παραγόντων (από το Θεώρημα μοναδικής παραγοντοποίησης σε πρώτους, για κάθε θετικό ακέραιο αυτό το γινόμενο είναι μοναδικό) και να εντοπίσουμε τους κοινούς παράγοντες, δηλαδή να ανάγουμε το πρόβλημα σε αυτό της ευρέσεως πρώτων παραγόντων. Δυστυχώς, ο αλγόριθμος αυτός δεν είναι **καθόλου αποδοτικός**: το πρόβλημα της παραγοντοποίησης είναι πολύ δύσκολο και, όπως θα δούμε σε μεγαλύτερο εξάμηνο, δεν γνωρίζουμε καν σήμερα αν υπάρχει αλγόριθμος που να το λύνει σε πολυωνυμικό χρόνο, ως προς το πλήθος των ψηφίων των αριθμών. Ξεχάστε επομένως ό,τι μπορεί να μάθατε στο σχολείο: αν οι αριθμοί είναι πολύ μεγάλοι σίγουρα δε σας συμφέρει να κάνετε παραγοντοποίηση.

Στην συνέχεια, θα προσπαθήσουμε πρώτα να δώσουμε κάποιες αλγοριθμικές λύσεις για το πρόβλημα της εύρεσης του μέγιστου κοινού διαιρέτη, όσο το δυνατόν πιο αποδοτικές, και κατόπιν θα δώσουμε ένα πλήρες πρόγραμμα βασισμένο στην καλύτερη από αυτές τις ιδέες.

Ένας απλός αλγόριθμος που υπολογίζει το μέγιστο κοινό διαιρέτη είναι ο εξής, όπου η συνάρτηση `MIN` επιστρέφει τη μικρότερη από τις παραμέτρους της (παρομοίως υπάρχει και η `MAX`):

```
z = MIN(a, b);
while (a % z != 0 OR b % z != 0) z = z - 1;
WRITELN(z);
```

Η ορθότητα του παραπάνω αλγορίθμου είναι προφανής: Δεν υπάρχει αριθμός $w > z$ τέτοιος ώστε να διαιρεί και τον a και τον b. Η πολυπλοκότητα του αλγορίθμου είναι της τάξης του $\texttt{MIN}(a, b)$ στη χειρότερη περίπτωση και αυτό συμβαίνει όταν οι a, b είναι πρώτοι μεταξύ τους. Ακόμη όμως και στην μέση περίπτωση ο αλγόριθμος δεν αποδίδει καλά: αν ο μικρότερος δε διαιρεί ακριβώς το μεγαλύτερο, τότε χρειάζονται τουλάχιστον $\texttt{MIN}(a, b)/2$ επαναλήψεις.

Ιδέα 1: $\gcd(a, b) = \gcd(a - b, b)$ αν $a > b$
(και αντίστοιχα αν $b > a$).

Χρησιμοποιώντας την ιδιότητα αυτή που είναι γνωστή από τα μαθηματικά, μπορούμε να μικρύνουμε τα ορίσματα (τους αριθμούς των οποίων το μέγιστο κοινό διαιρέτη αναζητούμε). Επαναλαμβάνοντας, θα φτάσουμε σε δύο ίσους αριθμούς, οπότε ο gcd τους θα είναι ο εαυτός τους. Αυτός θα είναι και ο gcd των δύο αρχικών αριθμών.

- Π.χ. $a = 54, b = 16$
  $\gcd(54, 16) = \gcd(38, 16) = \gcd(22, 16) = \gcd(6, 16) = \gcd(6, 10) = \gcd(6, 4) = \gcd(2, 4) = \gcd(2, 2) = 2$

Εκμεταλλευόμενοι αυτή την ιδέα, καταλήγουμε στον εξής αλγόριθμο:

```
while (a > 0 AND b > 0) if (a > b) a = a - b; else b = a - b;
WRITELN(a+b);
```

όπου το a+b στην τελευταία γραμμή είναι ένα τέχνασμα: αφού ένα από τα δύο είναι 0, το άθροισμα είναι ίσο με το άλλο, που είναι ο μέγιστος διαιρέτης.

Όσο για την πολυπλοκότητα, αυτή είναι της τάξης του MAX(a, b) για την χειρότερη περίπτωση, όταν για παράδειγμα $b = 1$. Πάντως, στη μέση περίπτωση ο αλγόριθμος με τις αφαιρέσεις είναι καλύτερος από τον προηγούμενο αλγόριθμο.

Ιδέα 2: Βελτίωση του Ευκλείδη.
$\gcd(a, b) = \gcd(a \bmod b, b)$ αν $a > b$
(και αντίστοιχα αν $b > a$).

Αν αντί για πολλές διαδοχικές αφαιρέσεις χρησιμοποιήσουμε το υπόλοιπο της ακέραιας διαίρεσης $a \bmod b$, τότε η καρδιά του αλγορίθμου γίνεται πολύ πιο αποδοτική. Αυτό βασίζεται στην παρατήρηση ότι, αν το w διαιρεί το a και το b και $a > b$, τότε το w διαιρεί και το $a \bmod b$.

- Π.χ. $a = 54, b = 16$
  $\gcd(54, 16) = \gcd(6, 16) = \gcd(6, 4) = \gcd(2, 4) = \gcd(2, 0) = 2$
- Π.χ. $a = 282, b = 18$
  $\gcd(282, 18) = \gcd(12, 18) = \gcd(12, 6) = \gcd(0, 6) = 6$

Έτσι καταλήγουμε στον **αλγόριθμο του Ευκλείδη** για τον υπολογισμό του μέγιστου κοινού διαιρέτη, που αποτελεί την καρδιά του παρακάτω προγράμματος. Προσέξτε επίσης τη χρήση της προκαθορισμένης συνάρτησης abs που υπολογίζει την απόλυτη τιμή.

```
PROGRAM gcd ()
{
  int a, b;

  WRITE("Give a: "); a = READ_INT();
  WRITE("Give b: "); b = READ_INT();
  WRITE("gcd(", a, ", ", b, ") = ");

  a = abs(a);
  b = abs(b);

  while (a > 0 AND b > 0)
   if (a > b) a = a % b; else b = b % a;
  WRITELN(a+b);
}
```

```
Give a: 42↵
Give b: 24↵
gcd(42, 24) = 6

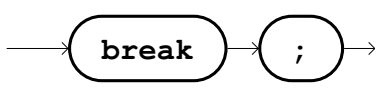
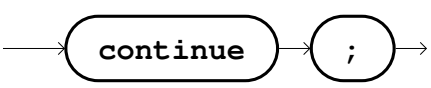
Give a: 14994↵
Give b: 51870↵
gcd(14994, 51870) = 42

Give a: 83742↵
Give b: 17821↵
gcd(83742, 17821) = 1
```

Στη χειρότερη περίπτωση ο αλγόριθμος έχει πολυπλοκότητα της τάξης του $\log(a + b)$. Συνήθως, όμως, τερματίζεται πιο γρήγορα. Μάλιστα, τη χειρότερη απόδοση ο αλγόριθμος του Ευκλείδη την παρουσιάζει αν τα a και b είναι δύο διαδοχικοί όροι της ακολουθίας Fibonacci, που θα δούμε στην επόμενη παράγραφο. Τότε, κατά την εκτέλεση του βρόχου, οι μεταβλητές a και b θα λαμβάνουν διαδοχικά όλες τις μικρότερες τιμές της ακολουθίας Fibonacci:

- Π.χ. $a = F_{10} = 55, b = F_9 = 34$
  $\gcd(55, 34) = \gcd(21, 34) = \gcd(21, 13) = \gcd(8, 13) = \gcd(8, 5) = \gcd(3, 5) =$
  $\gcd(3, 2) = \gcd(1, 2) = \gcd(1, 0) = 1$

3.4.3 Εντολή do...while

Το τρίτο είδος βρόχου της Pazcal είναι ο βρόχος **do**. . .**while**. Μοιάζει με το βρόχο **while**, όμως τώρα η συνθήκη αποτιμάται στο τέλος κάθε επανάληψης και όχι στην αρχή της.

do_while_stmt



Η εκτέλεση του βρόχου **do** . . . **while** γίνεται ως εξής (βλ. το διάγραμμα ροής στο σχήμα 3.2):

1. Εκτελείται η εντολή που βρίσκεται στο σώμα του βρόχου.
2. Αποτιμάται η λογική έκφραση.
3. Εάν η λογική έκφραση είναι αληθής, η εκτέλεση επαναλαμβάνεται από το πρώτο βήμα.
4. Εάν η λογική έκφραση είναι ψευδής, ο βρόχος τερματίζεται και η εκτέλεση περνά στην επόμενη εντολή του προγράμματος.



Σχήμα 3.2. Διάγραμμα ροής για το **do** . . . **while** loop.

Οι εντολές στο σώμα του βρόχου **do**. . .**while** εκτελούνται τουλάχιστον μια φορά. Αυτό συμβαίνει διότι η λογική έκφραση αποτιμάται στο τέλος και όχι στην αρχή του βρόχου. επομένως αυτό το είδος βρόχου είναι ιδανικό αν μία τουλάχιστον επανάληψη είναι επιθυμητή. Θυμίζουμε ότι αντίθετα λειτουργεί η εντολή **while**: Η αποτίμηση της λογικής έκφρασης γίνεται στην αρχή και επομένως μπορεί να γίνουν μηδέν επαναλήψεις.

Αριθμοί Fibonacci

Η ακολουθία των αριθμών Fibonacci είναι μία από τις πιο γνωστές αριθμητικές ακολουθίες:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, \ldots$$

Εξ ορισμού, οι δύο πρώτοι αριθμοί της ακολουθίας είναι το 0 και το 1. Κάθε επόμενος αριθμός είναι το άθροισμα των δύο προηγούμενων:

$$F_0 = 0, \quad F_1 = 1$$
$$F_{n+2} = F_n + F_{n+1}, \quad \forall n \in \mathbb{N}$$

Έστω ένας φυσικός αριθμός $n \in \mathbb{N}$. Ποιος είναι ο μεγαλύτερος αριθμός Fibonacci που δεν υπερβαίνει το n; Δηλαδή, δεδομένου ότι η ακολουθία Fibonacci είναι αύξουσα, πώς μπορούμε να βρούμε έναν όρο F_k της ακολουθίας τέτοιον ώστε $F_k \leq n$ και $F_{k+1} > n$;

Το παρακάτω πρόγραμμα λύνει ακριβώς αυτό το πρόβλημα. Χρησιμοποιεί δύο μεταβλητές, `previous` και `current` για να αποθηκεύσει δύο διαδοχικούς όρους της ακολουθίας Fibonacci (έστω τους F_k και F_{k+1} αντίστοιχα). Στη συνέχεια, υπολογίζει τον επόμενο όρο (που αντιστοιχεί στο F_{k+2}) και τον αποθηκεύει στη μεταβλητή `next`, αθροίζοντας τους δύο προηγούμενους, και μεταφέρει τους δύο τελευταίους όρους που έχει βρει μέχρι τώρα (`next` και `current`) στις μεταβλητές `current` και `previous`, ετοιμάζοντας την επόμενη επανάληψη. Ο βρόχος συνεχίζεται όσο ο όρος στη μεταβλητή `next` δεν υπερβαίνει το n. Όταν αυτό συμβεί, η μεταβλητή `previous` περιέχει την απάντησή μας.

```
PROGRAM fibonacci ()
{
  int n, current, previous, next;

  WRITE("Give n: "); n = READ_INT();
  if (n <= 1) WRITELN(n);
  else {
    previous = 0; current = 1;
    do {
      next = current + previous;
      previous = current; current = next;
    } while (current <= n);
    WRITELN(previous);
  }
}
```

```
Give n: 20↵
13

Give n: 100↵
89

Give n: 987↵
987

Give n: 1234567890↵
1134903170
```

Αθροιστικό πρόγραμμα

Θα αναπτύξουμε ένα πρόγραμμα που αθροίζει όσους αριθμούς θέλει ο χρήστης και στο τέλος παρουσιάζει το άθροισμα. Το πρόγραμμα αυτό θα εκτελείται επ' άπειρον, μέχρι ο χρήστης να βαρεθεί και να το σταματήσει π.χ. πατώντας "Ctrl+C" — Break. Αυτό γίνεται με τον εξωτερικό βρόχο "**do** ... **while (true)**".

Στη μεταβλητή `sum` αποθηκεύεται το άθροισμα των αριθμών που έχουν δοθεί. Αρχικοποιείται στην αρχή κάθε επανάληψης του εξωτερικού βρόχου, έτσι ώστε κάθε άθροιση να ξεκινά από το μηδέν. Στη συνέχεια, ξεκινά ένας φωλιασμένος βρόχος, με τον οποίο διαβάζεται ένας αριθμός τη φορά στη μεταβλητή `number` και αθροίζεται στη `sum`.

Μετά από αυτό διαβάζεται ένας χαρακτήρας στη μεταβλητή `symbol`. Αυτό που θέλουμε είναι να συνεχίζεται η άθροιση όσο ο χρήστης δίνει `'+'` στη θέση του `symbol`. Αντίθετα, να σταματάει και να εκτυπώνει το άθροισμα όταν ο χρήστης δώσει `'='`. Αυτό το πετυχαίνουμε με τη συνθήκη "**while** `(symbol == '+')`" του φωλιασμένου βρόχου. Όμως, είναι πιθανό ο χρήστης να μη δώσει κάποιο από αυτά τα δύο σύμβολα, αλλά κάτι διαφορετικό (π.χ. ένα κενό διάστημα). Στην περίπτωση αυτή θέλουμε να αγνοήσουμε το σύμβολο που έδωσε και να αναζητήσουμε το

επόμενο που θα είναι '+' ή '='. Αυτό επιτυγχάνεται με τον εσωτερικότερο φωλιασμένο βρόχο "**do** symbol = getchar(); **while** (symbol != '+' **AND** symbol != '=');".

```
PROGRAM bigsum ()                                          8+↵
{                                                          9=↵
  int sum, number;                                         17
  char symbol;                                             6+↵
                                                           3+↵
  do {                                                     12+↵
    sum = 0;                                               21=↵
    do {                                                   42
      number = READ_INT(); sum = sum + number;             Break
      do symbol = getchar();
      while (symbol != '+' && symbol != '=');
    } while (symbol == '+');
    WRITELN(sum);
  } while (true);
}
```

Προσέξτε ότι, με εξαίρεση τον εξωτερικότερο βρόχο που εκτελείται επ' άπειρον και θα μπορούσε εξίσου καλά να γραφεί με **while** αντί **do** . . . **while**, οι άλλοι δύο βρόχοι θέλουμε να εκτελούνται τουλάχιστον μία φορά — δεν έχει νόημα να αποτιμηθεί η συνθήκη πριν να διαβαστεί η τιμή της μεταβλητής symbol. Για το λόγο αυτό, προτιμάμε το βρόχο **do** . . . **while** εδώ.

3.4.4 Εντολές break και continue

Πολλές φορές είναι επιθυμητό να σταματήσει ένας βρόχος πρόωρα. Αυτό είναι ιδιαίτερα χρήσιμο για βρόχους που ειδάλλως θα έτρεχαν επ' άπειρον, βρίσκει όμως κι άλλες εφαρμογές. Η εντολή **break** προκαλεί τον άμεσο τερματισμό του βρόχου μέσα στον οποίο βρίσκεται. Αν βρίσκεται μέσα σε περισσότερους (φωλιασμένους) βρόχους, προκαλεί τον τερματισμό του εσωτερικότερου από αυτούς.

Μερικές φορές επίσης είναι χρήσιμο να σταματήσουμε να εκτελούμε το σώμα ενός βρόχου και να προχωρήσουμε κατευθείαν στην επόμενη επανάληψη (αν φυσικά υπάρχει). Αυτό ακριβώς επιτυγχάνεται με την εντολή **continue** που, πάλι, μας οδηγεί στην επόμενη επανάληψη του εσωτερικότερου βρόχου μέσα στον οποίο βρίσκεται, ή στον τερματισμό αυτού αν δεν υπάρχει επόμενη επανάληψη.

break_stmt



continue_stmt



Η ατυχής εικασία...

Ένας φίλος μας που σπουδάζει στο Μαθηματικό ισχυρίζεται ότι για κάθε πρώτο αριθμό p ισχύει $(17p) \bmod 4217 \neq 42$. Μας λέει ότι έχει κάνει τις πράξεις για όλους τους πρώτους

αριθμούς p μέχρι το $10,000$ — κάπου εκεί άρχισε να βλέπει αστεράκια και σταμάτησε — και ότι αληθεύει για όλους. Μας ζητάει να τον βοηθήσουμε, είτε αποδεικνύοντας ότι η εικασία του είναι σωστή, είτε ότι είναι λανθασμένη.

Για να αποδείξουμε ότι η εικασία του είναι σωστή σίγουρα θα χρειαστεί να ξεσκονίσουμε τη θεωρία αριθμών μας (που, ειρήσθω εν παρόδω, μάλλον ποτέ δεν έχουμε διδαχθεί στο σχολείο αν και θα έπρεπε!) οπότε μας φαίνεται λογικό να ξεκινήσουμε προσπαθώντας να αποδείξουμε ότι είναι λανθασμένη. Θα γράψουμε ένα πρόγραμμα που θα προσπαθήσει να βρει **αντιπαράδειγμα** (counterexample), δηλαδή έναν πρώτο αριθμό p τέτοιον ώστε να ισχύει $(17p) \bmod 4217 = 42$. Αν το πρόγραμμά μας βρει τέτοιο αντιπαράδειγμα, τότε έχουμε δείξει ότι η εικασία του φίλου μας είναι λανθασμένη. Αν αντίθετα είμαστε άτυχοι και το αντιπαράδειγμα δε βρεθεί, θα έχουμε μία παραπάνω ένδειξη ότι η εικασία μπορεί πραγματικά να αληθεύει και θα αναγκαστούμε να επιστρατεύσουμε τα μαθηματικά μας για να την αποδείξουμε ή για να την καταρρίψουμε.

Στη σελ. 61 είδαμε ένα πρόγραμμα που βρίσκει τους πρώτους αριθμούς μέχρι το $1,000$. Εύκολα μπορούμε να το προσαρμόσουμε να κάνει τη δουλειά που θέλουμε:

```
PROGRAM prime_conj ()
{
  int p, t;

  FOR (p, 3 TO 1000000 STEP 2) {
    t = 3;
    while (p % t != 0) t = t+2;
    if (p != t) continue;
    if ((17 * p) % 4217 == 42) {
      WRITESPLN("Counterexample:", p); break;
    }
  }
}
```

Τροποποιούμε κατ' αρχήν την εντολή **FOR** ώστε να δοκιμάσει όλους τους αριθμούς p μέχρι το $1,000,000$. Αν το πρόγραμμά μας αποτύχει να βρει αντιπαράδειγμα και θέλουμε να δοκιμάσουμε και με μεγαλύτερους πρώτους αριθμούς, μπορούμε να αυξήσουμε το όριο ή και να αντικαταστήσουμε το βρόχο **FOR** με βρόχο **while**, έτσι ώστε το πρόγραμμα να μη σταματήσει αν δε βρει αντιπαράδειγμα (με κίνδυνο φυσικά να τρέχει για πάρα πολύ καιρό ή και για πάντα).

Προσέξτε την εντολή "`if (p != t) continue;`" στο σημείο που το αρχικό πρόγραμμα της σελ. 61 καταλάβαινε ότι είχε εντοπίσει έναν πρώτο αριθμό και τον εκτύπωνε. Αν η τιμή του t είναι διαφορετική του p αυτό σημαίνει ότι ο αριθμός p δεν είναι πρώτος, άρα δεν έχει νόημα να συνεχίσουμε αυτήν την επανάληψη του **FOR** και μπορούμε να προχωρήσουμε στην επόμενη. Το υπόλοιπο του σώματος (μετά το **continue**) εκτελείται μόνο αν ο αριθμός p είναι πρώτος.

Προσέξτε επίσης την εντολή "`break;`" στο σημείο που το πρόγραμμά μας έχει βρει ένα αντιπαράδειγμα. Ο εξωτερικός βρόχος **FOR** δε χρειάζεται να εκτελεστεί άλλο και η εκτέλεση του προγράμματος μπορεί να τερματιστεί. Επομένως, με την εντολή **break**, αυτό που είχαμε γράψει στην αρχή της παραγράφου 3.4.1, ότι δηλαδή στους βρόχους **FOR** το πλήθος των επαναλήψεων που γίνονται είναι σταθερό και γνωστό τη στιγμή που αρχίζει η εκτέλεση του βρόχου, χρειάζεται να αναθεωρηθεί:

Σε ένα βρόχο **FOR**, το **μέγιστο** πλήθος επαναλήψεων που θα γίνουν είναι σταθερό και γνωστό τη στιγμή που αρχίζει η εκτέλεση του βρόχου. Ο βρόχος είναι δυνατόν να τερματιστεί νωρίτερα με χρήση της εντολής **break**.

Εκτελούμε λοιπόν το πρόγραμμά μας και, μετά από λίγα δευτερόλεπτα, εκτυπώνει:

```
Counterexample: 140443
```

Πραγματικά, βλέπουμε ότι ισχύει

$$17 \times 140,443 = 2,387,531 = 559 \times 4271 + 42$$

Η εικασία του φίλου μας είναι λανθασμένη.

3.4.5 Διαφορές μεταξύ βρόχων

1. **FOR**: Ο (μέγιστος) αριθμός επαναλήψεων είναι γνωστός πριν την εκτέλεση.
2. **while**: Ο (μέγιστος) αριθμός των επαναλήψεων δεν είναι γνωστός από πριν. Ισως να μη γίνει καμμία εκτέλεση του σώματος του βρόχου, γιατί πρώτα ελέγχεται η συνθήκη και μετά εκτελείται το σώμα του βρόχου.
3. **do** . . . **while**: Ο (μέγιστος) αριθμός των επαναλήψεων δεν είναι γνωστός από πριν. Γίνεται όμως τουλάχιστον μια εκτέλεση του σώματος του βρόχου, αφού η συνθήκη ελέγχεται στο τέλος του βρόχου.

3.5 Τα προγράμματα σε C

Παραδείγματα εντολών if, σελ. 45

- ```
  if (amount >= x) amount -= x;
  ```
- ```
  if (amount >= 1000000) printf("Found a millionaire!\n");
  ```
- ```
  if (x*x + y*y == z*z) {
    printf("Found a Pythagorian triple: %d %d %d\n", x, y, z);
    s = (z-x)*(z-y)/2;
    printf("Did you know that %d is a perfect square?\n");
  }
  ```
- ```
  if (a != 0) {
    double d = b*b-4*a*c;
    if (d >= 0) {
      double x = (-b+sqrt(d))/(2*a);
      printf("Found a solution: %d\n", x);
    }
  }
  ```
- ```
  if (year >= 1901 && year <= 2000)
    printf("We're in the 20th century!\n");
  ```
  ```

- ```
  if (grade >= 5) printf("You passed the exam.\n");
  else            printf("I'm sorry, you failed.\n");
  ```
- ```
  if (x % 2 == 0) printf("You have an even number: %d\n", x);
  else            printf("You have an odd number: %d\n", x);
  ```
- ```
  if (ch >= 'A' && ch <= 'Z') letter = ch;
  else                        printf("Not a capital letter\n");
  ```
- ```
  if (x > y)      printf("I win\n");
  else if (x < y) printf("You win\n");
       else       printf("We're tied\n");
  ```
- ```
  if (x > y)  printf("I win\n");
  if (x < y)  printf("You win\n");
  if (x == y) printf("We're tied\n");
  ```
- ```
  if (x>0)
    if (y>0)
      printf("first quadrant\n");
    else if (y<0)
      printf("fourth quadrant\n");
    else
      printf("on the x-axis\n");
  ```

Παραδείγματα εντολών case, σελ. 49

- ```
  switch (month) {
    case 1:  printf("Ιανουάριος\n");    break;
    case 2:  printf("Φεβρουάριος\n");   break;
    ...
    case 12: printf("Δεκέμβριος\n");    break;
    default: printf("άκυρος μήνας!\n"); break;
  }
  ```
- ```
  switch (month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      printf("31 days\n"); break;
    case 4: case 6: case 9: case 11:
      printf("30 days\n"); break;
    case 2:
      printf("28 or 29 days\n"); break;
  }
  ```
- ```
  remaining = 0;
  switch (month) {
    case 1:  remaining += 31;
    case 2:  remaining += 28;
    case 3:  remaining += 31;
    case 4:  remaining += 30;
    case 5:  remaining += 31;
    ...
    case 11: remaining += 30;
    case 12: remaining += 31;
  ```
  ```

```
}
remaining -= day - 1;
```

Παραδείγματα εντολών for, σελ. 50

- Πρόγραμμα counting

```
#include <stdio.h>

int main ()
{
  int i;
  printf("Look: I can count!\n");
  for (i=1; i<=10; i++)
    printf("%d\n", i);
  return 0;
}
```

- Πρόγραμμα powers_of_two

```
#include <stdio.h>

int main ()
{
  int i, p;
  for (p=1, i=0; i<=10; i++, p*=2)
    printf("2^%d = %d\n", i, p);
  return 0;
}
```

- Πρόγραμμα star_rectangle

```
#include <stdio.h>

int main ()
{
  int i, j;
  for (i=1; i<=5; i++) {
    for (j=1; j<=10; j++)
      printf("*");
    printf("\n");
  }
  return 0;
}
```

- Πρόγραμμα star_triangle

```
#include <stdio.h>

int main ()
{
  int i, j;
```

```
  for (i=1; i<=5; i++) {
    for (j=1; j<=2*i; j++)
      printf("*");
    printf();
  }
  return 0;
}
```

- Πρόγραμμα domino2

```
#include <stdio.h>

int main ()
{
  int n, count, i, j;

  printf("Give n: ");
  scanf("%d", &n);

  for (count=0, i=0; i<=n; i++)
    for (j=i; j<=n; j++, count++)
      printf("%d %d\n", i, j);
  printf("Total %d pieces.\n", count);

  return 0;
}
```

- Πρόγραμμα domino1

```
#include <stdio.h>

int main ()
{
  int n, count, i;

  printf("Give n: ");
  scanf("%d", &n);

  for (count=0, i=0; i<=n; i++, count+=i+1)
    printf("In %d piece(s) the largest number is %d\n", i+1, i);
  printf("Total %d pieces.\n", count);

  return 0;
}
```

- Πρόγραμμα domino0

```
#include <stdio.h>

int main ()
{
  int n, count;
```

```
  printf("Give n: ");
  scanf("%d", &n);

  count = (n+2)*(n+1)/2;
  printf("Total %d pieces.\n", count);

  return 0;
}
```

Παραδείγματα εντολών while, σελ. 59

- Πρόγραμμα powers_of_two_2

```
#include <stdio.h>

int main ()
{
  int i, p;
  for (p = 1, i = 0; p <= 10000000, p*=2, i++)
    printf("2^%d = %d\n", i, p);
  return 0;
}
```

- Πρόγραμμα line_punishment

```
#include <stdio.h>

int main ()
{
  for (;;)
    printf("I must not tell lies\n");
  return 0;
}
```

- Πρόγραμμα another_infinite_loop

```
int main ()
{
  int x = 17;
  while (x > 0)
    x = (x + 42) % 2012;
  return 0;
}
```

- Πρόγραμμα primes

```
#include <stdio.h>

int main ()
{
  int p, t;
  printf("2\n");
```

```
  for (p=3; p<=1000; p+=2) {
    t = 3;
    while (p % t != 0)
      t += 2;
    if (p == t)
      printf("%d\n", p);
  }
  return 0;
}
```

- ```
  z = a < b ? a : b;
  while (a % z != 0 || b % z != 0)
    z--;
  printf("%d\n", z);
  ```

- ```
  while (a > 0 && b > 0)
    if (a > b) a-=b; else b-=a;
  printf("%d\n", a+b);
  ```

- Πρόγραμμα gcd

```
#include <stdio.h>

int main ()
{
  int a, b;

  printf("Give a: ");
  scanf("%d", &a);
  printf("Give b: ");
  scanf("%d", &b);
  printf("gcd(%d, %d) = ", a, b);

  if (a < 0) a = -a;
  if (b < 0) b = -b;

  while (a > 0 && b > 0)
    if (a > b) a %= b; else b %= a;
  printf("%d\n", a+b);

  return 0;
}
```

Παραδείγματα εντολών do . . . while, σελ. 65

- Πρόγραμμα fibonacci

```
#include <stdio.h>

int main ()
{
  int n, current, previous, next;
```

```
  printf(”Give n: ”);
  scanf(”%d”, &n);

  if (n <= 1)
    printf(”%d\n”, n);
  else {
    previous = 0;
    current = 1;
    do {
      next = current + previous;
      previous = current;
      current = next;
    } while (current <= n);
    printf(”%d\n”, previous);
  }
  return 0;
}
```

- Πρόγραμμα bigsum

```
#include <stdio.h>

int main ()
{
  int sum, number;
  char symbol;

  for (;;) {
    sum = 0;
    do {
      scanf(”%d”, &number);
      sum += number;
      do
        symbol = getchar();
      while (symbol != ’+’ && symbol != ’=’);
    } while (symbol == ’+’);
    printf(”%d\n”, sum);
  }
  return 0;
}
```

- Πρόγραμμα prime_conj

```
#include <stdio.h>

int main ()
{
  int p, t;

  for (p=3; p<=1000; p+=2) {
    t = 3;
```

```
    while (p % t != 0)
      t += 2;
    if (p != t)
      continue;
    if ((17 * p) % 4271 == 42) {
      printf("Counterexample: %d\n", p);
      break;
    }
  }

  return 0;
}
```

3.6 Τα προγράμματα σε Pascal

Παραδείγματα εντολών if, σελ. 45

- `if amount >= x then amount := amount - x`
- `if amount >= 1000000 then writeln('Found a millionaire!')`
- ```
  if x*x + y*y = z*z then
  begin
    writeln('Found a Pythagorian triple: ', x, ' ', y, ' ', z);
    s := (z-x)*(z-y) div 2;
    writeln('Did you know that ', s, ' is a perfect square?')
  end
  ```
- ```
  { d and x must have been declared before! }
  if a <> 0 then
  begin
    d := b*b-4*a*c;
    if d >= 0 then
    begin
      x = (-b+sqrt(d))/(2*a);
      writeln('Found a solution: ', x)
    end
  end
  ```
- ```
  if (year >= 1901) and (year <= 2000) then
    writeln('We''re in the 20th century!')
  ```
- ```
  if grade >= 5 then writeln('You passed the exam.')
  else               writeln('I''m sorry, you failed.')
  ```
- ```
  if x mod 2 = 0 then writeln('You have an even number: ', x)
  else                writeln('You have an odd number: ', x)
  ```
- ```
  if (ch >= 'A') and (ch <= 'Z') then letter := ch
  else                                writeln('Not a capital letter')
  ```
- ```
  if x > y      then writeln('I win')
  else if x < y then writeln('You win')
       else          writeln('We''re tied')
  ```
  ```

- ```
  if x > y then writeln('I win');
  if x < y then writeln('You win');
  if x = y then writeln('We''re tied')
  ```

- ```
  if x>0 then
    if y>0 then      writeln('first quadrant')
    else if y<0 then writeln('fourth quadrant')
    else             writeln('on the x-axis')
  ```

Παραδείγματα εντολών case, σελ. 49

- ```
  case month of
    1:  writeln('Ιανουάριος');
    2:  writeln('Φεβρουάριος');
    ...
    12: writeln('Δεκέμβριος')
    { There's no default clause in Standard Pascal! }
  }
  ```

- ```
  case month of
    1, 3, 5, 7, 8, 10, 12: writeln('31 days');
    4, 6, 9, 11:           writeln('30 days');
    2:                     writeln('28 or 29 days')
  end
  ```

- ```
  { This cannot be written like this in Pascal!        }
  { Each case clause is separate and there's no NEXT. }
  ```

Παραδείγματα εντολών for, σελ. 50

- Πρόγραμμα counting

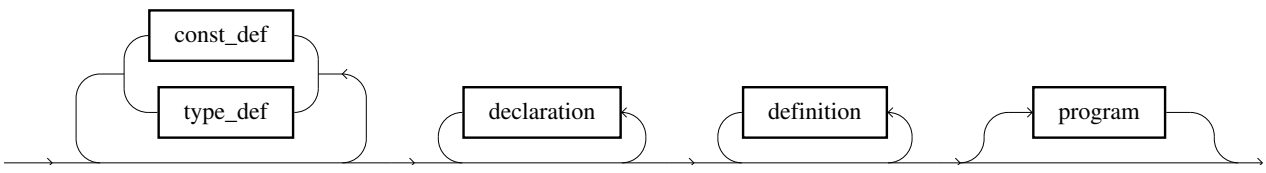
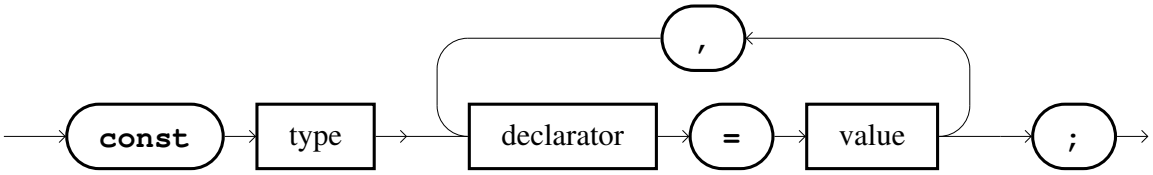
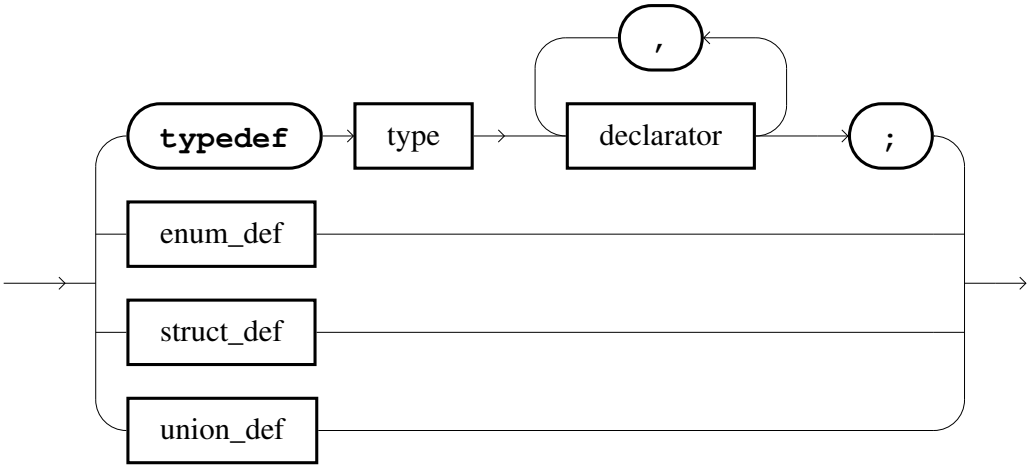
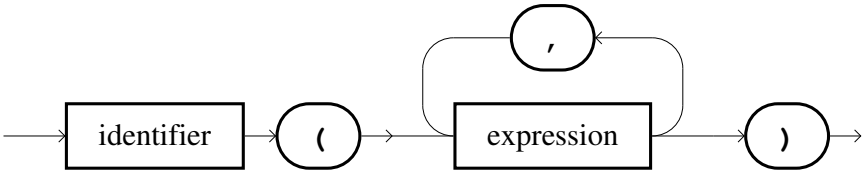
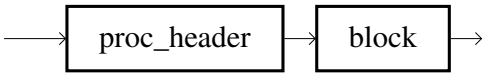
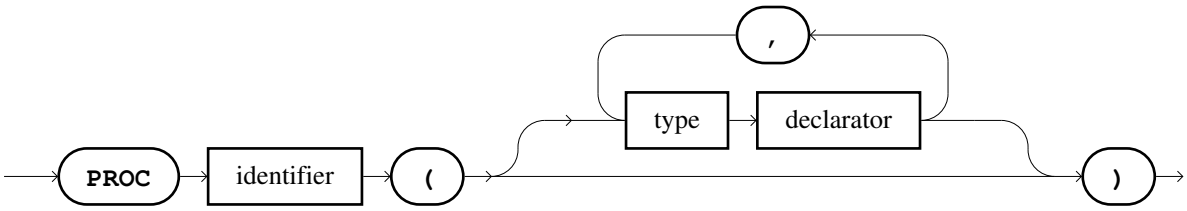
  ```
  program counting (output)
    var i : integer;
  begin
    writeln('Look: I can count!');
    for i := 1 to 10 do writeln(i)
  end.
  ```

- Πρόγραμμα powers_of_two

  ```
  program powers_of_two (output);
    var i, p : integer;
  begin
    p := 1;
    for i := 0 to 10 do
    begin writeln(2, '^', i, ' = ', p); p := p * 2 end
  end.
  ```

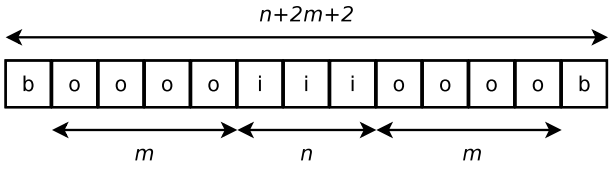
- Πρόγραμμα star_rectangle

  ```
  program star_rectangle (output);
    var i, j : integer;
  ```
  ```

```
begin
  for i := 1 to 5 do
  begin
    for j := 1 to 10 do write('*');
    writeln
  end
end.
```

- Πρόγραμμα star_triangle

```
program star_triangle (output);
  var i, j : integer;
begin
  for i := 1 to 5 do
  begin
    for j := 1 to 2*i do write('*');
    writeln
  end
end.
```

- Πρόγραμμα domino2

```
program domino2 (input, output)
  var n, count, i, j : integer;
begin
  write('Give n: '); read(n);
  count := 0;
  for i := 0 to n do
    for j := 1 to n do
    begin writeln(i, ' ', j); count := count + 1 end;
  writeln('Total ', count, ' pieces.')
end.
```

- Πρόγραμμα domino1

```
program domino1 (input, output);
  var n, count, i : integer;
begin
  write('Give n: '); read(n);
  count := 0;
  for i := 0 to n do
  begin
    writeln('In ', i + 1, ' piece(s) the largest number is ', i);
    count := count + i + 1
  end;
  writeln('Total ', count, ' pieces.')
end.
```

- Πρόγραμμα domino0

```
program domino0 (input, output);
  var n, count : integer;
begin
```

```
  write('Give n: '); read(n);
  count := (n+2)*(n+1) div 2;
  writeln('Total ', count, ' pieces.')
end.
```

Παραδείγματα εντολών while, σελ. 59

- Πρόγραμμα powers_of_two_2

```
program powers_of_two_2 (output);
  var i, p : integer;
begin
  p := 1; i := 0;
  while p <= 10000000 do
  begin writeln(2, '^', i, ' = ', p); p := p * 2; i := i + 1; end
end.
```

- Πρόγραμμα line_punishment

```
program line_punishment (output);
begin
  while true do writeln('I must not tell lies')
end.
```

- Πρόγραμμα another_infinite_loop

```
program another_infinite_loop ()
  var x : integer;
begin
  x := 17;
  while x > 0 do x := (x + 42) mod 2012
end.
```

- Πρόγραμμα primes

```
program primes (output);
  var p, t : integer
begin
  writeln(2); p := 3;
  while p <= 1000 do
  begin
    t := 3;
    while p mod t <> 0 do t := t+2;
    if p = t then writeln(p);
    p := p + 2
  end
end.
```

-
```
if a < b then z := a else z := b;
while (a mod z <> 0) or (b mod z <> 0) do z := z - 1;
writeln(z)
```

-
```
while (a > 0) and (b > 0) do if a > b then a := a - b else b := b - a;
writeln(a+b)
```

- Πρόγραμμα gcd

```
program gcd (input, output);
  var a, b : integer;
begin
  write('Give a: '); read(a);
  write('Give b: '); read(b);
  write('gcd(', a, ', ', b, ') = ');
  a := abs(a); b := abs(b);
  while (a > 0) and (b > 0) do if a > b then a := a mod b; else b := b mod a;
  writeln(a+b)
end.
```

Παραδείγματα εντολών do . . . while, σελ. 65

- Πρόγραμμα fibonacci

```
program fibonacci (input, output)
  var n, current, previous, next : integer;
begin
  write('Give n: '); read(n);

  if n <= 1 then writeln(n)
  else
  begin
    previous := 0; current := 1;
    repeat
      next := current + previous;
      previous := current; current := next;
    until current > n;
    writeln(previous)
  end
end.
```

- Πρόγραμμα bigsum

```
program bigsum (input, output)
  var sum, number : integer;
      symbol : char;
begin
  repeat
    sum := 0;
    repeat
      read(number); sum := sum + number;
      repeat read(symbol) until (symbol = '+') or (symbol = '=');
    until symbol <> '+';
    writeln(sum);
  until false
end.
```

- Πρόγραμμα prime_conj

```
program prime_conj (output);
  var p, t : integer;
      done : boolean;  { there's no break/continue in Pascal! }
begin
  p := 3; done := false;
  while not done and (p <= 1000000) do
  begin
    t := 3;
    while p mod t <> 0 do t := t+2;
    if (p = t) and ((17 * p) mod 4271 = 42) then
    begin writeln('Counterexample: ', p); done := true end;
    p := p+2
  end
end.
```

Κεφάλαιο 4

Προγραμματιστικές τεχνικές

Στο κεφάλαιο αυτό θα αναφερθούμε κατ' αρχήν στα υποπρογράμματα, δηλαδή στις διαδικασίες και τις συναρτήσεις με τις οποίες μπορεί ο προγραμματιστής να δομήσει το πρόγραμμα. Μέσα από τα υποπρογράμματα θα γενικεύσουμε τη δομή του απλού προγράμματος που είδαμε στο Κεφάλαιο 2, θα μιλήσουμε για τις δηλώσεις σταθερών και τύπων, θα γνωρίσουμε τους κανόνες εμβέλειας, τα γενικά και τα τοπικά δεδομένα. Στη συνέχεια θα αναφερθούμε περιληπτικά σε δύο σημαντικές τεχνικές προγραμματισμού: το δομημένο προγραμματισμό (structured programming) και τη βαθμιαία συγκεκριμενοποίηση (stepwise refinement).

4.1 Δομή του προγράμματος, ξανά

Στην αρχή του Κεφαλαίου 2 αναφερθήκαμε στη δομή ενός απλού προγράμματος που αποτελείται από την επικεφαλίδα του, τις τοπικές του δηλώσεις και το σώμα του. Τώρα θα μιλήσουμε για τη δομή μεγαλύτερων προγραμμάτων. Ένα πρόγραμμα Pazcal γενικά αποτελείται από ένα σύνολο **μονάδων κώδικα** (modules), κάθε μία από τις οποίες βρίσκεται σε ένα ξεχωριστό αρχείο προγράμματος. Η δομή ενός module δίνεται από το παρακάτω συντακτικό διάγραμμα.

module



Ένα module αποτελείται κατά σειρά από:

1. δηλώσεις σταθερών και τύπων,
2. δηλώσεις υποπρογραμμάτων, δηλαδή διαδικασιών και συναρτήσεων
3. ορισμούς υποπρογραμμάτων, και

4. τον ορισμό ενός (απλού) προγράμματος.

Όλα τα παραπάνω συστατικά είναι προαιρετικά και μπορούν να παραλείπονται. Το απλό πρόγραμμα, που ορίζεται με τη λέξη κλειδί **PROGRAM** πρέπει να υπάρχει ακριβώς σε ένα από τα modules που αποτελούν το πρόγραμμα: αυτό είναι το αρχικό σημείο της εκτέλεσης.

4.1.1 Σταθερές

Οι δηλώσεις σταθερών μοιάζουν με τις δηλώσεις μεταβλητών, των οποίων προηγείται η λέξη κλειδί **const**. Όμως, πρέπει υποχρεωτικά να συνοδεύονται από αρχικοποίηση. Στη συνέχεια, απαγορεύεται η ανάθεση σε σταθερές: αυτές διατηρούν την τιμή τους καθ' όλη τη διάρκεια του προγράμματος. Η σύνταξη της δήλωσης σταθεράς φαίνεται στο ακόλουθο συντακτικό διάγραμμα.

const_def



Παραδείγματα σταθερών:

```
const int N = 10000;
const REAL pi = 3.1415926535, e = 2.7182818284;
const char space = ' ';
```

Στις παραπάνω δηλώσεις, τα `N`, `pi`, `e` και `space` δεν είναι μεταβλητές. Είναι συνώνυμα των αντίστοιχων σταθερών `10000`, `3.1415926535`, `2.7182818284` και `' '`.

Οι σταθερές μπορούν να χρησιμοποιηθούν οπουδήποτε σε ένα πρόγραμμα, αντί των σταθερών παραστάσεων που εμφανίζονται στο δεξιό μελος του ορισμού τους. Είναι χρήσιμες όταν π.χ. είναι πιθανό οι τιμές τους να αλλάξουν σε επόμενη εκδοχή του προγράμματος.

Υπάρχουν και προκαθορισμένες σταθερές στην Pazcal, όπως οι `INT_MIN` και `INT_MAX`.

4.1.2 Δηλώσεις τύπων

Στις δηλώσεις τύπων συγκαταλέγονται οι δηλώσεις συνωνύμων τύπων, καθώς επίσης και οι δηλώσεις σύνθετων τύπων: απαριθμήσεων (enumerations), δομών (structures) και ενώσεων (unions). Τους σύνθετους τύπους θα τους αφήσουμε για το Κεφάλαιο 13.

Οι δηλώσεις συνωνύμων τύπων μοιάζουν και αυτές με τις δηλώσεις μεταβλητών. Αυτή τη φορά προηγείται η λέξη κλειδί **typedef** και δεν πρέπει να υπάρχει αρχικοποίηση. Η σύνταξη της δήλωσης συνωνύμου τύπου φαίνεται στο ακόλουθο συντακτικό διάγραμμα.

type_def



Παραδείγματα δηλώσεων τύπων:

```
typedef int number;
typedef bool bit;
```

Στις παραπάνω δηλώσεις, τα `number` και `bit` δεν είναι μεταβλητές. Είναι συνώνυμα των τύπων **`int`** και **`bool`**, αντίστοιχα. Μετά από αυτές τις δηλώσεις, μπορεί κανείς να γράψει:

```
number n;
bit b;
```

και να δηλώσει δύο νέες μεταβλητές: τη `n` με τύπο `number` (δηλαδή **`int`**) και την `b` με τύπο `bit` (δηλαδή **`bool`**). Τα συνώνυμα τύπων είναι ιδιαίτερα χρήσιμα σε συνδυασμό με τους σύνθετους τύπους και για την κατασκευή αφηρημένων τύπων δεδομένων (abstract data types).

Υπάρχουν και προκαθορισμένοι τύποι στην Pazcal, όπως είδαμε στο προηγούμενο κεφάλαιο, οι οποίοι είναι δεσμευμένες λέξεις (**`int`**, **`REAL`**, **`char`**, **`bool`**).

4.2 Διαδικασίες

Οι συναρτήσεις και οι διαδικασίες είναι πολύ βασικές στον προγραμματισμό και στη φιλοσοφία σχεδίασης προγραμμάτων με την Pazcal αλλά και με τις περισσότερες γλώσσες προγραμματισμού. Μερικές διαδικασίες και συναρτήσεις βιβλιοθήκης της Pazscal είναι προκαθορισμένες, όπως π.χ. η συνάρτηση `sqrt` που υπολογίζει την τετραγωνική ρίζα ενός αριθμού. Σε αυτό το κεφάλαιο θα καλύψουμε με μεγαλύτερη ακρίβεια τις διαδικασίες και τις συναρτήσεις που ορίζει ο προγραμματιστής.

4.2.1 Πώς δηλώνεται και καλείται μια διαδικασία

Για να καλέσουμε (call, invoke) μια **διαδικασία** (procedure), δηλαδή για να προκαλέσουμε την εκτέλεσή της, απλά χρειάζεται να επικαλεστούμε το όνομά της και να γράψουμε μέσα σε παρενθέσεις τις παραμέτρους (αν υπάρχουν) με τις οποίες θέλουμε να την καλέσουμε.

call



Για να δηλώσουμε μια διαδικασία, χρησιμοποιούμε την δεσμευμένη λέξη **PROC** ακολουθούμενη από το όνομά της και λίστα των **παραμέτρων**. Μετά την επικεφαλίδα της διαδικασίας, ακολουθεί το σώμα της διαδικασίας, που όπως και το σώμα του προγράμματος που είδαμε στο Κεφάλαιο 2, αποτελείται από δηλώσεις τοπικών μεταβλητών και εντολές. Το κύριο πρόγραμμα είναι κατά κάποιο τρόπο η κύρια (main) διαδικασία ενός module, από την οποία ξεκινάει η εκτέλεση.

proc_def



proc_header



Πριν προχωρήσουμε σε ένα παράδειγμα, ας δώσουμε μερικούς χρήσιμους ορισμούς:

- **Εμβέλεια** (scope) μιας μεταβλητής (ή γενικώς ενός ονόματος) είναι το κομμάτι του προγράμματος στο οποίο επιτρέπεται η χρήση της.
- **Τοπικές** (local) μεταβλητές είναι οι μεταβλητές που δηλώνονται μέσα στη διαδικασία και έχουν εμβέλεια μόνο μέσα στο σώμα των διαδικασιών.
- **Γενικές** (οικουμενικές, global) είναι οι μεταβλητές που έξω από τη διαδικασία, στο εξωτερικό επίπεδο του module, και έχουν εμβέλεια σε ολόκληρο το module, δηλαδή σε όλες τις διαδικασίες και στο κυρίως πρόγραμμα.
- Οι παράμετροι στην επικεφαλίδα μίας διαδικασίας λέγονται **τυπικές** (formal), ενώ αυτές που δίνονται στην κλήση της λέγονται **πραγματικές** (actual). Οταν γίνεται η κλήση της διαδικασίας πρέπει οι πραγματικές παράμετροι να αντιστοιχούν μία προς μία στη σειρά και στον τύπο με τις τυπικές παραμέτρους.

Και τώρα ένα απλό παράδειγμα χρήσης διαδικασίας. Το παρακάτω πρόγραμμα εμφανίζει στην οθόνη μία καρτ ποστάλ με ένα δέντρο. Χρησιμοποιεί τη διαδικασία `line` που εμφανίζει την κάρτα μία-μία γραμμή.

```
PROC line (char border, int n, char inside, int m, char outside)
{
  int i;

  WRITE(border);   // αριστερό πλαίσιο
  FOR (i, 1 TO m) WRITE(outside);
  FOR (i, 1 TO n) WRITE(inside);
  FOR (i, 1 TO m) WRITE(outside);
  WRITELN(border); // δεξιό πλαίσιο
}

PROGRAM tree_postcard ()
{
  int i;

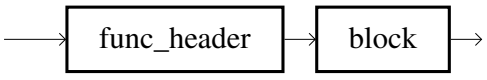
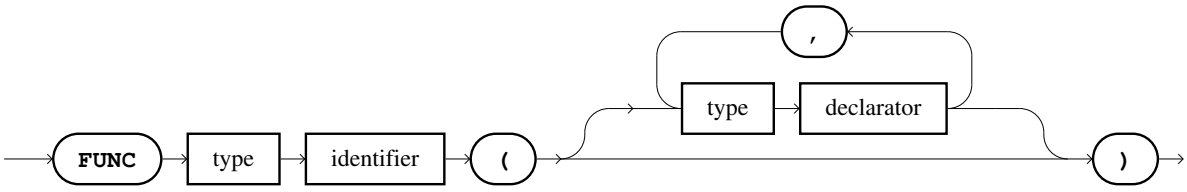
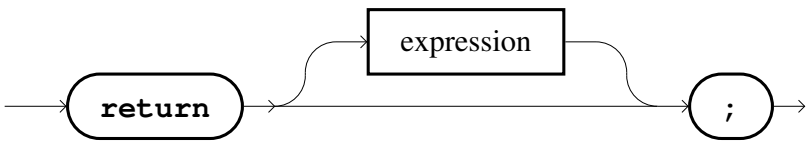
  line('+', 15, '-', 0, ' ');  // πλαίσιο, πρώτη γραμμή
  line('|', 15, ' ', 0, ' ');
  FOR (i, 1 TO 13 STEP 2) line('|', i, '@', (15-i)/2, ' ');
  FOR (i, 1 TO 3) line('|', 3, '#', 6, ' ');
  line('|', 15, ' ', 0, ' ');
  line('+', 15, '-', 0, ' ');  // πλαίσιο, τελευταία γραμμή
}
```

```
+---------------+
|               |
|       @       |
|      @@@      |
|     @@@@@     |
|    @@@@@@@    |
|   @@@@@@@@@   |
|  @@@@@@@@@@@  |
| @@@@@@@@@@@@@ |
|       ###     |
|       ###     |
|       ###     |
|               |
+---------------+
```

Υπάρχουν δύο τρόποι για να διαβάσει και να καταλάβει κανείς αυτό το πρόγραμμα:

- **Top-down**: διαβάζοντας πρώτα το κύριο πρόγραμμα και μετά τη διαδικασία `line`.
- **Bottom-up**: διαβάζοντας πρώτα τη διαδικασία `line` και μετά το κύριο πρόγραμμα.

Προσέξτε ότι, σύμφωνα με τη σύνταξη της Pazcal στην οποία οι διαδικασίες γράφονται πριν το κύριο πρόγραμμα, το top-down αντιστοιχεί σε ανάγνωση του κειμένου αυτού του προγράμματος από κάτω προς τα πάνω και το bottom-up αντιστοιχεί σε ανάγνωση από πάνω προς τα κάτω.

Όλες οι γραμμές από τις οποίες αποτελείται η καρτ ποστάλ παράγονται από τη διαδικασία `line` και είναι της μορφής:



Η διαδικασία `line` έχει πέντε παραμέτρους:

- `border`: ο χαρακτήρας που εμφανίζεται στα δύο άκρα της γραμμής (b), π.χ. `'|'`,

- `n`: το πλήθος των επαναλήψεων του χαρακτήρα που εμφανίζεται στο μέσο της γραμμής.
- `inside`: ο χαρακτήρας που εμφανίζεται n φορές στο μέσο της γραμμής (`i`), π.χ. `'@'`,
- `m`: το πλήθος των επαναλήψεων του χαρακτήρα που εμφανίζεται εκατέρωθεν του μέσου της γραμμής, και
- `outside`: ο χαρακτήρας που εμφανίζεται m φορές εις διπλούν, εκατέρωθεν του μέσου της γραμμής (`o`), π.χ. `' '`.

Προσέξτε ότι το κύριο πρόγραμμα καλεί τη διαδικασία `line` για να εμφανίσει όλες της γραμμές της καρτ ποστάλ και ότι οι παράμετροι κάθε κλήσης διαφέρουν. Σε κάθε κλήση, το συνολικό πλήθος χαρακτήρων κάθε γραμμής $n + 2m + 2$ είναι πάντα ίσο με 17, το επιθυμητό πλάτος της καρτ ποστάλ. Επίσης, προσέξτε ότι τόσο η διαδικασία `line` όσο και το κύριο πρόγραμμα έχουν μία ακέραια τοπική μεταβλητή με όνομα `i`. Οι δύο αυτές μεταβλητές είναι διαφορετικές (δηλαδή αντιστοιχούν σε δύο διαφορετικές θέσεις μνήμης) και δεν πρέπει να τις μπερδεύουμε.

4.2.2 Περισσότερες κλήσεις διαδικασιών

Η εμβέλεια των γενικών μεταβλητών είναι όλο το πρόγραμμα. Η εμβέλεια ενός τοπικού ονόματος (π.χ. μεταβλητής ή παραμέτρου) είναι το block μέσα στο οποίο αυτό δηλώνεται. Όμως, οι εμβέλειες στην Pazcal είναι εν γένει **φωλιασμένες** (nested). Αυτό γίνεται γιατί ένα module (η εμβέλεια που ορίζει τα γενικά ονόματα) μπορεί να περιέχει πολλά block, π.χ. στα σώματα διαδικασιών και του κυρίως προγράμματος, και κάθε block μπορεί να περιέχει άλλα block που εμφανίζονται μέσα του ως σύνθετες εντολές.

Τα προβλήματα αρχίζουν όταν σε μία δήλωση μεταβλητής ή παραμέτρου χρησιμοποιηθεί ένα όνομα που ήδη υπάρχει σε μία εξωτερικότερη εμβέλεια. Αυτό λέγεται **σύγκρουση ονομάτων** (collision of names). Η σύγκρουση ονομάτων επιτρέπεται, είναι χρήσιμη και συμβαίνει συχνά γιατί τα block (και ιδιαίτερα τα σώματα των διαδικασιών και των συναρτήσεων) αντιπροσωπεύουν ανεξάρτητες λειτουργίες και γράφονται ανεξάρτητα το ένα από το άλλο, ή και από διαφορετικούς προγραμματιστές που δεν είναι λογικό να περιμένουμε ότι θα έχουν συνεννοηθεί για να αποφύγουν να δώσουν στις μεταβλητές τους τα ίδια ονόματα.

Όταν υπάρχει σύγκρουση ονόματος, το όνομα που δηλώνεται στο εσωτερικότερο block **κρύβει** το όνομα που δηλώνεται στο εξωτερικότερο block. Με αυτόν τον τρόπο, ανοίγει μία "τρύπα" στην εμβέλεια της εξωτερικής μεταβλητής: στο εσωτερικό της τρύπας, το όνομα παριστάνει τη μεταβλητή που δηλώνεται εσωτερικά και δεν υπάρχει γενικά τρόπος να αναφερθούμε στη μεταβλητή που δηλώνεται εξωτερικά.

Το παρακάτω παράδειγμα χρησιμοποιεί επίτηδες τα ίδια ονόματα για γενικές και τοπικές μεταβλητές, καθώς και για παραμέτρους διαδικασιών. Η εκτέλεση του προγράμματος με το χέρι φαίνεται δεξιά. Προσέξτε ότι κάθε εμβέλεια περιέχει τα ονόματα (παραμέτρων και μεταβλητών) που είναι τοπικά μέσα σε αυτήν. Προσέξτε επίσης ότι διαφορετικά στιγμιότυπα της ίδιας εμβέλειας (π.χ. δυό διαδοχικές κλήσεις της διαδικασίας `p42`) αντιστοιχούν σε δύο διαφορετικά trace tables. Το κύριο πρόγραμμα `proc_example` εδώ δεν έχει κανένα τοπικό όνομα, γι' αυτό δεν υπάρχει αντίστοιχο trace table.

Γενικές μεταβλητές είναι οι `a`, `b` και `c` που ορίζονται στην αρχή του προγράμματος, έξω από τις διαδικασίες και το κύριο πρόγραμμα. Η διαδικασία `p42` έχει παραμέτρους `y` και `b`, που

ορίζονται στην επικεφαλίδα της, και τοπική μεταβλητή c, που ορίζεται στην αρχή του σώματός της. Εδώ έχουμε δύο συγκρούσεις ονομάτων: b και c. Η διαδικασία p17 έχει παραμέτρους a και x, που ορίζονται στην επικεφαλίδα της, και τοπική μεταβλητή b, που ορίζεται στην αρχή του σώματός της. Και εδώ έχουμε δύο συγκρούσεις ονομάτων: a και b.

Προσέξτε ότι μία διαδικασία (ή και το κύριο πρόγραμμα) μπορεί να χρησιμοποιήσει και να αλλάξει την τιμή μιας γενικής μεταβλητής (η οποία προφανώς δεν ξαναδηλώνεται στο εσωτερικό της, γιατί τότε θα ήταν τοπική και όχι γενική). Για παράδειγμα, η ανάθεση `a = a + b;` στο σώμα της διαδικασίας p42 αυξάνει την τιμή της γενικής μεταβλητής a.

```
int a, b, c;

PROC p42 (int y, int b)
{
  int c = 42;

  WRITESPLN(a, b, c, y);
  a = a + b; c = c + 1; b = c + b; y = y-1;
  WRITESPLN(a, b, c, y);
}

PROC p17 (int a, int x)
{
  int b = 17;

  WRITESPLN(a, b, c, x);
  p42(b, x);
  WRITESPLN(a, b, c, x);
}

PROGRAM proc_example ()
{
  a = 1; b = 2; c = 3;
  p17(b, c); p42(c, a);
}
```

Trace tables

Global	a	b	c
	1	2	3
	4		
	8		

p17	a	x	b
	2	3	17

p42	y	b	c
	17	3	42
	16	46	43

p42	y	b	c
	3	4	42
	2	47	43

Output

```
2 17 3 3
1 3 42 17
4 46 43 16
2 17 3 3
4 4 42 3
8 47 43 2
```

- **Τυπικές και πραγματικές παράμετροι**: Οι παράμετροι που γράφονται εντός των παρενθέσεων στην επικεφαλίδα της διαδικασίας (στη δήλωση), ονομάζονται **τυπικές** (formal) παράμετροι, ενώ αυτές που γράφονται στο εσωτερικό των παρενθέσεων στη κλήση του ονομάζονται **πραγματικές** (actual).

- **Παράμετροι και τοπικές μεταβλητές ή γενικές μεταβλητές**: Θα μπορούσε κανείς να σκεφτεί ότι θα ήταν καλή ιδέα αντί να χρησιμοποιεί παραμέτρους και τοπικές μεταβλητές, να κάνει όλες τις μεταβλητές του γενικές, έτσι ώστε να είναι ορατές παντού μέσα στο πρόγραμμα και να κερδίσει ευελιξία. Υπάρχουν λόγοι για να χρησιμοποιούμε παραμέτρους και τοπικές μεταβλητές όταν μπορούμε: Για να προστατεύσουμε κάποια δεδομένα από το να αλλοιωθεί η τιμή τους στο σώμα της διαδικασίας, και ίσως για να μην εξάγει

η διαδικασία περισσότερες πληροφορίες από όσες είναι αναγκαίες. Αυτές είναι σημαντικές αρχές στην κατασκευή λογισμικού γνωστές ως "απόκρυψη πληροφοριών" (δες και αντικειμενοστρεφή προγραμματισμό).

Στο σώμα του κύριου προγράμματος στο παραπάνω παράδειγμα καλείται η διαδικασία `p17` με παραμέτρους τις γενικές μεταβλητές b και `c`. Αυτές, όπως είπαμε, είναι οι **πραγματικές παράμετροι**, αυτές δηλαδή που δίνονται στην κλήση της διαδικασίας. Οι τιμές τους αντιγράφονται στις **τυπικές παραμέτρους** `a` και `x`, αντίστοιχα, αυτές δηλαδή που δηλώνονται στην επικεφαλίδα της διαδικασίας `p17`.

Πρέπει να θυμόμαστε ότι οι τυπικές παράμετροι μίας διαδικασίας και οι πραγματικές παράμετροι που δίνονται σε κάθε κλήση της πρέπει να συμφωνούν στο πλήθος και στον τύπο τους, διαφορετικά υπάρχει type mismatch.

4.3 Συναρτήσεις

Οι συναρτήσεις συντάσσονται με τον ίδιο τρόπο όπως οι διαδικασίες εκτός από την επικεφαλίδα. Εδώ χρησιμοποιούμε την δεσμευμένη λέξη **`FUNC`**, ακολουθούμενη από τον τύπο του αποτελέσματος της συνάρτησης, το όνομά της και λίστα των **παραμέτρων**. Μετά την επικεφαλίδα της συνάρτησης ακολουθεί και πάλι το σώμα της.

func_def



func_header



Επιπλέον στο σώμα της συνάρτησης πρέπει οπωσδήποτε να επιστρέφεται η τιμή του αποτελέσματος. Αυτό γίνεται με την εντολή **`return`**, που προκαλεί την άμεση επιστροφή από μία συνάρτηση. Η εντολή αυτή πρέπει να ακολουθείται από μία παράσταση, η οποία δίνει το αποτέλεσμα της συνάρτησης και ο τύπος της πρέπει να συμφωνεί με τον τύπο του αποτελέσματος της συνάρτησης, όπως δηλώνεται στην επικεφαλίδα.

return_stmt



Προσέξτε ότι η εντολή **return** μπορεί να χρησιμοποιηθεί και χωρίς παράσταση (αποτέλεσμα) για την άμεση επιστροφή από μία διαδικασία ή από το κύριο πρόγραμμα.

Σε αυτό το απλό πρόγραμμα που ακολουθεί, έχουμε μια συναρτήση που βρίσκει και επιστρέφει το μέγιστο κοινό διαιρέτη δύο αριθμών, χρησιμοποιώντας τον αλγόριθμο του Ευκλείδη που είδαμε στο προηγούμενο κεφάλαιο. Η βασική διαφορά μεταξύ μιας συνάρτησης και μιας διαδικασίας είναι ότι η συνάρτηση επιστρέφει μία τιμή ως αποτέλεσμα και η τιμή της συνάρτησης μπορεί να χρησιμοποιηθεί μέσα σε μια παράσταση.

```
FUNC int gcd (int a, int b)
{
  a = abs(a); b = abs(b);
  while (a > 0 AND b > 0)
    if (a > b) a = a % b; else b = b % a;
  return a+b;
}

PROGRAM gcd_func ()
{
  int x, y;

  WRITE(”Give x: ”); x = READ_INT();
  WRITE(”Give y: ”); y = READ_INT();
  WRITELN(”gcd(”, x, ”, ”, y, ”) = ”, gcd(x, y));
}
```

```
Give x: 42↵
Give y: 17↵
gcd(42, 17) = 1
```

4.4 Δομημένος προγραμματισμός

Θα πούμε λίγα λόγια για το **δομημένο προγραμματισμό** (structured programming). Η ιδεολογία του δομημένου προγραμματισμού υποστηρίζει ότι για κάθε ανεξάρτητη συγκεκριμένη λειτουργία πρέπει να γράψουμε μια ανεξάρτητη διαδικασία. Θα πρέπει λοιπόν να αναλύουμε όσο γίνεται το πρόβλημά μας σε ανεξάρτητα υποπροβλήματα και για το καθένα από αυτά να γράφουμε κάποιο υποπρόγραμμα (διαδικασία ή συνάρτηση). Επίσης πρέπει να αποφεύγεται η χρήση παρενεργειών. Αυτό σημαίνει ότι κάθε υποπρόγραμμα πρέπει να κάνει μια συγκεκριμένη λειτουργία, χωρίς να επηρεάζει το κυρίως πρόγραμμα ή άλλα υποπρογράμματα. Η επικοινωνία των επιμέρους μονάδων (πέρασμα τιμών κ.λπ.) γίνεται με χρήση παραμέτρων. Έτσι, το πρόγραμμα γίνεται ευανάγνωστο και είναι εύκολη η μεταφορά και η χρήση των υποπρογραμμάτων σε άλλα προγράμματα .

- Είναι ευκολότερο να γράψουμε ένα δομημένο πρόγραμμα, επειδή πολύπλοκα προβλήματα διασπώνται σε έναν αριθμό μικρότερων, απλούστερων εργασιών.
- Είναι ευκολότερο να ανιχνεύουμε λάθη σε δομημένο πρόγραμμα.
- Ένα σχετικό πλεονέκτημα είναι ο χρόνος που εξοικονομείται με δομημένα προγράμματα. Μπορούν να διορθωθούν ή να τροποποιηθούν πιο εύκολα. Όταν έχουμε κατασκευάσει μια διαδικασία που κάνει κάτι συγκεκριμένο, θα μπορούμε να τη χρησιμοποιούμε σε άλλα προγράμματα δίχως να σπαταλάμε χρόνο σε συγγραφή νέων υποπρογραμμάτων.

4.5 Ανάπτυξη με βαθμιαία συγκεκριμενοποίηση

(Program development by stepwise refinement)

Σε κάθε πρόβλημα πρέπει να καθορίσουμε τα εξής προκειμένου να αναπτύξουμε ένα πρόγραμμα:

- Από πού (πληκτρολόγιο, αρχεία) θα εισαγάγουμε τα δεδομένα και πώς θα τα αποθηκεύσουμε.
- Με ποιον τρόπο θα επεξεργαστούμε τα δεδομένα (Αλγόριθμος του προβλήματος).
- Πού θα παρουσιάσουμε τα αποτελέσματα (οθόνη, αρχεία) και με ποιον τρόπο.

Για τις ανάγκες αυτών των σημειώσεων το κάθε κομμάτι σχεδιασμού ενός προγράμματος είναι σχετικά απλό. Στη σύγχρονη τεχνολογία λογισμικού, όμως, το κάθε μέρος μπορεί να είναι πολύπλοκο, όπως πολύπλοκη μπορεί να είναι και η μεταξύ τους επικοινωνία.

Εισαγωγή και έλεγχος δεδομένων

- Επιλέγουμε τρόπο ανάγνωσης δεδομένων.
- Ελέγχουμε την ορθότητα των δεδομένων, εφόσον έχει προηγηθεί επικοινωνία με τον χρήστη για τις απαιτήσεις του προγράμματος.
- Αν το σύνολο των δεδομένων είναι ορθό προχωρούμε στην περαιτέρω επεξεργασία τους.
- Διαφορετικά τερματίζουμε (halt) την εκτέλεση του προγράμματος ή φιλικά δίνουμε τη δυνατότητα στον χρήστη να ξαναπροσπαθήσει.

Αλγόριθμος — Επεξεργασία δεδομένων

Συνήθως σχεδιάζουμε στο χαρτί μια αλγοριθμική λύση και έπειτα προσπαθούμε να την υλοποιήσουμε με προγραμματιστικές εντολές χρησιμοποιώντας βαθμιαία συγκεκριμενοποίηση.

4.6 Παρουσίαση και συντήρηση

Σε αυτό το κεφάλαιο γνωρίσαμε δυο μεθόδους ανάπτυξης προγραμμάτων: το δομημένο προγραμματισμό και τη βαθμιαία συγκεκριμενοποίηση.

Ωστόσω, ανεξαρτήτως μεθοδολογίας, κάθε προγραμματιστής θα πρέπει να επιδιώκει τα προγράμματά του να συγκεντρώνουν τα παρακάτω ποιοτικά χαρακτηριστικά:

- **Αναγνωσιμότητα** (readability). Ένα πρόγραμμα πρέπει να είναι γραμμένο κατά τέτοιο τρόπο ώστε να είναι εύκολα κατανοητό από άλλους προγραμματιστές, ή και από τον ίδιο τον προγραμματιστή μετά από καιρό, με την ανάγνωσή του ως κειμένου και μόνο με αυτή. Η αναγνωσιμότητα ενός προγράμματος είναι συνάρτηση πολλών παραγόντων, ορισμένοι από τους οποίους είναι οι εξής:

– **Απλότητα** (simplicity). Μεταξύ δυο διαφορετικών τμημάτων κώδικα που επιλύουν το ίδιο πρόβλημα θα πρέπει να επιλέγεται ο απλούστερος. Η επιλογή αυτή ελαχιστοποιεί την πιθανότητα ύπαρξης προγραμματιστικού σφάλματος, η οποία είναι μεγαλύτερη όσο αυξάνει η πολυπλοκότητα του προγράμματος. Φυσικά, η απλότητα θα πρέπει να θυσιάζεται όταν το αντάλλαγμα κρίνεται αξιόλογο, π.χ. μια σημαντική αύξηση στην ταχύτητα εκτέλεσης, μείωση στο μέγεθος του κώδικα, εξοικονόμηση μνήμης, κ.λπ.

– **Επιλογή ονομάτων** (naming). Η σωστή επιλογή ονομάτων για τις μεταβλητές, τις παραμέτρους και τα υποπρογράμματα είναι ιδιαίτερα σημαντική. Τα ονόματα που δίνονται στα αναγνωριστικά πρέπει να αποκαλύπτουν το περιεχόμενο ή τη λειτουργία τους. Για το λόγο αυτό, ο προγραμματιστής δεν πρέπει να περιορίζεται σε λίγους μόνος χαρακτήρες, για να κάνει οικονομία στην πληκτρολόγηση. Στο σχηματισμό μεγάλων ονομάτων μπορούν να χρησιμοποιούνται κατάλληλα χαρακτήρες υπογράμμισης (underscores) ή εναλλαγή πεζών και κεφαλαίων γραμμάτων για τη βελτίωση της αναγνωσιμότητας, π.χ.

```
monthly_income          incomeBeforeTaxes
```

Η χρήση επώνυμων σταθερών επίσης διευκολύνει την αναγνωσιμότητα των προγραμμάτων.

– **Στοίχιση** (indentation). Η σωστή στοίχιση των εντολών διευκολύνει πολύ την κατανόηση του προγράμματος. Για το λόγο αυτό θα πρέπει να χρησιμοποιούνται κενά διαστήματα στο εσωτερικό σύνθετων δομών του προγράμματος, προκειμένου να στοιχίζονται δεξιότερα τα στοιχεία που τις απαρτίζουν. Ο ακριβής αριθμός των κενών διαστημάτων που πρέπει να αφήνει ο προγραμματιστής είναι δική του επιλογή, μια κατάλληλη όμως τιμή είναι από 2 ως 4 διαστήματα.

Η στοίχιση των εντολών μπορεί να γίνει με πολλούς τρόπους, ο τρόπος όμως που θα ακολουθεί ο προγραμματιστής πρέπει να είναι συνεπής στα προγράμματά του. Παρακάτω παραθέτουμε τον τρόπο στοίχισης που χρησιμοποιούμε ως επί το πλείστον σε αυτές τις σημειώσεις:

Πρόγραμμα ή υποπρόγραμμα

```
PROGRAM ...          PROC ...             FUNC ...
{                    {                    {
  δηλώσεις             δηλώσεις             δηλώσεις
  εντολές              εντολές              εντολές
}                    }                    }
```

Απλές εντολές

```
if (...) εντολή      while (...) εντολή   FOR (...) εντολή
else εντολή

do εντολή
while (...);
```

Σύνθετες εντολές

```
if (...) {                  while (...) {              FOR (...) {
  εντολές                     εντολές                    εντολές
}                           }                          }
else {
  εντολές
}

do {                        switch (...) {
  εντολές                     case τιμή_1 : εντολές_1
} while (...);                case τιμή_2 : εντολές_2
                              ...
                              case τιμή_n : εντολές_n
                              default     : εντολές_{n+1}
                            }
```

– *Σχόλια* (comments). Κάθε αυτοτελές τμήμα του προγράμματος πρέπει να συνοδεύεται από σχόλια που εξηγούν τη λειτουργία του. Τα σχόλια πρέπει να αντιμετωπίζονται ως ένα αναπόσπαστο μέρος του προγράμματος, να γράφονται συγχρόνως με το πρόγραμμα και να ενημερώνονται, αν αυτό απαιτείται, με κάθε τροποποίησή του.

Σχόλια είναι απαραίτητο να γράφονται:

* Στην αρχή του προγράμματος και κάθε υποπρογράμματος, εξηγώντας τη λειτουργία αυτού, π.χ. παράμετροι εισόδου, αποτέλεσμα, κ.λπ.
* Στις δηλώσεις σταθερών και μεταβλητών, εξηγώντας το περιεχόμενο αυτών.
* Σε κάθε τμήμα προγράμματος που επιτελεί μια ξεχωριστή και μη προφανή λειτουργία.
* Στα **{**, **}** και **else**, όταν δεν είναι προφανές σε ποιες δομές του προγράμματος αντιστοιχούν.

Πρέπει όμως να σημειωθεί ότι όσο καλοδεχούμενο είναι ένα καλογραμμένο σχόλιο, άλλο τόσο ενοχλητικό είναι ένα κακογραμμένο, λανθασμένο ή παραπλανητικό σχόλιο. Θα πρέπει να αποφεύγονται τα περιττά σχόλια, δηλαδή αυτά που εξηγούν προφανείς λειτουργίες, καθώς και τα κρυπτογραφικά σχόλια, που σημαίνουν κάτι μόνο για τον προγραμματιστή που τα έγραψε.

- **Φιλικότητα προς το χρήστη** (user friendliness). Κατά την εκτέλεση του προγράμματος, ο χρήστης θα πρέπει να καθοδηγείται πλήρως από το πρόγραμμα με κατάλληλα μηνύματα και οδηγίες. Δηλαδή, η χρήση του προγράμματος θα πρέπει να είναι δυνατή χωρίς περαιτέρω προφορικές ή γραπτές οδηγίες.

- **Δόμηση** (structuring). Θα πρέπει να ακολουθούνται οι αρχές του δομημένου προγραμματισμού, δηλαδή:

  – Κάθε ανεξάρτητη λειτουργία να υλοποιείται ως ξεχωριστό υποπρόγραμμα.

  – Να γίνεται λελογισμένη χρήση καθολικών μεταβλητών και παρενεργειών μεταξύ διαφορετικών υποπρογραμμάτων.

  – Να μη γίνονται λογικά άλματα στον κώδικα.

  - Να αποφεύγονται δυσνόητα προγραμματιστικά τεχνάσματα.

- **Τεκμηρίωση** (documentation). Το πρόγραμμα θα πρέπει να συνοδεύεται από κατάλληλες οδηγίες και περιγραφή της λειτουργίας του.
- **Συντήρηση** (maintenance). Το πρόγραμμα θα πρέπει να μπορεί να τροποποιηθεί στο μέλλον, αν διαπιστωθεί η ανάγκη διόρθωσης σφαλμάτων.
- **Ενημέρωση** (updating). Το πρόγραμμα θα πρέπει να μπορεί να τροποποιηθεί στο μέλλον, αν παρουσιαστεί ανάγκη να ανανεωθεί ο κώδικάς του και τα αποθηκευμένα δεδομένα με την εμφάνιση νέων ή διαφορετικών απαιτήσεων.

Κεφάλαιο 5

Πίνακες (arrays)

5.1 Εισαγωγή

Συχνά, κατά την κατασκευή ενός προγράμματος, έχουμε μεταβλητές του ίδιου τύπου που θέλουμε να ομαδοποιήσουμε. Αυτό επιτυγχάνεται με την δομή του πίνακα. **Δομημένη μεταβλητή** (structured variable) λέγεται η μεταβλητή που μπορεί να αποθηκεύει μια συλλογή από τιμές δεδομένων. Κατά αντιπαράθεση με τον απλό τύπο (**`int`**, **`char`**, **`bool`**, **`REAL`**), ο τύπος μιας δομημένης μεταβλητής λέγεται δομημένος τύπος δεδομένων (structured data type). Ενα παράδειγμα είναι ο **πίνακας** (array). Ο πίνακας μπορεί να αποθηκεύει μία συλλογή από τιμές δεδομένων.

Ο πίνακας είναι μια μεταβλητή με δείκτες (indexed variable) που μπορεί να αποθηκεύει πολλά στοιχεία που όμως έχουν όλα τον ίδιο τύπο. Π.χ. αντί να χρησιμοποιούμε 5 μεταβλητές `a0`, `a1`, `a2`, `a3`, `a4` θα μπορούσαμε να δηλώσουμε:

```
int a[5];
```

Τώρα έχουμε ένα δυνατό εργαλείο στα χέρια μας. Μπορούμε να αναφερθούμε σε οποιοδήποτε στοιχείο του πίνακα χρησιμοποιώντας π.χ. `a[3]`, επιπλέον όμως και `a[i]`, `a[i+1]`, `a[i-k]`, κ.λπ.

Η δήλωση πινάκων γίνεται ακριβώς όπως και η δήλωση απλών μεταβλητών, όμως μετά το όνομα του πίνακα τοποθετούμε μέσα σε αγκύλες το πλήθος των στοιχείων του. Προσέξτε ότι η αρίθμηση των μεταβλητών ξεκινάει **από το μηδέν** και όχι από το ένα. Οι επιμέρους μεταβλητές του παραπάνω πίνακα είναι οι `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`. Δεν υπάρχει `a[5]` και είναι λάθος να το χρησιμοποιήσουμε.

Ο τύπος στοιχείων του πίνακα μπορεί να είναι οποιοσδήποτε βασικός τύπος (**`int`**, **`char`**, **`bool`**, **`REAL`**) καθώς και κάποιος άλλος τύπος που έχει ορίσει ο χρήστης ή κάποιος δομημένος τύπος. Μερικά παραδείγματα δήλωσης πινάκων:

```
REAL a[10], x[100];
int b[20];
char c[30];
```

Για να εκχωρήσουμε τιμές στα στοιχεία πίνακα μπορούμε να γράψουμε:

```
a[1] = 4.2;
a[2] = 2.9 + x/2;
a[3] = READ_REAL();
a[9] = a[1];

b[2] = b[2]+1;

c[26] = 't';
```

Παρατηρούμε ότι η εκχώρηση τιμών στα στοιχεία πίνακα γίνεται όπως ακριβώς και στις απλές μεταβλητές.

Για να δούμε, όμως, τώρα πώς εκχωρούμε πολλές τιμές σε ένα πίνακα από το πληκτρολόγιο (διάβασμα πίνακα).

Θεωρούμε ότι έχουμε τον πίνακα `int a[100];`. Το διάβασμα δέκα ακεραίων και η εκχώρησή τους στα δέκα πρώτα στοιχεία του παραπάνω πίνακα γίνεται ως εξής:

```
FOR (i, 0 TO 9) a[i] = READ_INT();
```

Μπορούμε επίσης να χρησιμοποιήσουμε τους εξής εναλλακτικούς τρόπους διαβάσματος του πίνακα:

- Εδώ πρώτα διαβάζεται το μέγεθος του πίνακα (`n`)

  ```
  n = READ_INT();
  FOR (i, 0 TO n-1) a[i] = READ_INT();
  ```

- Εδώ χρησιμοποιείται η ψευδοείσοδος `x` $= 0$ ως συνθήκη τερματισμού (ας σημειωθεί όμως ότι δεν υπάρχει έλεγχος μήπως ο χρήστης δώσει περισσότερα από 100 στοιχεία: σε αυτή την περίπτωση το πρόγραμμα θα τερματιστεί με μήνυμα σφάλματος)

  ```
  x = READ_INT(); i=0;
  while (x != 0) { a[i] = x; i = i+1; x = READ_INT(); }
  ```

5.2 Περισσότερα για τους πίνακες

Συνηθισμένες πράξεις και εντολές:

- `a[k] = a[k]+1;`
  Το περιεχόμενο `a[k]` του πίνακα στην k-οστή θέση αυξάνεται κατά 1.

- `a[k] = a[0]+a[n];`
  Το στοιχείο `a[k]` του πίνακα προκύπτει ως άθροισμα του n-οστού και του πρώτου στοιχείου του πίνακα.

- `FOR (i, 0 TO 99) WRITELN(a[i]);`
  Επαναλαμβάνει την εντολή `WRITELN(a[i])` 100 φορές εμφανίζοντας στην οθόνη τα στοιχεία του πίνακα από `a[0]` ως `a[99]`, ένα σε κάθε γραμμή.

- `FOR (i, 0 TO 99) a[i] = READ_INT();`
  Επαναλαμβάνει την εντολή `a[i] = READ_INT();` 100 φορές γεμίζοντας τον πίνακα από `a[0]` ως `a[99]` με τιμές από το πληκτρολόγιο, μία από κάθε γραμμή.

- `a[k] > a[k+1]`
  Αποτιμάται η έκφραση (expression): "είναι το k-οστό στοιχείο του πίνακα a μεγαλύτερο του επομένου στοιχείου του ίδιου πίνακα;"

- **Αρχικοποίηση (με μηδενικά) ενός πίνακα**

```
FOR (i, 0 TO 9) a[i] = 0;
```

- **Εύρεση ελαχίστου δεδομένου**

```
x = a[0];
FOR (i, 1 TO 9) if (a[i] < x) x = a[i];
```

5.3 Γραμμική αναζήτηση

Παράδειγμα:

Αναζήτηση αριθμού σε μονοδιάστατο πίνακα:

```
PROGRAM linsearch ()
{
  int x, n, a[100];
  // άλλες δηλώσεις
  // τίτλος επικεφαλίδα
  // οδηγίες στο χρήστη
  x = READ_INT();
  // διάβασμα του πίνακα
  // ψάξιμο στον πίνακα για τον x
  // παρουσίαση αποτελεσμάτων
}
```

Μια δυνατή συγκεκριμενοποίηση για "διάβασμα πίνακα, ψάξιμο, παρουσίαση αποτελεσμάτων" είναι και η ακόλουθη:

```
n = READ_INT();
FOR (i, 0 TO n-1) a[i] = READ_INT();

i=0;
while (i < n AND a[i] != x) i=i+1;

if (i < n) WRITESPLN("Το βρήκα στη θέση", i);
else       WRITELN("Δεν το βρήκα");
```

Εναλλακτικοί τρόποι για την αναζήτηση στον πίνακα και παρουσίαση:

- ```
  i = 0;
  do if (a[i] == x) break; else i = i+1;
  while (i < n);

  if (i < n) WRITESPLN(”Το βρήκα στη θέση”, i);
  else       WRITELN(”Δεν το βρήκα”);
  ```

- ```
  i = 0;
  do if (a[i] == x) found = true; else { found = false; i = i+1; }
  while (NOT found AND i < n);

  if (found) WRITESPLN(”Το βρήκα στη θέση”, i);
  else       WRITELN(”Δεν το βρήκα”);
  ```

- ```
  i = 0; found = false;
  do if (a[i] == x) found = true; else i = i+1;
  while (NOT found AND i < n);

  if (found) WRITESPLN(”Το βρήκα στη θέση”, i);
  else       WRITELN(”Δεν το βρήκα”);
  ```

- ```
  i = 0;
  do { found = a[i] == x; i = i+1; }
  while (NOT found AND i < n);

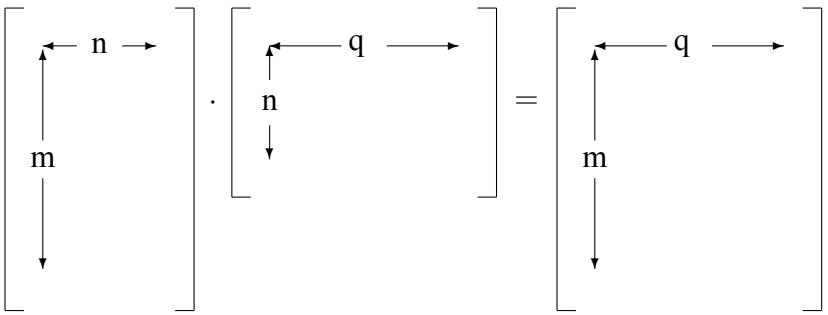
  if (found) WRITESPLN(”Το βρήκα στη θέση”, i-1);
  else       WRITELN(”Δεν το βρήκα”);
  ```

```
PROGRAM linsearch ()
{
  int x, n, a[100];
  int i;

  WRITE(”Δώσε τον αριθμό που θέλεις να βρεις: ”);      x = READ_INT();
  WRITE(”Δώσε το πλήθος των στοιχείων του πίνακα: ”);  n = READ_INT();
  WRITESPLN(”Δώσε τα”, n, ”στοιχεία του πίνακα:”);
  FOR (i, 0 TO n-1) a[i] = READ_INT();

  i=0;
  while (i < n AND a[i] != x) i=i+1;
```

```
  if (i < n) WRITESPLN("Το βρήκα στη θέση", i);
  else       WRITELN("Δεν το βρήκα");
}
```

Η μέθοδος αναζήτησης σε πίνακα που χρησιμοποιήσαμε λέγεται **γραμμική αναζήτηση**, γιατί δυνητικά πρέπει να ελεγχθούν όλα τα στοιχεία του πίνακα (n τον αριθμό) και άρα απαιτούνται δυνητικά $an + b$ βήματα, όπου a, b σταθερές (τάξη μεγέθους n δηλαδή $O(n)$).

5.4 Δυαδική αναζήτηση

Υπάρχει όμως και πιο γρήγορος τρόπος αναζήτησης η **δυαδική αναζήτηση** (binary search) που χρειάζεται δυνητικά $a \log_2 n + b$ υπολογιστικά βήματα, ($O(\log n)$ ή τάξη μεγέθους $\log n$), είναι δηλαδή πολύ πιο γρήγορη μέθοδος για μεγάλα n. Προϋποθέτει όμως ότι ο πίνακας είναι ήδη ταξινομημένος, π.χ. σε αύξουσα διάταξη. Η γενική ιδέα είναι "**διαίρει και κυρίευε**" (divide and conquer): Διαιρούμε τον πίνακα σε δυο μισά, ελέγχουμε (συγκρίνοντας με το μεσαίο στοιχείο) σε ποιο μισό θα πρέπει να είναι ο x και υποδιαιρούμε αυτό το μέρος πάλι σε δυο μισά, κ.ο.κ. Επειδή σε κάθε επανάληψη ο πίνακας μοιράζεται στα δυο και συνεχίζεται η αναζήτηση στο ένα από τα δυο μισά, εάν το αρχικό μήκος είναι n, η αναζήτηση τελειώνει στην χειρότερη περίπτωση (worst case) σε $\log_2 n$ επαναλήψεις (στο εξής θα γράφουμε απλά $\log n$ αντί για $\log_2 n$).

Σημειώνουμε εδώ ότι ο λογάριθμος με βάση 2 ενός αριθμού είναι (χοντρικά) το μήκος της δυαδικής παράστασής του και ότι η διαίρεση ενός δυαδικού δια 2 ισοδυναμεί με αποκοπή του τελευταίου του bit. Ακολουθεί πρόγραμμα δυαδικής αναζήτησης:

```
PROGRAM binsearch ()
{
  int x, n, a[100], i, first, last;

  WRITE("Δώσε τον αριθμό που θέλεις να βρεις: ");      x = READ_INT();
  WRITE("Δώσε το πλήθος των στοιχείων του πίνακα: ");  n = READ_INT();
  WRITESPLN("Δώσε τα", n, "στοιχεία του πίνακα σε αύξουσα σειρά:");
  FOR (i, 0 TO n-1) a[i] = READ_INT();

  // διαίρει και ψάχνε!
  first = 0; last = n-1;
  while (first <= last) {
    mid = (first + last) / 2;
    if (x < a[mid]) last = mid-1;
    else if (x > a[mid]) first = mid+1;
    else break;
  }

  // αποτελέσματα
  if (first <= last) WRITESPLN("Το βρήκα στη θέση", mid);
  else               WRITELN("Δεν το βρήκα");
}
```

Ας δούμε δύο εκτελέσεις αυτού του προγράμματος:

- Έστω ότι έχουμε διαβάσει από το πληκτρολόγιο την παρακάτω διατεταγμένη σειρά `n` $= 10$ ακέραιων αριθμών:

  ```
  3 15 17 21 29 31 37 39 45 55
  ```

  και έστω ότι αναζητούμε τον αριθμό `x` $= 37$. Από τον παρακάτω πίνακα ιχνών (trace table) παρατηρούμε τη διαδοχή τιμών των μεταβλητών `x`, `first`, `last`, `mid`, `a[0]`, `a[1]`, ..., `a[9]`:

x	first	last	mid	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
42	0	9		3	15	17	21	29	31	37	39	45	55
	5		4										
		6	7										
	6		5										
			6										

  και το αποτέλεσμα που εκτυπώνεται είναι:

  ```
  Το βρήκα στη θέση 6
  ```

- Αν αντίθετα αναζητούσαμε τον αριθμό `x` $= 42$, ο αντίστοιχος πίνακας ιχνών θα ήταν:

x	first	last	mid	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
37	0	9		3	15	17	21	29	31	37	39	45	55
	5		4										
	8		7										
		7	8										

  και το αποτέλεσμα θα ήταν:

  ```
  Δεν το βρήκα
  ```

5.5 Πολυδιάστατοι πίνακες

Ένας πίνακας μπορεί να έχει όμως και περισσότερες από μία διαστάσεις. Συνήθως, χρησιμοποιούμε διδιάστατους πίνακες. Ένας διδιάστατος πίνακας ορίζεται βάζοντας δύο ζεύγη αγκυλών στη δήλωσή του. Το πρώτο αντιστοιχεί στο πλήθος των γραμμών ενώ το δεύτερο στο πλήθος των στηλών.

```
int a[10][16];
```

Ορίσαμε δηλαδή ένα πίνακα δύο διαστάσεων, που έχει 10 γραμμές (με δείκτες 0 εως 9) και 16 στήλες (με δείκτες 0 έως 15).

Η χρήση, πρόσβαση (access) γίνεται με `a[i][j]`. (Προσοχή: όχι με `a[i, j]` το οποίο είναι λάθος στην Pazcal και στη C!)

Μπορούμε να διαβάσουμε τα στοιχεία ενός πίνακα από το πληκτρολόγιο ένα προς ένα π.χ. ως εξής:

```
FOR (i, 0 TO 9)
  FOR (j, 0 TO 15) a[i][j] = READ_INT();
```

Με τον παραπάνω τρόπο διαβάζουμε ένα-ένα τα στοιχεία του πίνακα, ξεκινώντας από τα στοιχεία της πρώτης γραμμής, στη συνέχεια διαβάζουμε τα στοιχεία της δεύτερης γραμμής, κ.ο.κ. μέχρι και την δέκατη γραμμή.

Συνηθισμένες πράξεις και εντολές:

- `a[k][m] = a[k][m] + 1;`
- `a[k][k] = a[i][k] + a[n][k];`
- `FOR (i, 0 TO 99) FOR (j, 0 TO 99) a[i][j] = READ_INT();`

  Διαβάζεται ο πίνακας από το πληκτρολόγιο.

- ```
  FOR (i, 0 TO 99) {
    FOR (j, 0 TO 99) WRITE(FORM(a[i][j], 4));
    WRITELN();
  }
  ```

  Εμφανίζεται ο πίνακας στην οθόνη.

5.5.1 Πολλαπλασιασμός πινάκων (matrix multiplication)

Έχουμε ένα πίνακα a, διαστάσεων $m \times n$ και ένα πίνακα b, διαστάσεων $n \times q$. Ο πίνακας c που θα προκύψει από τον πολλαπλασιασμό των δύο αυτών πινάκων θα έχει διαστάσεις $m \times q$. Το στοιχείο c_{ij} του πίνακα δίνεται από τον τύπο:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$


  ```

Ο αντίστοιχος κώδικας που διενεργεί τον πολλαπλασιασμό δίνεται παρακάτω. Προσοχή στο γεγονός ότι η αρίθμηση των γραμμών και των στηλών στην Pazcal ξεκινά από το μηδέν και όχι από το ένα (όπως συνήθως στα μαθηματικά).

```
REAL a[m][n], b[n][q], c[m][q];
...
FOR (i, 0 TO m-1)
  FOR (j, 0 TO q-1) {
    c[i][j] = 0;
    FOR (k, 0 TO n-1) c[i][j] = c[i][j] + a[i][k]*b[k][j];
  }
```

5.5.2 Μαγικά τετράγωνα

10	9	3	22	16
17	11	5	4	23
24	18	12	6	0
1	20	19	13	7
8	2	21	15	14

$n = 5$, άθροισμα = 60

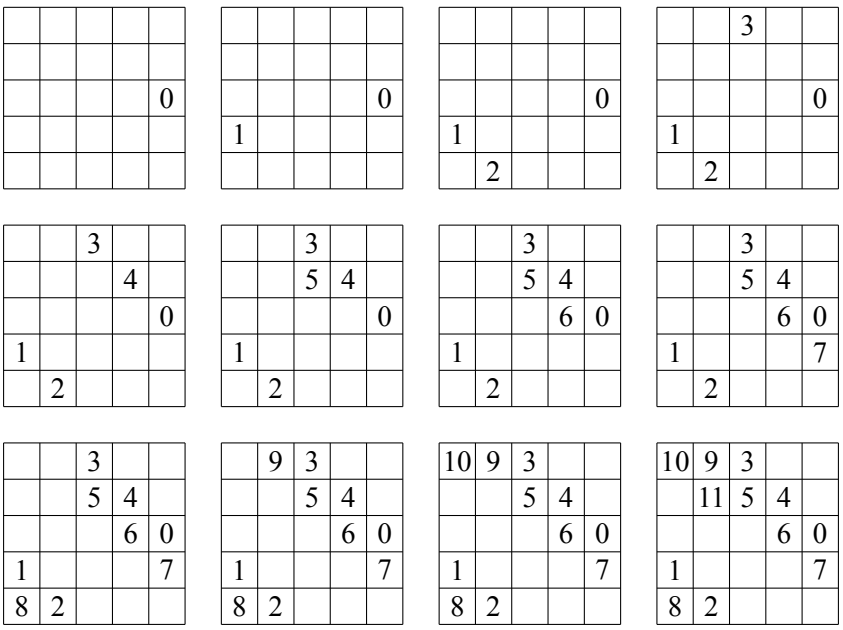
Μαγικό τετράγωνο λέγεται ένας διδιάστατος πίνακας, μεγέθους $n \times n$ που περιέχει όλους τους φυσικούς αριθμούς από το 0 μέχρι και το $n^2 - 1$ και έχει την ιδιότητα το άθροισμα των γραμμών, των στηλών και των διαγωνίων να είναι σταθερό.

Για την κατασκευή μαγικών τετραγώνων με περιττό n υπάρχει ένας απλός αλγόριθμος:

Η τοποθέτηση των αριθμών στον πίνακα γίνεται με αύξουσα σειρά.

Αρχίζουμε, τοποθετώντας τον αριθμό 0 στη μεσαία γραμμή της τελευταίας στήλης. Στη συνέχεια κινούμενοι προς τα κάτω και δεξιά τοποθετούμε τους αριθμούς $1, 2, \ldots n^2 - 1$ στις άδειες θέσεις που συναντάμε. Αν κατά την κίνησή μας βγούμε έξω από το κάτω μέρος του πίνακα τοποθετούμε τον αριθμό στην πρώτη γραμμή της αντίστοιχης στήλης. Ομοίως, αν βγούμε από την δεξιά μεριά του πίνακα τοποθετούμε τον αριθμό στην πρώτη στήλη της αντίστοιχης γραμμής.

Τέλος, αν κάποιο τετράγωνο είναι ήδη κατειλημμένο ή αν βγούμε από την κάτω δεξιά γωνία του πίνακα, ο αριθμός τοποθετείται αριστερά από τον αμέσως προηγούμενο αριθμό.



…

```
PROGRAM magic ()
{
  int a[19][19], i, j, k, h, m, n;

  WRITE(”Give an odd number (1-15), n = ”); n = READ_INT();

  i = n/2; j = n; k = 0;
  FOR (h, 1 TO n) {
    j = j-1; a[i][j] = k; k = k+1;
    FOR (m, 2 TO n) {
      j = (j+1)%n; i = (i+1)%n;
      a[i][j] = k; k = k+1;
    }
  }

  FOR (i, 0 TO n-1) {
    FOR (j, 0 TO n-1) WRITE(FORM(a[i][j], 4));
    WRITELN();
  }
}
```

5.6 Άλλοι τακτικοί τύποι

Απαριθμητοί τύποι

Δηλώνονται από εμάς απαριθμώντας όλες τις δυνατές τιμές. Η διάταξη έχει σημασία.

- ```
  enum fruit_t { appricot, apple, orange, pear, banana };
  enum fruit_t fruit;
  ...
  fruit = apple;
  ...
  if (fruit == banana) ...
  ```

- ```
  enum color    {white, red, blue, green, yellow, black, purple};
  enum sex      {male, female};
  enum day_t    {mon, tues, wednes, thurs, fri, satur, sun};
  enum operator {plus, minus, times, divide};
  enum boolean  {my_false, my_true};
  ```

Κανένα όνομα δεν μπορεί να εμφανιστεί σε δύο δηλώσεις τύπων. Για να δηλώσουμε μία μεταβλητή απαριθμητού τύπου πρέπει να χρησιμοποιήσουμε τη λέξη-κλειδί **enum** ακολουθούμενη από το όνομα του απαριθμητού τύπου, π.χ.

```
enum color c = green;
enum day_t today = monday;
```

Είναι όμως δυνατό να ορίσουμε αποφύγουμε τη χρήση του **enum** ορίζοντας ένα συνώνυμο τύπου, π.χ.

```
typedef enum day_t day;
day d = fri;
```

Κεφάλαιο 6

Αριθμητικοί υπολογισμοί

6.1 Εισαγωγή

Θα ασχοληθούμε με τον χειρισμό πραγματικών τιμών και γενικά με αριθμητικούς υπολογισμούς.

Ο τύπος **REAL** διαφέρει από τους άλλους απλούς τύπους στο ότι δεν είναι τακτικός (ordinal). Οι πραγματικοί αριθμοί (γενικώς) δεν παριστάνονται με ακρίβεια αλλά με κάποια προσεγγιστική τιμή: υπάρχει διαφορά μεταξύ διακριτών (discrete) και συνεχών (continuous) μεγεθών. Για να παραστήσουμε πραγματικούς χρησιμοποιούμε είτε σταθερό σημείο υποδιαστολής (fixed point) ή κινητό σημείο υποδιαστολής (floating point). Στη δεύτερη περίπτωση, η παράσταση αποτελείται από δύο ακέραια μέρη: τον εκθέτη (E, exponent) και τη μάντισα (M, mantissa). Η τιμή του πραγματικού αριθμού είναι $M * 10^E$.

Αν προσθέτουμε (ή γενικά κάνουμε αριθμητικές πράξεις με) αριθμούς που διαφέρουν σημαντικά σε τάξη μεγέθους, συνήθως προκύπτουν αριθμητικά λάθη, π.χ.

$$1000000 + 0.000000001 = 1000000$$

Η συνάρτηση `round(x)` δίνει τον πλησιέστερο ακέραιο (στρογγυλοποίηση), ενώ η `trunc(x)` το ακέραιο μέρος του `x` (αποκοπή). Η συναρτήσεις `floor(x)` και `ceil(x)` επιστρέφουν αντίστοιχα το μεγαλύτερο ακέραιο που δεν υπερβαίνει το `x` και το μικρότερο ακέραιο που δεν υπολείπεται του `x`. Το `floor(x)` διαφέρει από το `trunc(x)` αν ο `x` είναι αρνητικός αριθμός.

6.2 Μέθοδος Newton για εύρεση τετραγωνικής ρίζας

Θα αναπτύξουμε έναν αλγόριθμο για να βρίσκουμε την τετραγωνική ρίζα ενός πραγματικού αριθμού x (χωρίς να χρησιμοποιήσουμε την προκαθορισμένη συνάρτηση `sqrt`).

Έστω y μια προσέγγιση της $\sqrt{x}$ από κάτω (δηλαδή $y \leq \sqrt{x}$). Τότε η $\frac{x}{y}$ θα είναι επίσης προσέγγιση της $\sqrt{x}$ αλλά από πάνω (δηλαδή $\frac{x}{y} \geq \sqrt{x}$). Και αντιστρόφως : Αν $y \geq \sqrt{x}$, τότε $\frac{x}{y} \leq \sqrt{x}$. Για να πάρουμε μια καλύτερη προσέγγιση, σχηματίζουμε τον μέσο όρο αυτών, δηλαδή: $\frac{1}{2}(y + \frac{x}{y})$.

Με αυτή τη βασική ιδέα, ιδού ο αλγόριθμος:

Αρχίζουμε με μια οποιαδήποτε προσέγγιση (έστω π.χ. 1). Παράγουμε μια ακολουθία τιμών y_i (επαναλαμβάνοντας την παραπάνω ιδέα) που προσεγγίζουν τη ρίζα του x: $y_0 = 1$ και $y_{i+1} = \frac{1}{2}(y_i + \frac{x}{y_i})$

Παράδειγμα: $\sqrt{37} = \sim 6.0827625$. Τιμές των y_i: $19 \vdash \sim 10.473 \vdash \sim 7.003 \vdash \sim 6.143 \vdash \sim 6.08306 \vdash \sim 6.08276 \vdash \ldots$

Σημείωση:

Η μέθοδος αυτή είναι ειδική περίπτωση μιας γενικότερης μεθόδου που λέγεται Newton-Raphson και η οποία προσεγγίζει με συγκλίνουσα ακολουθία μια ρίζα μιας εξίσωσης του τύπου $f(x) = 0$.

Υλοποίηση με χρήση συνάρτησης :

```
FUNC REAL sqroot (REAL x)
{
  const REAL epsilon = 0.00001; // 1E-5
  REAL old, new = 1;
  do {
    old = new;
    new = (old + x/old) / 2;
  } while (NOT ( /* συνθήκη τερματισμού */ ));
  return new;
}
```

Τώρα θα συζητήσουμε τί συνθήκη τερματισμού μπορούμε να χρησιμοποιήσουμε.

Εναλλακτικές συνθήκες τερματισμού

1. Με μετρητή (`n=0;` και `n=n+1;` σε κάθε επανάληψη, `while (n <= 20)`). Μειονέκτημα: Ένας απόλυτος αριθμός επαναλήψεων δίνει υπερβολικά ακριβή ή ανεπαρκώς ακριβή προσέγγιση και αυτή εξαρτάται απο το μέγεθος της εισόδου.

2. `new * new == x`: Νοηματικά λανθασμένη συνθήκη. Οι πραγματικοί αριθμοί αποθηκεύονται κατά προσσέγγιση και άρα για πραγματικούς δεν έχει νόημα η συνθήκη $a = b$ (δηλαδή $a - b = 0$) αλλά $a - b < \epsilon$ ή μάλλον $\mid a - b \mid < \epsilon$, άρα:

3. `fabs(new * new - x) < epsilon`. Απόλυτη σύγκλιση. Όμως, πάλι, η ποιότητα της προσσέγγισης εξαρτάται από το μέγεθος της εισόδου.

4. `fabs(new * new - x) / new < epsilon`. Σχετική σύγκλιση. Τώρα καλύτερα. Αλλά για τον υπολογιστή έχει νόημα να διακόψουμε τις επαναλήψεις ακόμη και εάν δεν είμαστε κοντά στη ρίζα, εφόσον δυο διαδοχικές προσσεγγίσεις δεν διαφέρουν σημαντικά.

5. `fabs(new - old) < epsilon`. Απόλυτη σύγκλιση κατά Cauchy.

6. `fabs(new - old) / new < epsilon`. Σχετική σύγκλιση κατά Cauchy.

7. Συνδυασμοί από τα παραπάνω, π.χ. 1 και 6.

6.3 Προκαθορισμένες συναρτήσεις

Συναρτήσεις	Τύπος ορίσματος	Τύπος αποτελέσματος	Σχόλια
`abs`, `fabs` (απόλυτο)	**`int`**, **`REAL`**	ίδιος	
`sin`, `cos`, `exp`, `log`, `sqrt`, `atan` (ημ) (συν) (εκθ) (log) (ρίζα) (τοξεφ)	**`REAL`**	**`REAL`**	σε ακτίνια
`trunc`, `round` `floor`, `ceil`	**`REAL`**	**`REAL`**	

6.4 Υπολογισμός τριγωνομετρικών συναρτήσεων

Παράδειγμα: Υπολογισμός συνημιτόνου με το ανάπτυγμα Taylor

$$\cos(x) = \sum_{i=0}^{\infty}(-1)^i \frac{x^{2i}}{(2i)!}$$

για $i = i + 1$: $\quad (-1)^{i+1}\frac{x^{2i+2}}{(2i+2)!} = -[(-1)^i \frac{x^{2i}}{(2i)!}]\frac{x^2}{(2i+1)(2i+2)}$

συνεπώς: $n = n + 2$; $\quad \textit{newterm} = -\textit{oldterm} * \frac{x^2}{n*(n+1)}$

```
FUNC REAL mycos (REAL x)
{
  const REAL epsilon = 1E-5;
  REAL sqx = x * x, term = 1, sum = 1;
  int n = 1;
  do {
    n = n + 2;
    term = -term * sqx / (n*(n+1));
    sum = sum + term;
  } while (abs(term/sum) >= epsilon);
  return sum;
}
```

Κεφάλαιο 7

Αναδρομή (recursion)

7.1 Εισαγωγή

Αρκετά προβλήματα λύνονται εύκολα με τη βοήθεια αναδρομικών διαδικασιών ή αναδρομικών συναρτήσεων.

Αναδρομικές διαδικασίες ή συναρτήσεις αυτές που καλούν τον εαυτό τους μια ή και περισσότερες φορές προκειμένου να λύσουν σχετικά υποπροβλήματα.

Στον ορισμό της αναδρομικής διαδικασίας διαπιστώνουμε ότι το αρχικό πρόβλημα διασπάται σε μικρότερα προβλήματα του ίδιου τύπου με το αρχικό.

Η μέθοδος της αναδρομής, για παράδειγμα, μπορεί να χρησιμοποιηθεί για την εύρεση όλων των παραγόντων ενός αριθμού. Το απλούστερο παράδειγμα είναι ο υπολογισμός του (παραγοντικού του n) $n!$:

$$n! = n * (n-1) * (n-2) * \cdots * 2 * 1$$

Αναδρομικός ορισμός:

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n!, \qquad \forall n \in \mathbb{N} \end{aligned}$$

7.2 Υπολογισμός παραγοντικού

Η συνάρτηση που χρησιμοποιούμε είναι αναδρομική, αφού καλεί τον εαυτό της (με μικρότερο όρισμα).

```
FUNC int fact (int n)
{
  if (n==0) return 1; else return fact(n-1) * n;
}
```

κλήση: `if (n>=0) result = fact(n);`

Βλέπουμε ότι στο σώμα (εντολές) της συνάρτησης `fact` υπάρχει κλήση αυτής της ίδιας της συνάρτησης `fact` (για μικρότερη όμως παράμετρο). Οταν η εκτέλεση του σώματος φθάσει

στο σημείο αυτό, τότε δραστηριοποιείται νέο αντίγραφο της συνάρτησης (ή διαδικασίας αντιστοίχως). Δηλαδή κατά την εκτέλεση πρέπει ο επεξεργαστής να θυμάται, για την επιστροφή, όχι μόνο σε ποιο σημείο έγινε η κλήση της διαδικασίας (ή συνάρτησης), αλλά και από ποιο αντίγραφο (π.χ.) αυτής. Δημιουργείται έτσι μια στοίβα (stack) από αντίγραφα της διαδικασίας.

Παράδειγμα:

Έστω ότι θέλουμε να υπολογίσουμε το παραγοντικό του ακεραίου 3, τότε:

Το κυρίως πρόγραμμα καλεί fact(3)	συνεχίζει υπολογισμούς.
↓	↑
1. fact καλεί fact(2)	συνεχίζει τον υπολογισμό, επιστρέφει 6
↓	↑
2. fact καλεί fact(1)	συνεχίζει τον υπολογισμό, επιστρέφει 2
↓	↑
3. fact καλεί fact(0)	συνεχίζει τον υπολογισμό, επιστρέφει 1
↓	↑
4. fact υπολογίζει fact(0) $\longrightarrow$	επιστρέφει 1 $\rightarrow$

Δεν αρκεί αναδρομική λύση, πρέπει να υπάρχει και έλεγχος τερματισμού. Μια διαδικασία που καλεί τον εαυτό της επ' άπειρον είναι (συνήθως) άχρηστη.

7.3 Αριθμοί Fibonacci

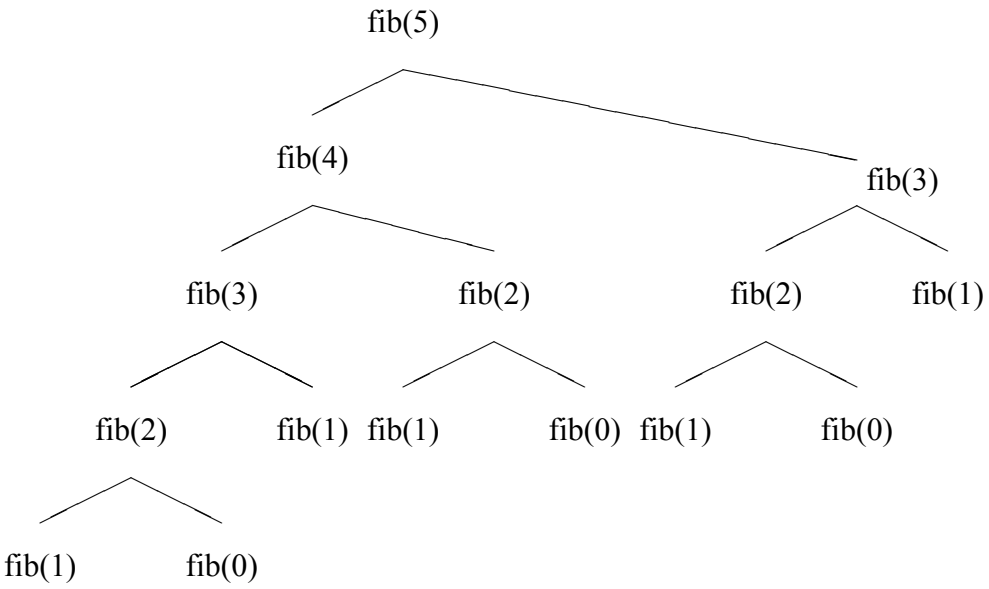
Οι αριθμοί Fibonacci ορίζονται αναδρομικά ως εξής:

$$F_0 = 1, \quad F_1 = 1$$
$$F_{n+2} = F_n + F_{n+1}, \quad \forall n \in \mathbb{N}$$

```
FUNC int fib (int n)
{
  if (n==0 OR n==1) return n; else return fib(n-1) + fib(n-2);
}
```

κλήση: `if (n>=0) result = fib(n);`

Αυτό είναι παράδειγμα μη αποδοτικής χρήσης της αναδρομής. Παρακάτω φαίνεται το δέντρο κλήσεων της συνάρτησης για τον υπολογισμό του `fib(5)`. Πόσες φορές υπολογίζεται η τιμή της `fib(1)`;



7.4 Μέγιστος κοινός διαιρέτης

Αναδρομική υλοποποίηση του Ευκλείδιου Αλγορίθμου:

```
FUNC int gcd (int i, int j)
{
  if (i==0 OR j==0)  return i+j;
  else if (i > j)    return gcd(i%j, j);
       else          return gcd(i, j%i);
}
```

7.5 Μια συνάρτηση παρόμοια με αυτήν του Ackermann

Ας δούμε μια συνάρτηση που ορίζεται με πιο περίπλοκη αναδρομή:

$$z(i,j,0) = j+1, \quad z(i,0,1) = i, \quad z(i,0,2) = 0, \quad z(i,0,n+3) = 1,$$
$$z(i,j+1,n+1) = z(i,z(i,j,n+1),n), \qquad \forall i,j,n \in \mathbb{N}$$

```
FUNC int z (int i, int j, int n)
{
  if (n==0) return j+1;
  else if (j==0)
    if (n==1)      return i;
    else if (n==2) return 0;
         else      return 1;
  else return z(i, z(i,j-1,n), n-1);
}
```

κλήση: `if (i>=0 AND j>=0 AND n>=0) result = z(i,j,n);`

Προσπαθήστε να υπολογίσετε `z(2,3,4)`.

7.6 Αμοιβαία (ταυτόχρονη) αναδρομή

Κανονικά ένα υποπρόγραμμα επιτρέπεται να κληθεί εάν έχει ήδη δηλωθεί. Ωστόσο, υπάρχουν περιπτώσεις που δύο υποπρογράμματα αλληλοκαλούνται: αυτό λέγεται αμοιβαία αναδρομή. Στην Pazcal, η αμοιβαία αναδρομή υποστηρίζεται γράφοντας την κεφαλή του δευτέρου υποπρογράμματος (χωρίς το σώμα του) πριν από το πρώτο υποπρόγραμμα. Η ίδια κεφαλή προφανώς επαναλαμβάνεται μετά, όταν ορίζεται και το σώμα του δευτέρου υποπρογράμματος. αλλά και μετά, μαζί με το σώμα του.

```
FUNC int f2 (int n);     // function prototype

FUNC int f1 (int n)
{
  if (n==0) return 5;
  else      return f1(n-1) * f2(n-1);
}

FUNC int f2 (int n)
{
  if (n==0) return 3;
  else      return f1(n-1) + 2*f2(n-1);
}
```

Υπολογίστε το `f2(3)`.

Κεφάλαιο 8

Ορθότητα

8.1 Εισαγωγή

Όπως έχουμε ήδη εξηγήσει, διακρίνουμε τρία επίπεδα ορθότητας:

- **Συντακτική ορθότητα.**
- **Νοηματική ορθότητα.**
- **Σημασιολογική ορθότητα.**

Στα προγράμματα Pazcal ο compiler ελέγχει τα δυο πρώτα επίπεδα ορθότητας. π.χ.

- Δεν μας επιτρέπει να παραλείψουμε το όνομα του προγράμματος ή την τελεία στο τέλος του προγράμματος. (συντακτικά λάθη)
- Δεν μας επιτρέπει να προσθέσουμε σ' έναν αριθμό ένα γράμμα του αλφαβήτου. (νοηματικά λάθη)

Εδώ θα συζητήσουμε περισσότερο για τη σημασιολογική ορθότητα, που φυσικά ελέγχεται σχετικά με κάποιες προδιαγραφές. Το τρίτο επίπεδο ορθότητας ελέγχεται με δοκιμές (testing) ή με επαλήθευση (verification).

Θα συζητήσουμε την επαλήθευση ορθότητας υποπρογραμμάτων, οι μέθοδοι, όμως, εφαρμόζονται γενικά και σε ολόκληρα προγράμματα.

Γενικώς, επιδιώκουμε ένα πρόγραμμα να έχει απλή δομή, να αποτελείται από δομικά στοιχεία (building blocks), υποπρογράμματα επαναχρησιμοποιούμενα (modules) για τα οποία έχει γίνει ήδη αυστηρή επαλήθευση ορθότητας.

Πώς μπορούμε να αποδείξουμε ότι μια διαδικασία η οποία σχεδιάστηκε για μια λειτουργία πράγματι την υλοποιεί;

Παράδειγμα:

Να επαληθεύσετε ότι το παρακάτω υποπρόγραμμα υπολογίζει το γινόμενο δυο φυσικών αριθμών.

```
FUNC int mult (int x, int y)
{
  int i, z = 0;
  FOR (i, 1 TO x) z = z+y;
  return z;
}
```

Ισχυρισμός: Η συνάρτηση υπολογίζει πράγματι το γινόμενο δύο φυσικών αριθμών.

Πώς μπορούμε να βεβαιωθούμε ότι αληθεύει ο ισχυρισμός;

α) με **δοκιμές**:

1. εκτέλεση με το χέρι για διάφορες τιμές, πίνακας ιχνών (trace table) π.χ. `mult`

x	y	z	i
5	6	0	
		6	1
		12	2
		18	3
		24	4
		30	5

2. Εμπειρική δοκιμή με εκτέλεση από τον Η/Υ για διάφορες τιμές. (Σε μεγαλύτερη διαδικασία βάζουμε και μηνύματα εξόδου για έλεγχο της ροής). Οι δοκιμές είναι χρήσιμες για να βρίσκουμε λάθη. Δεν είναι όμως ξεκάθαρο πόσες δοκιμές πρέπει να κάνουμε για να βεβαιωθούμε ότι το πρόγραμμα είναι σωστό.

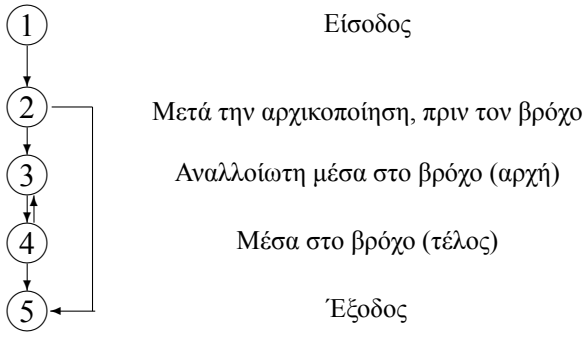
β) Θεωρητική μέθοδος: **Μαθηματική επαλήθευση ορθότητας.**

Αντί για συγκεκριμένες τιμές όπως στη μέθοδο (α), εδώ δουλεύουμε με συγκεκριμένες ιδιότητες των τιμών ή και σχέσεις μεταξύ των τιμών, οι οποίες ονομάζονται **βεβαιώσεις** (assertions), και ισχύουν πάντα όταν η διεργασία (process) φτάνει σε ένα ορισμένο σημείο. Στην περίπτωση επαναληπτικού βρόχου, η βεβαίωση στην αρχή του σώματος ισχύει πάντα όταν η διεργασία φτάνει (και πάλι) στην αρχή του σώματος και λέγεται **αναλλοίωτη** βεβαίωση βρόχου (loop invariant assertion).

Ξαναγράφουμε το πρόγραμμα με σχόλια-αριθμούς στα σημεία που θα βάλουμε βεβαιώσεις:

```
FUNC int mult (int x, int y)
{
  int i, /*1*/ z = 0; /*2*/
  FOR (i, 1 TO x)
    /*3*/ z = z+y /*4*/;
  /*5*/ return z;
}
```

Στο διάγραμμα που ακολουθεί περιγράφουμε όλους τους δυνατούς τρόπους ροής του προγράμματος:



Παρατήρηση:

Έστω z_0 η αρχική τιμή του z. Η μεταβλητή z μεταβάλλεται από τη **for** loop ως εξής:

(για i = 1) $z_1 = z_o + y$

(για i = 2) $z_2 = z_1 + y$

(για i = 3) $z_3 = z_2 + y$

...

(για i = n) $z_n = z_{n-1} + y$

Με πρόσθεση κατά μέλη έχουμε: $z_i = z_o + y * i$, άρα $z_i = y * i$, διότι $z_0 = 0$.

Οι βεβαιώσεις είναι:

/* 1. $x \geq 0, y \geq 0$: Βεβαίωση εισόδου. */

/* 2. $x \geq 0, y \geq 0, z = 0$. */

/* 3. $x \geq 0, y \geq 0, z = y * (i - 1)$ και $i \leq x$. Αναλλοίωτη βρόχου. */

/* 4. $x \geq 0, y \geq 0, z = y * i$. */

/* 5. $x \geq 0, y \geq 0, z = y * x$. Βεβαίωση εξόδου. */

Η επαλήθευση πρέπει να γίνει για όλους τους δυνατούς τρόπους ροής του προγράμματος, δηλ.: $1 \to 2$, $2 \to 3$, $2 \to 5$, $3 \to 4$, $4 \to 3$, $4 \to 5$.

- $1 \to 2$: Προφανές: x, y αμετάβλητα. Στο z εκχωρήθηκε η τιμή 0.
- $2 \to 3$: x, y, z αμετάβλητα. $i = 1$, συνεπώς $i - 1 = 0$ άρα $z = 0$. $i = 1 \leq x$, αλλοιώς δε θα ήταν σε αυτό το σημείο η διεργασία.
- $2 \to 5$: Δεν εκτελείται το σώμα του βρόχου, άρα $x = 0$. x, y, z:αμετάβλητα. Συνεπώς $z = x * y$ διότι $0 = 0 * y$.
- $3 \to 4$: x, y αμετάβλητα. νέο_z =παλιό_z+y, άρα παλιό_z =νέο_z−y. Ισχυε: παλιό_z=$(i-1) * y$, άρα νέο_z-y=$(i - 1) * y$, άρα νέο_ $z = i * y$.
- $4 \to 3$: μόνο το i αλλάζει: νέο_i = παλιό_i + 1. Ισχυε: z = παλιό_i∗y. Αρα z = (νέο_i−1)∗y Επίσης: $i \leq x$, αλλοιώς δεν θα ήταν σε αυτό το σημείο η διεργασία.
- $4 \to 5$: Τέλος εκτέλεσης βρόχου. Αρα $i = x$. Ισχυε: $z = i * y$. Αρα $z = x * y$.

Φυσικά, το πιο ενδιαφέρον, άρα και πιο δύσκολο, είναι να βρούμε τις απαιτούμενες βεβαιώσεις. Η συμβουλή μου είναι να αρχίζετε την κατασκευή βεβαιώσεων από το τέλος (προς την αρχή), δηλαδή από αυτό που θέλετε να αποδείξετε, την βεβαίωση εξόδου.

Άλλο Παράδειγμα:

Υπολογισμός δυνάμεων με την μέθοδο του επαναλαμβανόμενου τετραγωνισμού

Ιδέα: $y^6 = (y^3)^2 = (y^2 * y)^2 = y^4 * y^2$,
$y^{11} = y^{10} * y = (y^5)^2 * y = (y^4 * y)^2 * y = ((y^2)^2 * y)^2 * y = y^8 * y^2 * y^1$.
Μας χρειάζεται δηλαδή το δυαδικό ανάπτυγμα του εκθέτη, π.χ.:
6 σε βάση 10 = 110 σε βάση 2, 11 σε βάση 10 = 1011 σε βάση 2.

```
FUNC REAL power (REAL y, int j)
{
  /*1*/ REAL x=y, z; int i=j; /*2*/
  if (i<0) { /*3*/ x=1/x; i=abs(i); }
  /*4*/ z=1;
  while (i>0) {
   /*5*/ if (i%2 != 0) z=z*x;
   /*6*/ x=x*x; i=i/2; /*7*/
  }
  /*8*/ return z;
}
```

Μέθοδος α)

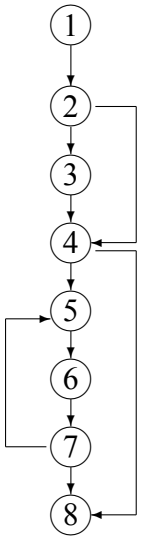
Trace Tables

power	y	j	x	i	z
	3	6	3	6	
					1
			9	3	
					9
			81	1	
					729
729			6561	0	

power	y	j	x	i	z
	2	11	2	11	
					1
					2
			4	5	
					8
			16	2	
			256	1	
					2048
2048			4194304	0	

Μέθοδος β)

Ροές και βεβαιώσεις:



/* 1. y is real and j is integer: Βεβαίωση εισόδου. */

/* 2. $(x = y)$ and $(i = j)$ */

/* 3. $(i < 0)$ */

/* 4. $(y^j = x^i)$ and $(i \geq 0)$ */

/* 5. $(y^j = z * x^i)$ and $(i > 0)$: Αναλλοίωτη βρόχου */

/* 6. $(y^j = z * x^i)$ if even(i), $(y^j = z * x^{i-1})$ if odd(i) */
[even = άρτιος]

/* 7. $y^j = z * x^i$ */ [και στις δύο περιπτώσεις, διότι γενικώς:
$w^k = (w^2)^{k/2} = (w^2)^{k \operatorname{div} 2} = w^{k \bmod 2}$]

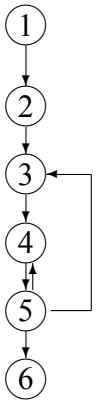
/* 8. $y^j = z$: Βεβαίωση εξόδου. */

Δεν αρκεί να αποδείξουμε την ορθότητα σε περίπτωση τερματισμού. Απομένει να αποδείξουμε ότι η διεργασία αυτή θα σταματήσει κάποτε. Η απόδειξη ορθότητας σε περίπτωση τερματισμού λέγεται **μερική ορθότητα**. Για την **ολική ορθότητα** χρειάζεται και απόδειξη τερματισμού.

Συνθήκη τερματισμού (termination condition) λέγεται μια γνησίως φθίνουσα θετική συνάρτηση που εγγυάται τον τερματισμό. Στο παραπάνω παράδειγμα η συνθήκη τερματισμού του βρόχου είναι η συνάρτηση i (σταματάει όταν i=0). Με τέτοιες επαληθεύσεις ασχολείται ο κλάδος της **Αξιωματικής Σημασιολογίας**.

Το τελευταίο παράδειγμα περιέχει loop φωλιασμένο μέσα σε loop.

```
PROGRAM nest ()
{
  int a, b, /*1*/ k = 0; /*2*/
  FOR (a, 1 TO 100) /*3*/
    FOR (b, 1 TO 100) /*4*/
      k = k+1 /*5*/ ;
  WRITELN(k); /*6*/
}
```



Ροές και βεβαιώσεις:

/* 1. Βεβαίωση εισόδου: integer declarations */

/* 2. $k = 0$ */

/* 3. $a \leq 100$, $k = 100 * (a - 1)$ */

/* 4. $b \leq 100$, $k = 100 * (a - 1) + (b - 1)$ */

/* 5. $k = 100 * (a - 1) + b$ */

/* 6. $k = 100 * 99 + 100 = 10000$: Βεβαίωση εξόδου */

Κεφάλαιο 9

Από την Pazcal στη C

Στο κεφάλαιο αυτό γίνεται μία σταδιακή μετάβαση από την Pazcal στη C. Υπενθυμίζεται ότι οι επεκτάσεις της Pazcal (όσα δηλαδή προσθέσαμε για εκπαιδευτικούς λόγους και δεν υπάρχουν στην απλή C) γράφονται πάντα με κεφαλαία γράμματα, π.χ. η "εντολή" **WRITE**. Στη συνέχεια του κεφαλαίου εξηγούνται οι διαφορές των δύο γλωσσών και περιγράφεται ο τρόπος με τον οποίο ο προγραμματιστής μπορεί να "μεταφράσει" όλες τις επεκτάσεις της Pazcal σε απλή C. Οπουδήποτε δίνεται κώδικας σε δύο στήλες, η αριστερή στήλη αντιστοιχεί σε κώδικα C και η δεξιά στον ισοδύναμο κώδικα σε Pazcal, π.χ.

```c
#include <stdio.h>

int main ()
{
  printf("Hello world!\n");
  return 0;
}
```

```
PROGRAM hello ()
{
  WRITELN("Hello world!");
}
```

Συγχρόνως, στη συνέχεια του κεφαλαίου, περιγράφονται μερικά ακόμη χαρακτηριστικά της C τα οποία είναι καλό (αλλά όχι απαραίτητο) να γνωρίζει κανείς όταν αρχίζει να προγραμματίζει. Στη συνέχεια των σημειώσεων, μετά από αυτό το κεφάλαιο, η γλώσσα που χρησιμοποιείται θα είναι ένα μείγμα Pazcal και απλής C.

9.1 Τύποι δεδομένων

Ενώ στην Pazcal υπάρχουν μόνο τέσσερις απλοί τύποι δεδομένων (**int**, **REAL**, **char** και **bool**), η C διαθέτει μια φαινομενική πληθώρα τέτοιων τύπων. Στην ουσία, οι διαφορετικοί τύποι είναι δύο: ακέραιοι (**int** αλλά και **char**) και πραγματικοί αριθμοί, σε αναπαράσταση κινητής υποδιαστολής (**float** και **double**). Η C όμως υποστηρίζει πολλές παραλλαγές αυτών των δύο βασικών τύπων, που διαφέρουν στην ύπαρξη προσήμου (**signed** ή **unsigned**) και στο μέγεθος (**short**, κανονικό ή **long**).

Οι απλοί τύποι δεδομένων που υποστηρίζει η C δίνονται στον πίνακα που ακολουθεί. Από πάνω προς τα κάτω, οι τιμές αυτών των τύπων χρησιμοποιούν σταδιακά περισσότερη μνήμη για την αναπαράστασή τους. Οι λέξεις που είναι σημειωμένες με αχνότερα γράμματα μπορούν να

παραλείπονται (φυσικά στην περίπτωση του **signed int** μπορεί να παραλείπεται μόνο η μία από τις δύο λέξεις).

char	**signed char**	**unsigned char**
signed short int		**unsigned short int**
	signed int	**unsigned int**
signed long int		**unsigned long int**
	float	
	double	
	long double	

Οι χαρακτήρες είναι στην πραγματικότητα ακέραιοι αριθμοί: κάθε χαρακτήρας αντιστοιχεί στον κωδικό με τον οποίο αυτός αντιπροσωπεύεται στον πίνακα ASCII. Για παράδειγμα, αν η μεταβλητή c έχει δηλωθεί με τύπο **char**, οι παρακάτω αναθέσεις είναι ισοδύναμες:

```
c = 'a';
c = 97;    // ο ASCII κωδικός του γράμματος 'a' είναι 97
```

Οι διάφοροι τύποι ακεραίων διαφέρουν ως προς το εύρος τιμών που μπορούν να αναπαραστήσουν. Για παράδειγμα, ο τύπος **unsigned char** μπορεί να αναπαραστήσει τις τιμές από 0 έως $2^8 = 255$, ο τύπος **signed char** τις τιμές από $-128 = -2^7$ έως $127 = 2^7 - 1$, ενώ ο τύπος **int**, στους σημερινούς υπολογιστές, μπορεί συνήθως να αναπαραστήσει τιμές από $-2,147,483,648 = -2^{31}$ έως $2,147,483,647 = 2^{31} - 1$. Οι τύποι **float**, **double** και **long double** διαφέρουν όχι μόνο ως προς το εύρος των πραγματικών τιμών που μπορούν να αναπαραστήσουν, αλλά και ως προς την ακρίβεια της αναπαράστασης (δηλαδή χοντρικά, ως προς το πλήθος των δεδαδικών ψηφίων της mantissa, μετά την υποδιαστολή).

9.2 Πρόγραμμα και υποπρογράμματα

Ενώ στην Pazcal ένα πρόγραμμα μπορεί να αποτελείται από το κυρίως πρόγραμμα (**PROGRAM**), διαδικασίες (**PROC**) και συναρτήσεις (**FUNC**), στη C υπάρχουν μόνο συναρτήσεις.

- Οι συναρτήσεις γράφονται όπως στην Pazcal, μόνο που δε χρειάζεται η λέξη-κλειδί **FUNC**:

```
int f (...)
{
  ...
}
```

```
FUNC int f (...)
{
  ...
}
```

- Οι διαδικασίες γράφονται ως συναρτήσεις, που επιστρέφουν τον ειδικό τύπο **void**. Ο τύπος αυτός σημαίνει ότι η εν λόγω συνάρτηση δεν έχει καμία σημαντική πληροφορία να επιστρέψει, απλά επιστρέφει.

```
void p (...)
{
  ...
}
```

```
PROC p (...)
{
  ...
}
```

- Τέλος, το κυρίως πρόγραμμα είναι κι αυτό μία συνάρτηση που φέρει πάντοτε το ειδικό όνομα `main`. Ο τύπος που επιστρέφει πρέπει να είναι **`int`** και η τιμή που επιστρέφει είναι ορατή από το λειτουργικό σύστημα μετά την εκτέλεση του προγράμματος. Κατά σύμβαση, αυτή η τιμή υποδηλώνει αν η εκτέλεση του προγράμματος ολοκληρώθηκε φυσιολογικά (τιμή 0) ή με κάποιο σφάλμα εκτέλεσης (τιμή διάφορη του 0).

```
int main ()
{
  ...
  return 0;
}
```

```
PROGRAM example ()
{
  ...
}
```

9.3 Ανάθεση

Μία βασική διαφορά ανάμεσα στην Pazcal και τη C είναι ότι, στη C, η ανάθεση δεν είναι εντολή. Είναι μία παράσταση που η αποτίμησή έχει δύο επιπτώσεις στην εκτέλεση του προγράμματος:

- Αφενός προκαλεί την ανάθεση της τιμής που βρίσκεται στο δεξιό μέλος στη μεταβλητή που βρίσκεται στο αριστερό μέλος, όπως και στην Pazcal.

- Αφετέρου, όπως όλες οι παραστάσεις, υπολογίζει μία τιμή. Η τιμή που υπολογίζεται από την ανάθεση είναι ακριβώς η τιμή που ανατίθεται στη μεταβλητή του αριστερού μέλους. Αυτό σημαίνει ότι ο κώδικας που ακολουθεί είναι έγκυρος στη C:

```
y = (x = 17) + 25;
x = y = z = 0;
```

```
x = 17; y = x + 25;
x = 0; y = 0; z = 0;
```

  Η πρώτη γραμμή αναθέτει στη μεταβλητή `x` την τιμή 17 και στη μεταβλητή `y` την τιμή 42 (που υπολογίζεται ως το άθροισμα του 17, που επιστρέφει η πρώτη ανάθεση, και του 25). Η δεύτερη γραμμή αναθέτει στις τρεις μεταβλητές `x`, `y` και `z` την τιμή 0. Αυτό γίνεται γιατί ο τελεστής της ανάθεσης είναι **δεξιά προσεταιριστικός**, δηλαδή η παράσταση `x = y = z = 0` είναι ισοδύναμη με `x = (y = (z = 0))`.

Επίσης, η C υποστηρίζει τελεστές **σύνθετης ανάθεσης** για κάθε τελεστή με δύο τελούμενα. Η σημασία των τελεστών αυτών εξηγείται στις ακόλουθες δύο γραμμές:

```
x += 42;
i %= n+1;
```

```
x = x + 42;
i = i % (n+1);
```

Οι τελεστές `++` και `--` προκαλούν αύξηση και μείωση κατά ένα, αντίστοιχα (το ίδιο με το `+= 1` και `-= 1`). Εμφανίζονται με δύο μορφές: είτε ακολουθούν το τελούμενο που μεταβάλλουν (postfix), είτε προηγούνται αυτού (prefix). Αν χρησιμοποιηθούν σαν εντολές, χωρίς να χρησιμοποιηθεί το αποτέλεσμα που υπολογίζουν, οι δύο μορφές είναι ισοδύναμες:

```
i++;  /* ή */  ++i;
i--;  /* ή */  --i;
```

```
i = i+1;
i = i-1;
```

Αν όμως το αποτέλεσμα χρησιμοποιηθεί, τότε οι δύο μορφές διαφέρουν ως προς την τιμή του αποτελέσματος. Ο τελεστής postfix επιστρέφει την τιμή *πριν γίνει* η αύξηση ή η μείωση, ενώ ο τελεστής prefix επιστρέφει την τιμή *αφού γίνει* η αύξηση ή η μείωση.

```
i = 3; x = i++;
i = 3; x = ++i;
```

```
i = 3; x = i; i = i+1;
i = 3; i = i+1; x = i;
```

Και στις δύο από τις παραπάνω γραμμές, η μεταβλητή `i` θα έχει την τιμή 4 μετά την εκτέλεσή τους. Όμως, μετά την εκτέλεση της πρώτης από τις δύο, η τιμή της μεταβλητής `x` θα είναι 3 (όσο ήταν η τιμή του `i` **πριν** την αύξηση). Αντίθετα, μετά την εκτέλεση της δεύτερης, η τιμή της μεταβλητής `x` θα είναι 4 (όσο είναι η τιμή του `i` **μετά** την αύξηση).

Προσέξτε ότι η C απαγορεύει τη χρήση παραστάσεων που μεταβάλλουν δύο φορές την τιμή της ίδιας μεταβλητής. Η παρακάτω γραμμή θεωρείται λάθος και μπορεί να έχει διαφορετικό αποτέλεσμα σε διαφορετικούς υπολογιστές, διαφορετικούς μεταγλωττιστές ή ακόμα και στον ίδιο υπολογιστή και μεταγλωττιστή, αν εκτελεστεί πολλές φορές.

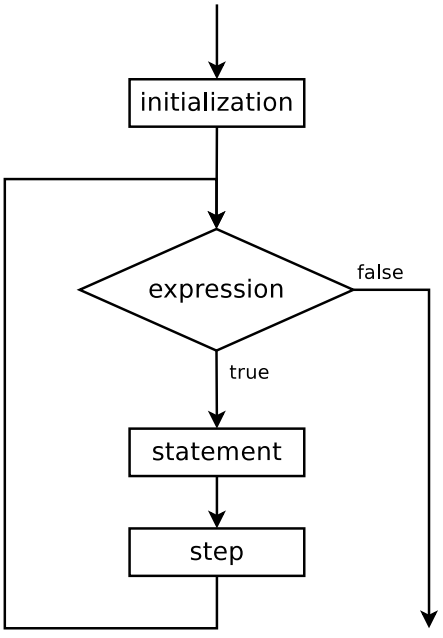
```
i = i++;    // λάθος!
```

9.4 Βρόχος for

Η C υποστηρίζει τρία είδη βρόχων: **`while`**, **`do`** . . . **`while`** (ακριβώς όπως και η Pazcal) και **`for`**. Αυτό το τρίτο είδος βρόχου είναι ουσιαστικά γενίκευση του βρόχου **`while`** και, αν και μπορεί να χρησιμοποιηθεί για την υλοποίηση του **`FOR`** της Pazcal, έχει σημαντικές διαφορές. Η σύνταξη του βρόχου **`for`** στη C είναι η εξής:

`for` (*initialization* ; *expression* ; *step*) *statement*

και το αντίστοιχο διάγραμμα ροής:



Η αρχικοποίηση (initialization) και το βήμα (step) είναι παραστάσεις στις οποίες συνήθως γίνεται κάποια ανάθεση. Η αρχικοποίηση εκτελείται μόνο στην αρχή του βρόχου, ενώ το βήμα

εκτελείται στο τέλος κάθε επανάληψης του σώματος. Αν παραλειφθούν, τότε θεωρούνται κενές (δηλαδή δε γίνεται καμία αρχικοποίηση στην αρχή και καμία ενέργεια στο τέλος κάθε επανάληψης). Επίσης, αν παραλειφθεί η συνθήκη (condition) εννοείται **true**. Ο βρόχος **for** `(;;) ...` είναι συνώνυμος με το **while** `(true) ...`.

Στις παρακάτω γραμμές φαίνεται πώς η εντολή **for** της C μπορεί να χρησιμοποιηθεί για να δώσει το ίδιο αποτέλεσμα με την FOR της Pazcal.

```
for (i=1; i<=10; i++) ...
for (i=8; i>=1; i--) ...
for (i=1; i<=10; i+=2) ...
```

```
FOR (i, 1 TO 10) ...
FOR (i, 8 DOWNTO 1) ...
FOR (i, 1 TO 10 STEP 2) ...
```

Προσέξτε όμως ότι στην εκτέλεση της **for** δε γίνεται τίποτα ιδιαίτερο ούτε για τα όρια του βρόχου ούτε για την προστασία της μεταβλητής ελέγχου. Αυτό σημαίνει ότι τα ακόλουθα τμήματα κώδικα, σε C και σε Pazcal, θα δώσουν διαφορετικά αποτελέσματα:

```
// διαφορετικό αποτέλεσμα!
n = 3;
for (i=1; i<=n; i++) n++;
```

```
n = 3;
FOR (i, 1 TO n) n = n+1;
```

Συγκεκριμένα, η εκτέλεση της εντολής **for** δε θα τερματιστεί ποτέ γιατί οι μεταβλητές i και n θα αυξάνουν συγχρόνως και ποτέ η πρώτη δε θα ξεπεράσει τη δεύτερη. Αντίθετα, η εκτέλεση της εντολής FOR στην Pazcal θα τερματιστεί και η τελική τιμή του n θα είναι 6, όπως είδαμε στην Ενότητα 3.4.1.

Ο τελεστής , (κόμμα) της C μπορεί να χρησιμοποιηθεί για να γίνουν περισσότερες αρχικοποιήσεις ή βήματα σε κάθε επανάληψη της **for**.

```
for (i=1, s=0; i <= 10; i++) s += i;
for (i=1, j=100; i <= j; i++, j--) printf("%d %d\n", i, j);
```

(Γενικά, ο τελεστής κόμμα υπολογίζει το αριστερό του τελούμενο, υπολογίζει το δεξιό του τελούμενο και επιστρέφει την τιμή του δεξιού τελουμένου.)

Τέλος, η C99 υποστηρίζει τη δήλωση μεταβλητών μέσα στην εντολή **for**, π.χ.

```
for (int i=0; i < 10; i++) ...
```

Στην περίπωση αυτή, η μεταβλητή i είναι ορατή μόνο μέσα στο σώμα του βρόχου και δεν μπορεί να χρησιμοποιηθεί μετά το τέλος της εκτέλεσής του.

9.5 Έξοδος στην οθόνη

Για την εκτύπωση στην οθόνη, η C χρησιμοποιεί κατ' εξοχήν τη συνάρτηση printf. Πριν τη χρησιμοποιήσετε, πρέπει να προσθέσετε στην αρχή του αρχείου προγράμματός σας την ακόλουθη γραμμή:

```
#include <stdio.h>
```

Η πρώτη παράμετρος της printf είναι μία συμβολοσειρά που ονομάζεται format. Η συμβολοσειρά αυτή περιέχει αυτά που θέλουμε να εκτυπώσουμε. Μπορεί να περιέχει το χαρακτήρα

αλλαγής γραμμής `\n`, αν θέλουμε να προκαλέσουμε αλλαγή γραμμής. Αν μέσα στο format εμφανίζεται ο χαρακτήρας `%`, τότε αυτός έχει ειδική σημασία, ανάλογα με το τι ακολουθεί:

- `%d`: στη θέση του εκτυπώνεται η επόμενη παράμετρος της `printf`, που πρέπει να είναι ένας ακέραιος αριθμός.
- `%c`: στη θέση του εκτυπώνεται η επόμενη παράμετρος της `printf`, που πρέπει να είναι ένας χαρακτήρας (δηλαδή ένας ακέραιος αριθμός που αντιστοιχεί στον ASCII κωδικό ενός χαρακτήρα).
- `%lf`: στη θέση του εκτυπώνεται η επόμενη παράμετρος της `printf`, που πρέπει να είναι ένας πραγματικός αριθμός τύπου **`double`**.
- `%s`: στη θέση του εκτυπώνεται η επόμενη παράμετρος της `printf`, που πρέπει να είναι μία συμβολοσειρά.

Προσέξτε ότι το format μπορεί να περιέχει οσεσδήποτε φορές το χαρακτήρα `%` και, κάθε φορά, εκτυπώνεται η επόμενη παράμετρος της `printf`:

```
printf("Hello\n");
printf("%d", i+1);
printf("%d %lf", i, r);
printf("%c", c);
```

```
WRITELN("Hello");
WRITE(i+1);
WRITESP(i, r);
WRITE(c);
```

Η `printf` υποστηρίζει πολλές επιλογές για τη μορφοποίηση των τιμών που θα εκτυπωθούν, όπως για παράδειγμα:

```
printf("%5d", i);
printf("%5.3lf", r);
```

```
WRITE(FORM(i,5));
WRITE(FORM(r,5,3));
```

Ανατρέξτε στη σελίδα εγχειριδίου (man page) της `printf`, ή αναζητήστε τη στο Internet.

Προέξτε ότι με την `printf` το format είναι αυτό που καθορίζει με ποια μορφή θε εκτυπωθούν οι προς εκτύπωση τιμές. Αυτό φαίνεται ιδιαίτερα στην παρακάτω εντολή, όπου χαρακτήρες και ακέραιοι τυπώνονται με όλους τους δυνατούς συνδυασμούς μορφών:

```
// οι χαρακτήρες είναι ακέραιοι!                    a 97 a 97
printf("%c %d %c %d\n", 'a', 97, 97, 'a');
```

9.6 Είσοδος από το πληκτρολόγιο

Για την είσοδο τιμών από το πληκτρολόγιο, στη C χρησιμοποιούμε τις περισσότερες φορές τη συνάρτηση `scanf`, για την οποία και πάλι θα πρέπει να κάνουμε `#include <stdio.h>`.

Η `scanf` λειτουργεί όπως και η `printf` και η πρώτη παράμετρος (format) καθορίζει τι πρόκειται να διαβαστεί. Οι υπόλοιπες παράμετροι όμως της `scanf` πρέπει να είναι **δείκτες** προς τις μεταβλητές όπου θα αποθηκευτούν οι τιμές. Περισσότερα για δείκτες θα βρείτε στην επόμενη ενότητα.

```
scanf("%d", &i);
scanf("%lf", &r);
```

```
i = READ_INT();
r = READ_REAL();
```

Για την εισαγωγή χαρακτήρων μπορεί να χρησιμοποιηθεί φυσικά και η συνάρτηση `getchar`:

```
c = getchar();
  /* ή */
scanf("%c", &c);
```

```
c = getchar();
```

Προσέξτε ότι στη C δεν υπάρχει κάτι αντίστοιχο με τη `SKIP_LINE` της Pazcal. Αν θέλετε να διαβάσετε και να αγνοήσετε όλους τους υπόλοιπους χαρακτήρες μέχρι και την αλλαγή γραμμής στο τέλος του buffer εισόδου, μπορείτε να χρησιμοποιήσετε ένα βρόχο:

```
while (getchar() != '\n');
```

9.7 Δείκτες

Αναλυτική παρουσίαση των δεικτών γίνεται στο Κεφάλαιο 14.

Δείκτης (pointer): η διεύθυνση μιας περιοχής της μνήμης όπου βρίσκεται μια μεταβλητή. Τα περιεχόμενα της θέσης μνήμης όπου δείχνει ο δείκτης p συμβολίζονται με `*p`. Για κάθε δείκτη, πρέπει να δηλώνεται ο τύπος της θέσης μνήμης όπου αυτός δείχνει (π.χ. δείκτης σε ακέραιο)

```
int *p;    // η μεταβλητή p είναι δείκτης σε ακέραιο
...
// ο δείκτης p τοποθετείται να δείχνει σε κάποια ακέραια μεταβλητή
...
*p = 42;
WRITELN(*p + 1);
```

Κενός δείκτης (`NULL`): ειδική τιμή δείκτη που δε δείχνει πουθενά. Απαγορεύεται η προσπέλαση της μνήμης μέσω ενός κενού δείκτη.

```
int *p;
...
p = NULL;
WRITELN(*p);    // λάθος!
```

Δεικτοδότηση: μπορούμε να πάρουμε τη διεύθυνση μιας μεταβλητής (π.χ. για να βάλουμε κάποιον δείκτη να δείχνει εκεί) με τον τελεστή `&`.

```
int x = 17, *p;
p = &x;
```

Αποδεικτοδότηση: όπως είδαμε, μπορούμε να αναφερθούμε στο περιεχόμενο μιας διεύθυνσης με τον τελεστή `*`.

```
WRITELN(*p);  // τυπώνει 17, στο προηγούμενο παράδειγμα
*p = 42;
WRITELN(x);   // τυπώνει 42, τα *p και x είναι η ίδια μεταβλητή!
```

Παράδειγμα με δείκτες

```
PROGRAM example ()
{
  int x = 42, y = 17;
  int *p, *q;
  p = &x;
  q = &y;
  *p = *p - *q;
  *q = *p *y;
  q = p;
  (*q)++;
  *p -= 3;
  WRITESPLN(x, y);     // τυπώνει: 23 425
}
```

Δείκτες αντί περάσματος με αναφορά

Στο παρακάτω παράδειγμα, οι παράμετροι της συνάρτησης `normalize` είναι δείκτες προς δύο ακέραιες μεταβλητές. Με αυτόν τον τρόπο, η συνάρτηση αυτή κάνει ανάγωγο το κλάσμα που ο αριθμητής του δείχνεται από το δείκτη `p` και ο παρονομαστής του δείχνεται από το δείκτη `q`. Προσέξτε τη χρήση του τελεστή `&` στην κλήση της `normalize` μέσα στη συνάρτηση `main`.

Αν οι παράμετροι `p` και `q` ήταν ακέραιου τύπου, τότε η `normalize` θα έκανε ανάγωγο το κλάσμα αλλά, μετά την επιστροφή της, η `main` δε θα έβλεπε καμία διαφορά στις τιμές των τοπικών της μεταβλητών `x` και `y`.

```
int gcd (int a, int b);

void normalize (int *p, int *q)
{
  int g = gcd(*p, *q);
  *p /= g;  *q /= g;
}

int main ()
{
  int x, y;

  scanf("%d %d", &x, &y);
  normalize(&x, &y);
  printf("%d %d\n", x, y);
  return 0;
}
```

Σε πολλές γλώσσες προγραμματισμού (π.χ. Pascal), όχι όμως στη C, δεν είναι αναγκαία η χρήση δεικτών για το σκοπό αυτό. Υπάρχει ο τρόπος περάσματος **κατ' αναφορά** (by reference) για τις παραμέτρους των υποπρογραμμάτων, που κάνει ουσιαστικά το ίδιο πράγμα.

9.8 Πίνακες και δείκτες

Έστω ένας πίνακας τριών ακεραίων και ένας δείκτης σε ακέραιο:

```
int a[3] = {7, 6, 42};
int *p;
```

Οι παρακάτω τρεις αναθέσεις είναι ισοδύναμες: όλες θέτουν το δείκτη `p` να δείχνει στο πρώτο στοιχείο του πίνακα (`a[0]`).

```
p = &(a[0]);
p = &a;
p = a;
```

δηλαδή, ένας πίνακας είναι ουσιαστικά ένας δείκτης στο πρώτο στοιχείο του.

Η παράσταση `a[i]` είναι ισοδύναμη με `*(a+i)`.

Οι πίνακες όμως είναι σταθεροί δείκτες, δηλαδή δεν μπορούν να αλλάξουν τιμή:

```
int a[3] = {7, 6, 42};
int *p = &a;
int x;

p = &x;  // σωστό
a = &x;  // λάθος!
```

Αριθμητική δεικτών

Συνεχίζοντας το προηγούμενο παράδειγμα:

```
WRITELN(*p);       // τυπώνει 7
WRITELN(*(p+1));   // τυπώνει 6
p = p+2;
WRITELN(*p);       // τυπώνει 42
```

δηλαδή με τον τελεστή πρόσθεσης (αντίστοιχα, αφαίρεσης) μπορούμε να μετακινήσουμε ένα δείκτη για να δείχνει σε επόμενο (αντίστοιχα, προηγούμενο) στοιχείο ενός πίνακα. Όμως, με την αριθμητική δεικτών δεν επιτρέπεται να βγούμε έξω από τον πίνακα στον οποίο έδειχνε ο αρχικός δείκτης!

9.9 Συμβολοσειρές

Οι συμβολοσειρές δεν είναι τίποτα περισσότερο από πίνακες χαρακτήρων, δηλαδή ισοδύναμα, δείκτες σε χαρακτήρες. Περισσότερα για αυτές, και για χρήσιμες συναρτήσεις βιβλιοθήκης που τις επεξεργάζονται, στο Κεφάλαιο 13.

Για να γνωρίζει ένα πρόγραμμα πού τελειώνει μία συμβολοσειρά, στη C γίνεται η σύμβαση ότι μετά το τέλος των χαρακτήρων υπάρχει ο ειδικός χαρακτήρας `'\0'` (δηλαδή ο χαρακτήρας με ASCII κωδικό μηδέν).

```
char a[15] = ”Hello world!”, b[15];
// a[12] == ’\0’
```

Αντιγραφή συμβολοσειράς

```
void strcpy (char *t, char *s)
{
  while ((*t++ = *s++) != ’\0’);
}

int main ()
{
  strcpy(b, a);
  printf(”%s\n”, a);
  return 0;
}
```

Εκτύπωση συμβολοσειράς

```
void putchar (char c);

void puts (char *p)
{
  while (*p != ’\0’) putchar(*p++);
}

int main ()
{
  char s[] = ”Hello world!\n”;
  puts(s);
  return 0;
}
```

Κεφάλαιο 10

Ταξινόμηση (sorting)

Μια από τις σημαντικότερες εφαρμογές των Η/Υ είναι η **ταξινόμηση** (sorting) ενός γραμμικού πίνακα ανάλογα με κάποιο κριτήριο (αύξουσα τάξη, αλφαβητικά κ.ο.κ.). Ίσως κάποιος αναρωτηθεί γιατί είναι τόσο απαραίτητη η μελέτη της ταξινόμησης. Η απάντηση δίνεται από την πραγματικότητα:

- Οι λέξεις σε ένα λεξικό είναι ταξινομημένες κατά αλφαβητική σειρά.
- Τα ονόματα στον τηλεφωνικό κατάλογο είναι ταξινομημένα κατά αλφαβητική σειρά.
- Οι σελίδες ενός εγγράφου είναι κατά αριθμητική σειρά διαρθρωμένες.
- Στις τηλεπικοινωνίες και στα δίκτυα υπολογιστών η διαχείριση και η αποθήκευση πληροφοριών συνοδεύεται συνήθως με αλγόριθμους ταξινόμησης και εύρεσης αριθμών.
- Φοροτεχνικοί κώδικες απαιτούν αρκετές φορές αλγόριθμους ταξινόμησης ή εύρεση αριθμών.
- Στις βιβλιοθήκες είναι ταξινομημένα τα υπάρχοντα βιβλία και άρθρα.

Και ο κατάλογος είναι ανεξάντλητος.

Μια βασική διαδικασία που σίγουρα χρειαζόμαστε είναι η εναλλαγή δυο τιμών, στην προκειμένη περίπτωση των τιμών των ακέραιων μεταβλητών στις οποίες δείχνουν τα `x` και `y`.

```
PROC swap (int *x, int *y)
{
  int save;
  save = *x; *x = *y; *y = save;
}
```

Άλλος τρόπος υλοποίησης, που μπορεί να εφαρμοστεί μόνο για την αντιμετάθεση ακέραιων αριθμών και εκμεταλλεύεται τις ιδιότητες των πράξεων της πρόσθεσης και της αφαίρεσης.

```
PROC swap (int *x, int *y)
{
  *x =  *x + *y; *y = *x - *y; *x = *x - *y;
}
```

Με τον δεύτερο τρόπο δε χρειάζεται βοηθητική μεταβλητή στην swap. Ωστόσο ο πρώτος είναι πολύ καλύτερος διότι είναι πιο αναγνώσιμος, πιο αποδοτικός και δεν είναι επιρρεπής σε σφάλματα υπερχείλισης, όταν οι τιμες που πρέπει να αντιμετατεθούν είναι πολύ μεγάλες κατ' απόλυτο τιμή.

10.1 Ταξινόμηση με επιλογή (selection sort)

Ιδέα: βρες το μικρότερο στοιχείο του πίνακα και αντιμετάθεσέ το με εκείνο που βρίσκεται στην πρώτη θέση. Στη συνέχεια, βρες το επόμενο μικρότερο στοιχείο και αντιμετάθεσέ το με εκείνο που βρίσκεται στη δεύτερη θέση. Συνέχισε έτσι μέχρι να ταξινομηθούν όλα τα στοιχεία.

```
FOR (i, 0 TO n-2) {
  int minj = i;
  FOR (j, i+1 TO n-1) if (a[j] < a[minj]) minj = j;
  swap(&(a[i]), &(a[minj]));
}
```

Αναλλοίωτη: στο τέλος της i-οστής επανάληψης του εξωτερικού βρόχου, στις πρώτες θέσεις του πίνακα μέχρι και την a[i] βρίσκονται τα τα $i+1$ μικρότερα στοιχεία σε αύξουσα σειρά.

Πλήθος συγκρίσεων: $O(n^2)$.

10.2 Ταξινόμηση με εισαγωγή (insertion sort)

Ιδέα: Προσπέρασε το πρώτο στοιχείο. Σύγκρινε το δεύτερο με το πρώτο και βάλε το στη σωστή σειρά. Σύγκρινε το τρίτο με τα δύο πρώτα και βάλε το στη σωστή σειρά, κ.ο.κ. Σε κάθε βήμα, συγκρίνεις το i-οστό στοιχείο με τα προηγούμενα, που είναι ήδη ταξινομημένα, και το εισάγεις στη σωστή σειρά "σπρώχνοντας" δεξιότερα τα στοιχεία που είναι μεγαλύτερά του.

```
FOR (i, 1 TO n-1) {
  int x = a[i], j = i;
  while (j > 0 AND a[j-1] > x) { a[j] = a[j-1]; j = j-1; }
  a[j] = x;
}
```

Αναλλοίωτη: στο τέλος της i-οστής επανάληψης του εξωτερικού βρόχου, τα στοιχεία που βρίσκονται στις πρώτες θέσεις του πίνακα μέχρι και την a[i] είναι σε αύξουσα σειρά.

Πλήθος συγκρίσεων: $O(n^2)$.

10.3 Ταξινόμηση φυσαλλίδας (bubble sort)

Το επόμενο βήμα για την ταξινόμηση είναι η ιδέα ότι με τις συγκρίσεις και εναλλαγές θέσεων μπορούμε να φέρουμε τον μικρότερο αριθμό στην αριστερότερη θέση, κατόπιν παρομοίως τον δεύτερο μικρότερο στη δεύτερη από αριστερά θέση, κ.ο.κ. (χρειαζόμαστε έτσι n επαναλήψεις).

Με μια πρόσθετη ιδέα να συγκρίνουμε και να εναλλάσσουμε μόνο γειτονικά στοιχεία, καταλήγουμε σε ένα αλγόριθμο ταξινόμησης με φυσαλλίδες (bubblesort). Ιδέα:

```
FOR (i, 0 TO n-2)
  FOR (j, n-2 DOWNTO i)
    if (a[j] > a[j+1]) swap(&(a[j]), &(a[j+1]));
```

Η ταξινόμηση γίνεται με διαδοχικά περάσματα. Με το πρώτο πέρασμα (i=0) ο μικρότερος αριθμός καταλήγει στην αριστερότερη θέση με (n-1) συγκρίσεις, αλλά συγχρόνως γίνονται και άλλες μετατοπίσεις προς τη σωστή κατεύθυνση. Στο i-οστο πέρασμα τοποθετείται στην i-οστή θέση το i-οστό (σε τάξη μεγέθους) στοιχείο. Αν μετρήσουμε τον αριθμό των συγκρίσεων θα δούμε ότι είναι:

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}=O(n^2)$$

Σημείωση:

$O(f(n))$ σημαίνει τάξη μεγέθους $f(n)$, οι σταθερές παραλείπονται.

Ας δούμε μια εκτέλεση του προηγούμενου προγράμματος:

Έστω ότι αρχικά έχουμε τον πίνακα $n=7$ αριθμών: `12 4 9 8 6 7 5`.

Trace Table:

i	j	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
		12	4	9	8	6	7	5
0	5 4 3 2 1 0	 4	 12	 5 	 5 9 	 5 8 	5 6 	7
1	5 4 3 2 1	 	 5	 12	 6 	 6 9 	 8 	
2	5 4 3 2	 	 	 6	 12	 7 	7 9 	8
3	5 4 3	 	 	 	 7	 12	8 	9
4	5 4	 	 	 	 	 8	 12	
5	5						9	12

Τα παραπάνω αποτελέσματα παρουσιάζονται σε μια πιο σχηματική μορφή ως εξής (Με έντονη γραφή εμφανίζονται τα στοιχεία που εκάστοτε έχουν συγκριθεί και πιθανόν αντιμεταττεθεί):

input: 12 4 9 8 6 7 5

---

12 4 9 8 6 **5 7**
12 4 9 8 **5 6** 7
12 4 9 **5 8** 6 7
12 4 **5 9** 8 6 7
12 **4 5** 9 8 6 7
$i = 0$: **4 12** 5 9 8 6 7

---

4 12 5 9 8 **6 7**
4 12 5 9 **6 8** 7
4 12 5 **6 9** 8 7
4 12 **5 6** 9 8 7
$i = 1$: 4 **5 12** 6 9 8 7

---

4 5 12 6 9 **7 8**
4 5 12 6 **7 9** 8
4 5 12 **6 7** 9 8
$i = 2$: 4 5 **6 12** 7 9 8

---

4 5 6 12 7 **8 9**
4 5 6 12 **7 8** 9
$i = 3$: 4 5 6 **7 12** 8 9

---

4 5 6 7 12 **8 9**
$i = 4$: 4 5 6 7 **8 12** 9

$i = 5$: 4 5 6 7 8 **9 12**

Στη μέθοδο της φυσαλλίδας όπως την υλοποίησαμε παραπάνω, ο αριθμός συγκρίσεων είναι $O(n^2)$ για κάθε περίπτωση. Εάν σταματήσουμε την επανάληψη μόλις βεβαιωθούμε ότι δεν χρειάζονται πια άλλες εναλλαγές, τότε ο αριθμός των συγκρίσεων κατά μέσο όρο θα είναι πολύ μικρότερος (στην καλύτερη περίπτωση $O(n)$ στη χειρότερη όμως περίπτωση πάλι $O(n^2)$:

```
FOR (i, 0 TO n-2) {
  bool swaps = false;
  FOR (j, n-2 DOWNTO i)
    if (a[j] > a[j+1]) { swaps = true; swap(&(a[j]), &(a[j+1])); }
  if (NOT swaps) break;
}
```

10.4 Ταξινόμηση με συγχώνευση (merge sort)

Υπάρχουν πολλές μέθοδοι ταξινόμησης: selection sort, insertion sort, bubble sort, merge sort, quick sort, κ.λπ. Όπως θα δούμε μπορεί να επιτευχθεί αλγόριθμος που ταξινομεί και στη χειρότερη ακόμα περίπτωση με αριθμό συγκρίσεων $O(n \log n)$. Επίσης θα αποδείξουμε οτι είναι αδύνατον να επιτευχθεί αλγόριθμος (που ταξινομεί δια συγκρίσεων) με μικρότερο αριθμό συγκρίσεων. Η μέθοδος ταξινόμησης με συγχώνευση (merge sort, John von Neumann, 1945) χρησιμοποιεί τη στρατηγική "διαίρει και κυρίευε". Η καλύτερη δυνατή υλοποίηση είναι με **αναδρομή** (recursion).

Παράδειγμα: Για να ταξινομήσω: 5, 7, 2, 13, 29, 3, 16, 17

1. διαιρώ σε δύο: 5, 7, 2, 13 και 29, 3, 16, 17
2. ταξινομώ με την ίδια ιδέα και τα δύο μέρη (αναδρομικά): 2, 5, 7, 13, και 3, 16, 17, 29,
3. μετά συγχωνεύω: 2, 3, 5, 7, 13, 16, 17, 29.

```
PROC mergesort (int a[], int first, int last)
{
  int mid;

  if (first >= last) return;

  mid = (first + last) / 2;
  mergesort(a, first, mid);   mergesort(a, mid+1, last);
  merge(a, first, mid, last);
}

PROC merge (int a[], int first, int mid, int last)
{
  int b[last-first+1];

  int i = first, j = mid+1, k = 0;
  while (i <= mid AND j <= last) if (a[i] < a[j]) b[k++] = a[i++];
                                 else             b[k++] = a[j++];

  while (i <= mid)   b[k++] = a[i++];
  while (j <= last)  b[k++] = a[j++];

  FOR (i, 0 TO k-1)  a[first+i] = b[i];    // copy back
}
```

Η διαδικασία `mergesort` ταξινομεί το τμήμα του πίνακα `a` που βρίσκεται μεταξύ των θέσεων `first` και `last`, συμπεριλαμβανομένων. Διαιρεί αυτό το τμήμα σε δύο μισά: από `first` μέχρι `mid` και από `mid+1` μέχρι `last`. Ταξινομεί αναδρομικά αυτά τα δύο μισά και στη συνέχεια τα συγχωνεύει.

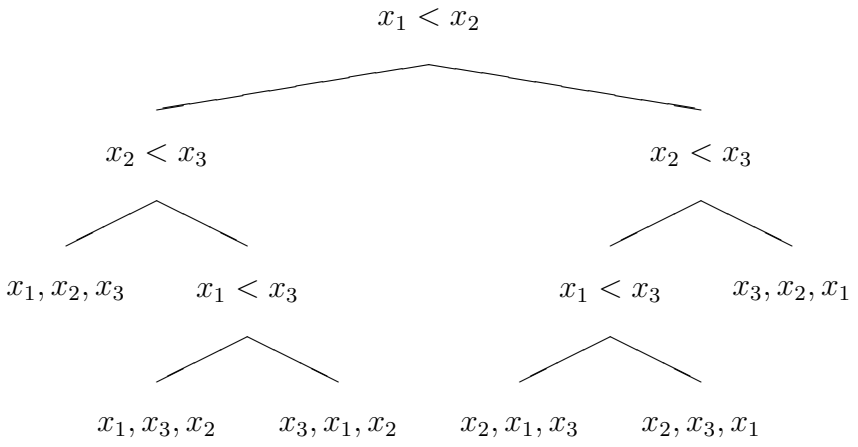
Η συγχώνευση των δύο ταξινομημένων μισών πινάκων σε έναν πίνακα, που στο τέλος είναι και αυτός ταξινομημένος, γίνεται με τη διαδικασία `merge`. Χρησιμοποιείται ένας βοηθητικός πίνακας `b`. Συγκρίνονται ένα προς ένα τα αντίστοιχα στοιχεία των δύο μισών πινάκων και κάθε φορά το μικρότερο τοποθετείται πριν το άλλο στον πίνακα `b`, μέχρι να φτάσουμε στο τέλος κάποιου από τα δύο μισά. Οι υπόλοιπες θέσεις του `b` συμπληρώνονται από τα στοιχεία του μισού που δεν έχουμε φτάσει στο τέλος του. Τέλος, η εντολή **`FOR`** αντιγράφει τον ταξινομημένο βοηθητικό πίνακα `b` στον `a` (copy back).

Κάθε εκτέλεση της διαδικασίας `merge` κάνει περίπου τόσες συγκρίσεις όσα είναι τα στοιχεία του πίνακα `b`, δηλαδή `last-first+1`. Με τη διαδικασία `mergesort` ένας πίνακας μήκους n υποδιαιρείται σε δύο πίνακες μήκους $n/2$ κ.ο.κ., δηλαδή τελειώνουμε σε $O(\log n)$ βήματα[1] όπου χρησιμοποιείται επανειλημμένα η `merge`. Στο i-οστό βήμα ο αρχικός πίνακας έχει χωριστεί σε 2^i μέρη τα οποία απαιτούν 2^{i-1} εκτελέσεις της `merge` για να συγχωνευθούν ανά δύο. Το πλήθος των συγκρίσεων που κάνουν όλες οι `merge` ενός βήματος είναι ίσο με n, αφού κάθε μία διατρέχει πλήρως τα δύο μέρη που της αναλογούν. Ο συνολικός αριθμός συγκρίσεων, επομένως, είναι $O(n \log n)$.

Θα δούμε τώρα ότι το $O(n \log n)$ αποτελεί και **κάτω όριο** (lower bound) της **πολυπλοκότητας** (complexity) της ταξινόμησης.

Θεώρημα: Οποιοσδήποτε αλγόριθμος που ταξινομεί (όλες τις διατάξεις από) n αριθμούς χρειάζεται τουλάχιστον $O(n \log n)$ συγκρίσεις.

Απόδειξη: Η ιδέα προέρχεται από τη **θεωρία πληροφορίας** (information theory). Με μια σύγκριση χωρίζονται όλες οι δυνατότητες στα δύο. Συνεπώς οι δυνατότητες μπορούν να παρασταθούν με κάποιο δυαδικό δέντρο αποφάσεων: π.χ. δες σχήμα για x_1, x_2, x_3. Ο ελάχιστος αριθμός συγκρίσεων είναι άρα το ελάχιστο δυνατό ύψος δυαδικού δέντρου που έχει στα φύλλα του όλες τις δυνατές μεταθέσεις ($n!$).
Δηλαδή: Πολυπλοκότητα $\geq \log n! \geq (n/4) \log n = O(n \log n)$ (για όλα τα $n > 4$). □



Ένας άλλος τρόπος να υλοποιηθούν οι `merge` και `mergesort`, που εκμεταλλεύεται την αντιστοιχία μεταξύ δεικτών και πινάκων της C και την αριθμητική δεικτών, είναι ο εξής:

[1] Ο δυαδικός λογάριθμος ενός αριθμού είναι (περίπου) ίσος με το μήκος της δυαδικής του παράστασης. Οταν διαιρούμε ένα δυαδικό αριθμό με 2 είναι σα να διαγράφουμε το τελευταίο του bit. Συνεπώς ο αριθμός διαιρέσεων που απαιτούνται για να μηδενιστεί ο αριθμός είναι ίσος με το μήκος του, δηλαδή με το δυαδικό του λογάριθμο.

```
PROC mergesort (int n, int *a)
  /* n: πλήθος των στοιχείων προς ταξινόμηση
      a: δείκτης στο τμήμα του πίνακα προς ταξινόμηση */
{
  int mid;

  if (n <= 1) return;

  mid = n/2;
  mergesort(mid, a);   mergesort(n-mid, a+mid);
  merge(a, a+mid, a+n);
}

PROC merge (int *first, int *mid, int *last)
{
  int b[last-first];
  int *i = first, *j = mid, *k = b;

  while (i < mid AND j < last) if (*i < *j) *k++ = *i++;
                               else         *k++ = *j++;

  while (i < mid)  *k++ = *i++;
  while (j < last) *k++ = *j++;

  i = first; j = b; while (j < k) *i++ = *j++;    // copy back
}
```

10.5 Ταξινόμηση με διαμέριση (quick sort)

Τέλος θα δώσουμε και την ιδέα της ταξινόμησης με διαμέριση (quick sort, Tony Hoare, 1960). Η ταξινόμηση γίνεται με εφαρμογή της διαδικασίας `partition` και αναδρομική εφαρμογή της διαδικασίας `quicksort` σε κάθε ένα από τα προκύπτοντα τμήματα, μέχρις ότου κάθε τμήμα γίνει αρκετά μικρό (αποτελείται από ένα μόνο στοιχείο).

```
PROC quicksort (int a[], int first, int last)
{
  int i;

  if (first >= last) return;

  i = partition(a, first, last);
  quicksort(a, first, i);  quicksort(a, i+1, last);
}
```

Με τη διαδικασία `partition` γίνονται τα εξής:

- Επιλέγεται μια κεντρική τιμή `x` των στοιχείων που θα ταξινομηθούν (στην παρακάτω υλοποίηση, το `x` είναι το μεσαίο στοιχείο του πίνακα, θα μπορούσε όμως να είναι π.χ. το πρώτο, το τελευταίο, ή και ένα τυχαίο στοιχείο).
- Τα στοιχεία διατρέχονται από αριστερά μέχρι να βρεθεί κάποιο στοιχείο `a[i] >= x` και από δεξιά μέχρι να βρεθεί κάποιο στοιχείο `a[j] <= x`. Τότε εναλλάσσεται η θέση τους.
- Η σάρωση από δεξιά και αριστερά και η εναλλαγή επαναλαμβάνονται μέχρις ότου οι δύο σαρώσεις συναντηθούν, δηλαδή μέχρι να γίνει `i>=j`. 'Ετσι προκύπτουν δύο τμήματα: ένα (από τη θέση `j` και αριστερά) με στοιχεία που είναι μικρότερα ή ίσα του `x`, και ένα (από τη θέση `i` και δεξιά) με στοιχεία που είναι μεγαλύτερα ή ίσα του `x`.

```
FUNC int partition (int a[], int first, int last)
{
  int x = a[(first + last)/2]; // επιλογή ενός στοιχείου pivot
  int i = first, j = last;

  while (true) {
    while (a[i] < x) i++;
    while (x < a[j]) j--;
    if (i >= j) break;
    swap(&(a[i]), &(a[j]));
    i++; j--;
  }
  return j;
}
```

Με τη μέθοδο quick sort, στη χειρότερη περίπτωση απαιτούνται $O(n^2)$ συγκρίσεις. Όμως, στη μέση περίπτωση απαιτούνται μόνο $O(n \log n)$ συγκρίσεις και — στην πράξη — η μέθοδος αυτή είναι συχνά γρηγορότερη από άλλες καθαρές μεθόδους ταξινόμησης πολυπλοκότητας $O(n \log n)$, όπως η merge sort.

Κεφάλαιο 11

Τεχνολογία λογισμικού

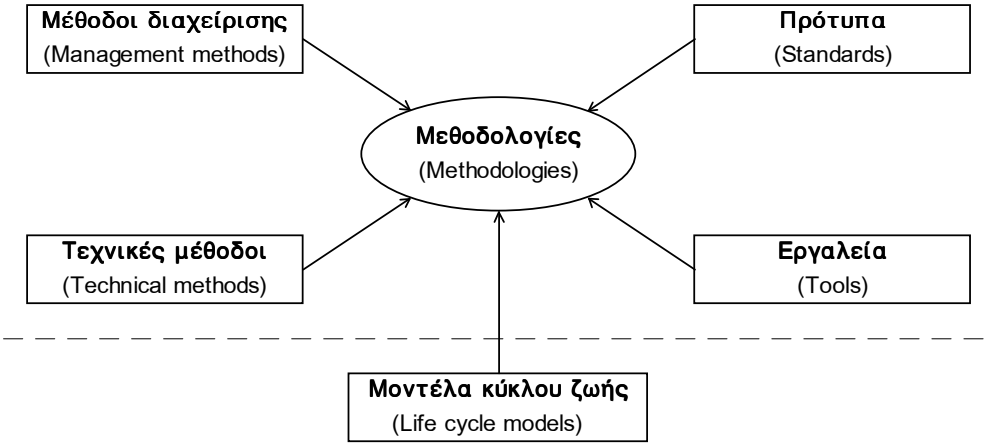
Η **τεχνολογία λογισμικού** (software engineering) είναι ο κλάδος της πληροφορικής που ασχολείται με την ανάπτυξη του λογισμικού. Σκοπός του κλάδου αυτού είναι η εύρεση τεχνικών και εργαλείων, των οποίων η χρησιμοποίηση στην παραγωγή λογισμικού θα εξασφαλίζει για το παραγόμενο προϊόν:

- παράδοση μέσα σε προδιαγεγραμμένα χρονικά όρια,
- κόστος μέσα σε προδιαγεγραμμένα όρια
- καλή ποιότητα
- αξιοπιστία, και
- δυνατή και όχι δαπανηρή συντήρηση.

Το λογισμικό θεωρείται αξιόπιστο όταν πληρεί τις προδιαγραφές του, δεν παράγει λανθασμένα αποτελέσματα, δεν αυτοκαταστρέφεται, προβαίνει σε λογικές ενέργειες όταν αντιμετωπίζει απρόβλεπτες καταστάσεις και σταματάει απρόβλεπτα μόνο όταν η παραπέρα συνέχιση της λειτουργίας του είναι εντελώς αδύνατη.

Η δημιουργία και ανάπτυξη του κλάδου της τεχνολογίας λογισμικού θεωρήθηκε απαραίτητη ήδη από το τέλος της δεκαετίας του 1960. Ο λόγος ήταν ότι, αντίθετα με την κατασκευή υλικού (hardware) στους ηλεκτρονικούς υπολογιστές που γίνεται με μεγάλη επιτυχία και αξιοπιστία και μικρό κόστος, η κατασκευή λογισμικού (software) παρουσιάζει σοβαρότατες δυσκολίες που πολλαπλασιάζονται δυσανάλογα, καθώς το μέγεθος του λογισμικού αυξάνει. Αποτέλεσμα είναι να σημειώνονται υπερβάσεις στο χρονοδιάγραμμα και τον προϋπολογισμό έργων λογισμικού και το παραγόμενο λογισμικό να είναι κακής ποιότητας, αναξιόπιστο και πολυδάπανο στη συντήρησή του.

Το ζητούμενο από την τεχνολογία λογισμικού είναι μια μεθοδολογία κατασκευής λογισμικού. Η ανάπτυξη όμως μιας τέτοιας μεθοδολογίας αποδείχθηκε μια πολύ δύσκολη υπόθεση. Σήμερα υπάρχουν πολλές μεθοδολογίες, καμιά όμως δεν είναι τέλεια και εφαρμόσιμη σε όλες τις περιπτώσεις. Οι περισσότερες από αυτές τις μεθοδολογίες θεωρούν το λογισμικό ως ένα βιομηχανικό προϊόν και περιγράφουν την κατασκευή του σε διακριτές φάσεις. Τα μοντέλα ανάπτυξης λογισμικού που βασίζονται σε αυτή τη φιλοσοφία ονομάζονται **μοντέλα κύκλου ζωής λογισμικού** (life cycle models). Το υπόβαθρο μιας μεθοδολογίας είναι πάντα ένα μοντέλο κύκλου ζωής. Οι μεθοδολογίες όμως εμπεριέχουν μεθόδους, τεχνικές και εργαλεία που δίνουν κατευθύνσεις για το πώς πρέπει να γίνουν οι εργασίες κάθε φάσης του κύκλου ζωής (βλέπε σχήμα 11.1).



Σχήμα 11.1. Μεθοδολογίες ανάπτυξης λογισμικού και τα συστατικά τους.

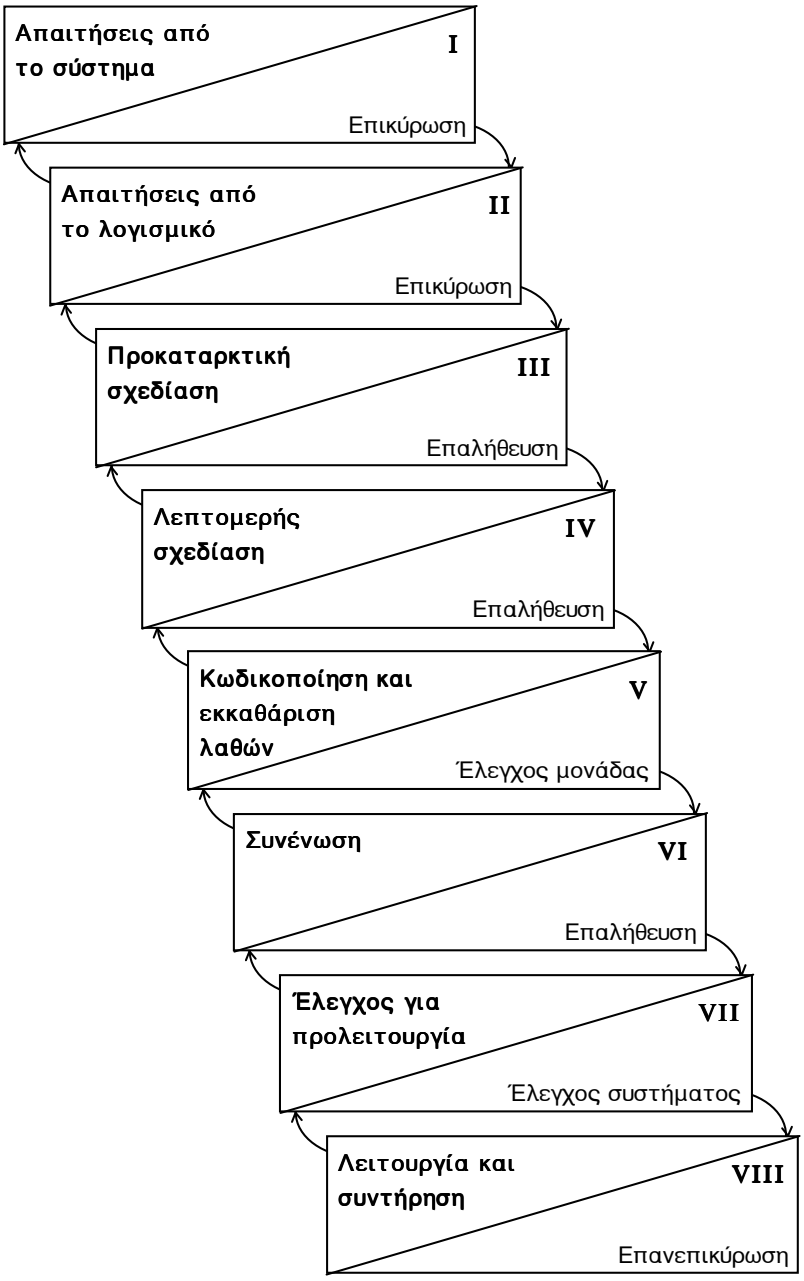
11.1 Μοντέλα κύκλου ζωής

Τα **μοντέλα κύκλου ζωής** (life cycle models) παρέχουν οδηγίες που αφορούν στις εργασίες που πρέπει να γίνουν σε ένα έργο λογισμικού. Οι οδηγίες καθορίζουν τη σειρά των εργασιών αυτών, τα κριτήρια μετάβασης από τη μια εργασία στην άλλη, τα ενδιάμεσα προϊόντα και τα κριτήρια τερματισμού μιας εργασίας. Έχουν αναπτυχθεί διάφορα μοντέλα κύκλου ζωής. Στο σχήμα 11.2 απεικονίζεται το **μοντέλο του καταρράκτη** (waterfall model), που χωρίζει τον κύκλο ζωής σε οκτώ φάσεις. Κάθε φάση επεξεργάζεται τα αποτελέσματα της προηγούμενης φάσης. Το τέλος κάθε φάσης σημαδεύεται από μια διαδικασία επικύρωσης ή επαλήθευσης, της οποίας σκοπός είναι να απαλείψει όσο γίνεται περισσότερα λάθη από τα προϊόντα αυτής της φάσης. Οπισθοδρομήσεις για διόρθωση λαθών περιορίζονται αν είναι δυνατό στην προηγούμενη μόνο φάση. Αν και έχει υποστεί μεγάλη κριτική, το μοντέλο αυτό με διάφορες παραλλαγές είναι εκείνο που χρησιμοποιείται στην πράξη σήμερα.

Πολλά άλλα μοντέλα έχουν προταθεί ως εναλλακτικές λύσεις. Το **μοντέλο της πρωτοτυποποίησης** (prototyping model) προτείνει την κατασκευή ενός πρωτότυπου του συστήματος όσο γίνεται πιο γρήγορα. Το πρωτότυπο εκτελεί ορισμένες μόνο από τις λειτουργίες του τελικού συστήματος λογισμικού και έχει ως σκοπό να αποκτηθεί εμπειρία που αφορά την εφικτότητα του συστήματος και την ακρίβεια των προδιαγραφών. Η κατασκευή του τελικού συστήματος λογισμικού μπορεί να γίνει μέσω της κατασκευής και αναθεώρησης πολλών τέτοιων πρωτοτύπων, καθένα από τα οποία βασίζεται στο προηγούμενο. Τέλος, το **αντικειμενοστρεφές μοντέλο** (object-oriented model) έχει περίπου τις ίδιες φάσεις όπως το μοντέλο του καταρράκτη, υποστηρίζει τον αντικειμενοστρεφή προγραμματισμό και την επαναχρησιμοποίηση λογισμικού.

11.1.1 Ανάλυση

Στη φάση (ή στις φάσεις, ανάλογα με τη μεθοδολογία που ακολουθείται) ανάλυσης προδιαγράφονται οι απαιτήσεις από το σύστημα λογισμικού, καθορίζεται δηλαδή τί θα κάνει συνολικά το σύστημα. Η απάντηση σε αυτό το ερώτημα είναι γενικά δύσκολο να βρεθεί και να καταγραφεί. Τα αποτελέσματα της φάσης ανάλυσης αποτελούν ένα είδος έγγραφου συμβολαίου ανάμεσα στον πελάτη και τον κατασκευαστή λογισμικού.



Σχήμα 11.2. Μοντέλο του καταρράκτη.

11.1.2 Σχεδίαση

Εξαιτίας του μεγέθους του λογισμικού, η κατασκευή του δεν μπορεί να γίνει χωρίς να προϋπάρξει κάποια φάση σχεδίασής του. Κατά τη φάση αυτή, το λογισμικό διαιρείται σε ενότητες, οι οποίες σχεδιάζονται και αργότερα κατασκευάζονται χωριστά. Οι δομικές αυτές μονάδες διαιρούνται με τη σειρά τους σε άλλες απλούστερες, έτσι ώστε το πολύπλοκο πρόβλημα της κατασκευής ολόκληρου του συστήματος λογισμικού να μετατραπεί σε περισσότερα απλούστερα προβλήματα. Το σχέδιο λογισμικού μπορεί να είναι είτε αρχιτεκτονικό/προκαταρκτικό (architectural/preliminary software design), οπότε είναι μια παράσταση της μακροσκοπικής δομής του λογισμικού, είτε λεπτομερές (detailed), οπότε αναπαριστά τη μικροσκοπική του δομή.

11.1.3 Κωδικοποίηση

Στη φάση της κωδικοποίησης μεταφράζονται οι περιγραφές του σχεδίου λογισμικού σε κώδικα. Σκοπός της φάσης δεν είναι μόνο η παραγωγή ορθού κώδικα, αλλά και ευανάγνωστου, εύκολου στην κατανόηση και καλά τεκμηριωμένου. Τέτοιος κώδικας διευκολύνει τον έλεγχο ορθότητας, τον εντοπισμό και εκκαθάριση λαθών καθώς και τη συντήρηση. Η φάση της κωδικοποίησης αναλίσκεται σε τρεις δραστηριότητες:

- γράψιμο του κώδικα
- τεκμηρίωση, και
- συνένωση και έλεγχος ορθότητας.

11.1.4 Έλεγχος

Η φάση του ελέγχου ορθότητας του κώδικα είναι ιδιαίτερα σημαντική. Ο κώδικας δεν είναι αρκετό να γραφεί, πρέπει και να ελεγχθεί η λειτουργία του μέσα στο σύστημα λογισμικού. Οι δομικές μονάδες, όπως προκύπτουν από τη σχεδίαση του λογισμικού, ελέγχονται κατ' αρχήν χωριστά και έπειτα γίνεται η συνένωση και ο έλεγχος του συνόλου. Ο εντοπισμός και η εκκαθάριση λαθών είναι εξαιρετικά δύσκολη εργασία και βασίζεται συνήθως στην εμπειρία και τη διαίσθηση των προγραμματιστών.

11.1.5 Συντήρηση

Παρότι το λογισμικό δε φθείρεται, έχει ανάγκη από συντήρηση. Ο όρος αυτός χρησιμοποιείται για να περιγράψει όλες τις δραστηριότητες που ακολουθούν την παράδοση του λογισμικού μέχρι την απόσυρσή του. Οι δραστηριότητες αυτές αποσκοπούν σε αλλαγές με σκοπό τη βελτίωση του λογισμικού, την προσαρμογή του σε νέα περιβάλλοντα και τη διόρθωσή του.

11.2 Μη τεχνικά ζητήματα

Όπως κάθε παραγωγική διαδικασία, έτσι και η τεχνολογία λογισμικού αναπόφευκτα ασχολείται και με ορισμένα μη τεχνικά ζητήματα, απαραίτητα όμως για τη σωστή και αποδοτική ανάπτυξη λογισμικού.

11.2.1 Διοίκηση και οργάνωση

Η διοίκηση ενός έργου λογισμικού αποσκοπεί στο να τηρηθούν οι προβλεφθείσες προθεσμίες παράδοσης του έργου, ο προβλεφθείς προϋπολογισμός και η προβλεφθείσα ποιότητα. Οι προβλέψεις γίνονται από την ίδια τη διοίκηση. Έπιμέρους δραστηριότητες της διοίκησης είναι η λήψη αποφάσεων, ο σχεδιασμός, ο ορισμός εργασιών, η επιτήρηση και η σύνταξη των εκθέσεων και των αναφορών. Η οργάνωση της ομάδας ανάπτυξης λογισμικού είναι επίσης ιδιαίτερα σημαντική, καθώς η καλή συνεργασία και επικοινωνία αυτής της ομάδας παίζει καθοριστικό ρόλο στην ομαλή αποπεράτωση του έργου.

11.2.2 Κοστολόγηση

Η εκτίμηση του κόστους του λογισμικού είναι ένα από τα δυσκολότερα και πλέον επιρρεπή σε λάθη προβλήματα της τεχνολογίας λογισμικού. Από τη μια πλευρά είναι απαραίτητο να γίνει μια ακριβής εκτίμηση του κόστους όσο το δυνατό νωρίτερα κατά την ανάπτυξη του λογισμικού. Από την άλλη όμως ακριβής εκτίμηση είναι πολύ δύσκολο να γίνει νωρίς γιατί είναι άγνωστοι πάρα πολλοί παράγοντες που τελικά συνεισφέρουν στην αύξηση του κόστους του λογισμικού και πολύ συχνά έχουν ως αποτέλεσμα την υπέρβαση του προϋπολογισμού ή του χρονοδιαγράμματος. Οι τεχνικές που έχουν αναπτυχθεί για την κοστολόγηση είναι ως επί το πλείστον εμπειρικές και τα αποτελέσματά τους αμφισβητήσιμα.

11.2.3 Εξασφάλιση ποιότητας

Το λογισμικό είναι ίσως το μοναδικό τεχνικό κατασκεύασμα που σήμερα πωλείται χωρίς εγγύηση. Ο βασικός λόγος είναι ότι η εξασφάλιση της ποιότητας του λογισμικού είναι ιδιαίτερα δύσκολη. Το πρόβλημα έγκειται στο ότι δεν υπάρχει ακόμα κοινά αποδεκτός ορισμός της ποιότητας, δεν έχει βρεθεί τεχνική που να βοηθά στην επιβολή της ποιότητας στα έργα λογισμικού ούτε αποτελεσματικός τρόπος για τη μέτρηση της ποιότητας ενός προϊόντος λογισμικού.

11.2.4 Διαχείριση σχηματισμών

Το σύνολο των οντοτήτων που αποτελούν το λογισμικό ονομάζεται σχηματισμός λογισμικού (software configuration). Στις οντότητες αυτές περιλαμβάνονται ο κώδικας, τα έγγραφα τεκμηρίωσης του λογισμικού (προδιαγραφές απαιτήσεων, περιγραφές σχεδίων, εγχειρίδια χρήσης και εγκατάστασης, κ.λπ.) και τα έγγραφα τεκμηρίωσης του έργου (πλάνο έργου, πλάνο ελέγχου ορθότητας, κ.λπ.). Κάθε οντότητα έχει πιθανώς πολλές εκδόσεις, λόγω των αλλαγών που γίνονται στην αρχική έκδοση (διόρθωση λαθών, αναθεωρήσεις, κ.λπ.), και κατά συνέπεια και όλο το λογισμικό έχει πολλές εκδόσεις. Η διαχείριση σχηματισμών (configuration management) ασχολείται με τη διαχείριση των οντοτήτων λογισμικού και όλων των εκδόσεών τους και έχει ως σκοπό οι διάφορες αλλαγές να μη δημιουργούν σύγχυση, παρερμηνείες και προβλήματα κατά την ανάπτυξη ενός έργου λογισμικού.

11.2.5 Διαπροσωπεία ανθρώπου-συστήματος

Σε κάθε σύστημα βασισμένο σε ηλεκτρονικό υπολογιστή είναι απαραίτητο να επιτευχθεί μια επικοινωνία ανάμεσα στον υπολογιστή και τους ανθρώπους που θα τον χρησιμοποιήσουν.

Το ρόλο του διερμηνέα παίζει η διαπροσωπεία ανθρώπου-συστήματος (man-machine interface), δηλαδή το τμήμα του υλικού και του λογισμικού που αναλαμβάνει τη σύνδεση του χρήστη με το υπολογιστικό σύστημα. Το λογισμικό που κατασκευάζεται οφείλει να είναι εύχρηστο και φιλικό προς το χρήστη (user friendly). Όμως, τόσο ο σχεδιασμός εύχρηστων και φιλικών διαπροσωπειών όσο και η υλοποίησή τους παρουσιάζει πολλές δυσκολίες.

11.3 Εργαλεία

Διάφορα εργαλεία διευκολύνουν το έργο των τεχνολόγων λογισμικού. Τα εργαλεία αυτά χρησιμοποιούνται σε διάφορες φάσεις της ανάπτυξης λογισμικού και κατατάσσονται στις εξής κατηγορίες:

- εργαλεία ανάλυσης και ελέγχου κώδικα,
- εργαλεία διαχείρισης, ελέγχου και συντήρησης,
- εργαλεία προδιαγραφών και ανάλυσης απαιτήσεων ή σχεδίου,
- εργαλεία κατασκευής και γέννησης προγραμμάτων,
- εργαλεία προσομοίωσης και μοντελοποίησης, και
- εργαλεία υποστήριξης και περιβάλλοντα προγραμματισμού.

11.4 Βιβλιογραφία

[1] A. Macro and J. Buxton. *The Craft of Software Engineering*. Addison-Wesley, 1987.

[2] S. L. Pfleeger and J. M. Atlee. *Software Engineering: Theory and Practice*. Prentice Hall, 4th edition, 2009.

[3] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. Mc-Graw Hill International, 7th edition, 2010.

[4] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2010.

[5] Ε. Σ. Σκορδαλάκης. *Εισαγωγή στην Τεχνολογία Λογισμικού*. Εκδόσεις Συμμετρία, 1991.

Κεφάλαιο 12

Επεξεργασία κειμένου

Μία από τις πιο διαδεδομένες χρήσεις του υπολογιστή είναι για επεξεργασία κειμένου.

Να πώς διαβάζουμε και επεξεργαζόμαστε χαρακτήρες — π.χ. τους μετράμε — μεχρι το τέλος του αρχείου. Η συνάρτηση `getchar` επιστρέφει `EOF` όταν προσπαθήσουμε να διαβάσουμε πέρα από το τέλος του αρχείου.

```
int n = 0;

while (getchar() != EOF) n++;
printf("%d characters were read.\n", n);
```

Στο επόμενο παράδειγμα αντιγράφουμε την είσοδο στην έξοδο, αποθηκεύοντας τους χαρακτήρες στη μεταβλητή `c`. Η μεταβλητή αυτή πρέπει να έχει τύπο **`int`** γιατί η τιμή `EOF` δεν μπορεί να αποθηκευθεί σε μεταβλητή τύπου **`char`**.

```
while (true) {
  int c = getchar();
  if (c == EOF) break;
  putchar(c);
}
```

Ισοδύναμα, το ίδιο τμήμα προγράμματος μπορεί να γραφεί ως εξής:

```
int c;

while ((c = getchar()) != EOF) putchar(c);
```

Ενώ ακεραίους (που αποθηκεύονται στη μεταβλητή `i`) διαβάζουμε μέχρι το τέλος του αρχείου και επεξεργαζόμαστε — π.χ. προσθέτουμε — ως εξής:

```
int i, sum = 0;

while (true) {
  if (scanf("%d", &i) != 1) break;
  sum += i;
}
```

Εδώ δεν μπορεί να χρησιμοποιηθεί η συνάρτηση `READ_INT` της Pazcal, γιατί αν προσπαθήσουμε με αυτήν να διαβάσουμε πέρα από το τέλος του αρχείου, θα προκύψει σφάλμα εκτέλεσης. Αντίθετα, η συνάρτηση `scanf` της βιβλιοθήκης της C επιστρέφει το πλήθος των μεταβλητών που διαβάστηκαν. Στο παραπάνω παράδειγμα το αποτέλεσμά της είναι 1 (μόνο το `i` διαβάστηκε) εκτός αν φτάσαμε στο τέλος του αρχείου.

Το ίδιο παράδειγμα μπορεί να γραφεί ισοδύναμα ως εξής:

```
int i, sum = 0;

while (scanf("%d", &i) == 1) sum += i;
```

12.1 Χαρακτήρες και γραμμές

Το παρακάτω κομμάτι προγράμματος διαβάζει ένα κείμενο από την είσοδο, μετράει τον αριθμό χαρακτήρων και τον αριθμό γραμμών, υπολογίζει τον μέσο όρο μήκους της γραμμής και παρουσιάζει τα αποτελέσματα στην οθόνη.

```
int lines = 0, chars = 0;

while (true) {
  int c = getchar();
  if (c == EOF) break;
  if (c == '\n') lines++; else chars++;
}

printf("%d lines were read\n", lines);
if (lines > 0) printf("%0.3lf characters per line\n", 1.0 * chars / lines);
```

Προσέξτε ότι σε αυτό το παράδειγμα δε μετράμε τις αλλαγές γραμμής ως χαρακτήρες.

Εάν το κείμενο δεν είναι σε αρχείο, αλλά δίνεται από το χρήστη κατευθείαν από το πληκτρολόγιο, τότε ο χρήστης πρέπει να πατήσει <ctrl>-d για να σημάνει το τέλος του αρχείου.

Στο παραπάνω παράδειγμα, προσέξτε την έκφραση `1.0 * chars / lines`. Ο πολλαπλασιασμός με το `1.0` χρειάζεται ώστε η τιμή της μεταβλητής `chars` να μετατραπεί σε πραγματικό αριθμό, έτσι ώστε στη συνέχεια να γίνει πραγματική (και όχι ακέραια) διαίρεση. Το ίδιο μπορεί να γίνει στη C με **μετατροπή τύπου** (type cast):

```
(double) chars / lines
```

12.2 Λέξεις

Τα επόμενα παραδείγματα αφορούν λέξεις. Π.χ. μία λέξη ορίζεται ως πεπερασμένη ακολουθία γραμμάτων του αλφαβήτου που ακολουθείται από οποιονδήποτε χαρακτήρα που δεν είναι γράμμα. Θα χρειαστούμε λοιπόν έναν έλεγχο για το αν ένας χαρακτήρας που διαβάστηκε είναι γράμμα ή όχι.

```
FUNC bool isletter (char c)
{
  return c >= 'a' AND c <= 'z' OR c >= 'A' AND c <= 'Z';
}
```

Το παρακάτω τμήμα προγράμματος μετράει χαρακτήρες, λέξεις και γραμμές. Προσέξτε ότι εδώ μετράμε ακόμα και τις αλλαγές γραμμής ως χαρακτήρες.

```
int c, lines = 0, chars = 0, words = 0;

c = getchar();
while (c != EOF)
  if (isletter(c)) {
    words++;
    do { chars++; c = getchar(); } while (isletter(c));
  }
  else {
    chars++;
    if (c == '\n') lines++;
    c = getchar();
  }
```

Στο παραπάνω παράδειγμα, είναι βασική **αναλλοίωτη** του εξωτερικού βρόχου **while** ότι έχουμε διαβάσει έναν χαρακτήρα παραπάνω (μπροστά) από την είσοδο, ο οποίος βρίσκεται στη μεταβλητή c (αν φυσικά υπήρχε, διαφορετικά η μεταβλητή c περιέχει την τιμή EOF). Αυτό γίνεται γιατί δεν υπάρχει τρόπος να καταλάβουμε ότι μία λέξη τελείωσε, εκτός αν διαβάσουμε τον επόμενο από αυτήν χαρακτήρα και διαπιστώσουμε ότι δεν είναι γράμμα.

Το παρακάτω παράδειγμα διαβάζει ένα κείμενο από την είσοδο και, για κάθε λέξη του, μετράει πόσα γράμματα αυτή έχει. Συγχρόνως, μετράει πόσες λέξεις υπάρχουν με πλήθος γραμμάτων από 1 μέχρι 20 και στο τέλος εμφανίζει τα αποτελέσματα.

```
int i, c, freq[21];

FOR (i, 1 TO 20) freq[i] = 0;

c = getchar();
while (c != EOF)
  if (isletter(c)) {
    int n = 0;
    do { n++; c = getchar(); } while (isletter(c));
    if (n <= 20) freq[n]++;
  }
  else c = getchar();

FOR (i, 1 TO 20) printf("%4d words of length %2d\n", freq[i], i);
```

Συνάρτηση μετατροπής κεφαλαίων γραμμάτων σε μικρά:

```
FUNC char tolower (char ch)
{
  if (ch >= 'A' AND ch <= 'Z') return ch - 'A' + 'a';
  else                         return ch;
}
```

Συνάρτηση μετατροπής μικρών γραμμάτων σε κεφαλαία:

```
FUNC char toupper (char ch)
{
  if (ch >= 'a' AND ch <= 'z') return ch - 'a' + 'A';
  else                         return ch;
}
```

12.3 Αντιστροφή χαρακτήρων κάθε γραμμής

Το παρακάτω τμήμα προγράμματος διαβάζει ένα κείμενο από την είσοδο και εμφανίζει τους χαρακτήρες κάθε γραμμής σε αντίστροφη σειρά. Εδώ πρέπει να αποθηκευθούν οι χαρακτήρες κάθε γραμμής, όπως διαβάζονται, και κάνουμε την υπόθεση ότι το μέγιστο πλήθος χαρακτήρων σε μία γραμμή είναι ίσο με MAX.

```
const int MAX = 80;

int i, c, line[MAX];

while ((c = getchar()) != EOF) {
  int n = 0;
  while (c != '\n') { line[n++] = c; c = getchar(); }
  FOR (i, n-1 DOWNTO 0) putchar(line[i]);
  putchar('\n');
}
```

12.4 Αναζήτηση λέξεων

Αλλα προγράμματα επεξεργασίας ψάχνουν για διάφορες λέξεις-κλειδιά των οποίων τις εμφανίσεις μετρούν ή αντικαθιστούν με κάτι άλλο, κάνουν διάφορες στατιστικές για το κείμενο ή αλλάζουν τη μορφή του κειμένου κατα κάποιον τρόπο π.χ. μήκος γραμμών, κεφαλαία-μικρά, πλήθος κενών, κενές γραμμές, παράγραφοι, στοίχιση δεξιού περιθωρίου κ.λπ.

Παράδειγμα εύρεσης λέξης-κλειδιού σε κείμενο:

```
...

// η λέξη-κλειδί έχει 3 χαρακτήρες
FOR (j, 0 TO 2) key[j] = getchar();

...

// έστω i το μήκος της γραμμής
FOR (k, 0 TO i-3)
  if (line[k] == key[0] AND line[k+1] == key[1] AND line[k+2] == key[2])
     WRITELN(”keyword found!”);
```

Κεφάλαιο 13

Δομημένοι τύποι

Σε αυτό το κεφάλαιο παρουσιάζονται μερικοί ακόμα σύνθετοι τύποι δεδομένων, οι οποίοι περιέχουν ως δομικά συστατικά άλλους τύπους και γι' αυτό αποκαλούνται **δομημένοι τύποι**. Η περιγραφή αναφέρεται στους τύπους συμβολοσειρών, εγγραφών, ενώσεων και αρχείων.

13.1 Συμβολοσειρές (strings)

Οι **συμβολοσειρές** (strings) είναι πίνακες χαρακτήρων (δηλαδή, ισοδύναμα όπως εξηγήθηκε σε προηγούμενο κεφάλαιο, δείκτες σε χαρακτήρες). Κατά σύμβαση, το τέλος μίας συμβολοσειράς σηματοδοτείται από τον ειδικό χαρακτήρα `'\0'`, δηλαδή το χαρακτήρα με ASCII κωδικό ίσο με μηδέν. Στη δήλωση των μεταβλητών συμβολοσειρών όπου δεσμεύεται ο χώρος αποθήκευσής τους πρέπει να υπολογίζεται το μέγιστο πλήθος χαρακτήρων που μπορεί να περιέχουν αυτές οι συμβολοσειρές (συμπεριλαμβανομένου και του τελικού χαρακτήρα `'\0'`).

Παράδειγμα δήλωσης

```
char name[30], address[80];
```

Ορίζονται δυο μεταβλητές συμβολοσειρές: η `name` που έχει μήκος το πολύ 30 χαρακτήρες, και η `address` που έχει μήκος το πολύ 80 χαρακτήρες.

Παράδειγμα ανάγνωσης και εκτύπωσης

```
printf("What's your name?\n");
scanf("%s", name);
printf("Hi %s, how are you?\n", name);
```

13.1.1 Προκαθορισμένες συναρτήσεις

Για να χρησιμοποιηθούν οι παρακάτω συναρτήσεις πρέπει να προηγηθεί στην αρχή του προγράμματος:

```
#include <string.h>
```

- `strlen(s)`: επιστρέφει το μήκος της συμβολοσειράς `s`.

  Παράδειγμα

  ```
  n = strlen("ABCDEF");     // n = 6
  ```

- `strstr(s, key)`: επιστρέφει `NULL` αν η συμβολοσειρά `key` δεν εμφανίζεται στη συμβολοσειρά `s`, διαφορετικά επιστρέφει ένα δείκτη στον πρώτου χαρακτήρα της πρώτης εμφάνισης της `key` στην `s`.

  Παράδειγμα

  ```
  char s[] = "It is raining today";
  char key[] = "aining";
  char *p = strstr(s, key);      // p = s+7
  ```

- `strcat(t, s)`: προσθέτει (συνενώνει) τη συμβολοσειρά `s` στο τέλος της συμβολοσειράς `t`. Η τελευταία πρέπει να διαθέτει αρκετό χώρο έτσι ώστε να μπορεί να χωρέσει το αποτέλεσμα της συνένωσης (concatenation).

  Παράδειγμα

  ```
  char a[10] = "abc";
  strcat(a, "def");// a = "abcdef"
  ```

- `strcpy(t, s)`: αντιγράφει τη συμβολοσειρά `s` στη συμβολοσειρά `t`. Η τελευταία πρέπει να διαθέτει αρκετό χώρο έτσι ώστε να μπορεί να χωρέσει την πρώτη συμβολοσειρά.

  Παράδειγμα

  ```
  char a[10];
  strcpy(a, "ding");
  a[1] = 'o';
  printf("%s\n", a);// dong
  ```

- `strcmp(s1, s2)`: συγκρίνει τις δύο συμβολοσειρές σύμφωνα με τη **λεξικογραφική διάταξη** και επιστρέφει 0 αν είναι ίσες, θετικό αριθμό αν η `s1` είναι μεγαλύτερη της `s2`, ή αρνητικό αριθμό αν η `s1` είναι μικρότερη της `s2`.

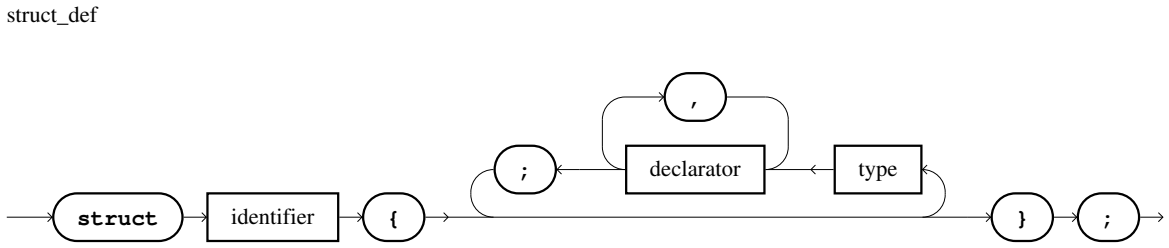
  Παράδειγμα

  ```
  if (strcmp(name, "John") == 0)
    printf("Hi John, how's Mary?\n");

  n = strcmp("ding", "dong");          // n < 0
  ```

13.2 Δομές (structures)

Ο τύπος της **δομής** (structure) είναι ο πιο σύνθετος τύπος δεδομένων της Pazcal και της C. Μια δομή αποτελείται από ένα πλήθος επιμέρους μεταβλητών, κάθε μια από τις οποίες μπορεί να είναι οποιουδήποτε έγκυρου τύπου, ακόμα και άλλη δομή. Αυτές οι επιμέρους μεταβλητές ονομάζονται **πεδία** (fields) της δομής.

Η δήλωση του τύπου μίας δομής δίνεται από το παρακάτω συντακτικό διάγραμμα:



Οι δομές χρησιμοποιούνται για την ομαδοποίηση των δεδομένων ενός προβλήματος που αποτελούν λογικές οντότητες. Για παράδειγμα, σε ένα πρόγραμμα διαχείρισης του αρχείου ενός σχολείου, απαιτείται για κάθε μαθητή να ξέρουμε το ονοματεπώνυμό του, τη διεύθυνσή του, το τηλέφωνό του, την τάξη και το τμήμα του, τη βαθμολογία του, κ.λπ. Τα δεδομένα αυτά, κάθε ένα από τα οποία μπορεί να παρασταθεί με μια μεταβλητή κατάλληλου τύπου, μπορούν να οργανωθούν σε μια δομή για κάθε μαθητή. Ο τύπος αυτής της δομής θα μπορούσε να είναι:

```
struct student_t {
  char firstName[20];  // όνομα
  char lastName[30];   // επώνυμο
  int  class, room;    // τάξη, τμήμα
  int  grade[15];      // βαθμοί μαθημάτων
};
```

Αν θέλουμε να δηλώσουμε μια μεταβλητή αυτού του τύπου, μπορούμε να γράψουμε:

```
struct student_t s;
```

Η προσπέλαση στα πεδία των δομών γίνεται με το συμβολισμό:

δομή.*μέλος*

Για παράδειγμα, μπορούμε να εκτυπώσουμε στην οθόνη το ονοματεπώνυμο του μαθητή `s` με την εντολή:

```
WRITESPLN(s.firstName, s.lastName);
```

και με παρόμοιο τρόπο μπορούμε να εκχωρήσουμε τον αριθμό 3 στο πεδίο που παριστάνει την τάξη του παραπάνω μαθητή:

```
s.class = 3;
```

Ο τύπος της δομής, όπως οι περισσότεροι τύποι δεδομένων, μπορεί να χρησιμοποιηθεί ως τύπος παραμέτρου ενός υποπρογράμματος ή ως τύπος αποτελέσματος συνάρτησης. Ως ένα πιο σύνθετο παράδειγμα, η παρακάτω συνάρτηση υπολογίζει το μέσο όρο της βαθμολογίας του μαθητή s που δίνεται ως παράμετρος.

```
FUNC REAL average (struct student_t s)
{
  REAL sum = 0.0;
  int i;

  FOR (i, 0 TO 14) sum += s.grade[i];
  return sum / 15;
}
```

Μια δομή μπορεί να περιέχει ως πεδίο της μια άλλη δομή. Για να δούμε ένα τέτοιο παράδειγμα, θα προσθέσουμε στον τύπο **struct** student_t την ημερομηνία γέννησης του μαθητή. Πρώτα θα ορίσουμε ένα νέο τύπο δομής για την αναπαράσταση των ημερομηνιών. Θα ορίσουμε επίσης ένα συνώνυμο τύπου για συντομία.

```
struct date_t {
  int day, month, year;
};

typedef struct date_t date;
```

Στη συνέχεια θα αλλάξουμε τον ορισμό του **struct** student_t προσθέτοντας ένα ακόμα πεδίο.

```
struct student_t {
  char firstName[20];  // όνομα
  char lastName[30];   // επώνυμο
  int  class, room;    // τάξη, τμήμα
  int  grade[15];      // βαθμοί μαθημάτων
  data birthDate;      // ημερομηνία γέννησης
};
```

Αν τώρα η μεταβλητή s είναι του παραπάνω τύπου, για να παρουσιάσουμε με ευανάγνωστο τρόπο την ημερομηνία γέννησης μπορούμε να χρησιμοποιήσουμε την ακόλουθη εντολή:

```
WRITELN(s.birthDate.day, "/", s.birthDate.month, "/", s.birthDate.year);
```

Προσέξτε ότι υπάρχουν δυο τελείες σε κάθε πεδίο που εκτυπώνεται: κατ' αρχήν βρίσκουμε την ημερομηνία γέννησης του μαθητή, δηλαδή s.birthDate, και μετά π.χ. την ημέρα του μήνα που αντιστοιχεί σε αυτήν, δηλαδή s.birthDate.day.

Η Pazcal και η C δεν έχουν προκαθορισμένο τύπο για τους μιγαδικούς αριθμούς. Μπορούμε όμως να ορίσουμε για αυτό το σκοπό έναν τύπο δομής:

```
struct complex_t { REAL re, im; };

typedef struct complex_t complex;
```

Στη συνέχεια, μπορούμε να ορίσουμε σύνθετες πράξεις με μιγαδικούς αριθμούς ως συναρτήσεις που δέχονται δεδομένα του τύπου `complex` ως παραμέτρους. Για παράδειγμα, ο πολλαπλασιασμός μιγαδικών αριθμών μπορεί να υλοποιηθεί ως εξής:

```
FUNC complex cMult (complex x, complex y)
{
  complex result;
  result.re = x.re * y.re - x.im * y.im;
  result.im = x.re * y.im + x.im * y.re;
  return result;
}
```

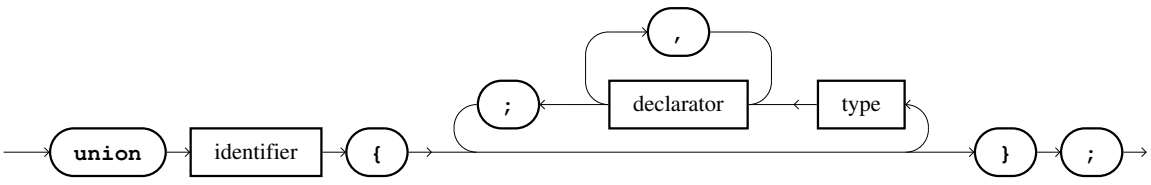
ενώ η συνάρτηση που υπολογίζει το μέτρο ενός μιγαδικού αριθμού μπορεί να γραφεί ως:

```
FUNC REAL cNorm (complex x)
{
  return sqrt(x.re * x.re + x.im * x.im);
}
```

13.3 Ενώσεις (unions)

Ο τύπος της **ένωσης** (union) έχει ακριβώς την ίδια μορφή με τον τύπο της δομής. Όμως, ενώ σε μία δομή μπορεί κανείς να χρησιμοποιήσει συγχρόνως όλα τα πεδία της, σε μία ένωση μπορεί να χρησιμοποιήσει το πολύ ένα κάθε στιγμή!

union_def



Οι ενώσεις χρησιμοποιούνται για να εξοικονομείται χώρος στη μνήμη, όταν π.χ. γνωρίζουμε ότι κάποιο δεδομένο θα είναι είτε ακέραιος αριθμός είτε πραγματικός (και θέλουμε να χρησιμοποιήσουμε αναπαράσταση **`int`** και **`REAL`** αντίστοιχα), αλλά δε γνωρίζουμε ποιο από τα δύο.

```
union number_t { int i; REAL r; };

union number_t n;
```

Μπορούμε π.χ. να ξεκινήσουμε να χρησιμοποιούμε το πεδίο `r` της ένωσης, με την παρακάτω ανάθεση:

```
n.r = 1.2;
```

```
printf("%lf\n", n.r);
```

και αμέσως μετά μπορούμε να αναθέσουμε εκ νέου και να αρχίσουμε να χρησιμοποιούμε το πεδίο `i` της ένωσης:

```
n.i = 42;
printf("%d\n", n.i);
```

όμως, τότε θα είναι λάθος να προσπαθήσουμε να εκτυπώσουμε το πεδίο `r`!

```
printf("%lf\n", n.r);    // λάθος!
```

13.4 Αρχεία (files)

Τα **αρχεία** (files) είναι δομημένοι τύποι μεταβλητού μεγέθους που αποτελούνται από στοιχεία πληροφορίας (bytes), αποθηκευμένα το ένα μετά το άλλο. Τις περισσότερες φορές, τα στοιχεία αυτά είναι αποθηκευμένα στην περιφερειακή μνήμη του υπολογιστή, δηλαδή σε μια δισκέτα ή στο σκληρό δίσκο. Είναι πολύ συνηθισμένη η χρήση αρχείων κειμένου, δηλαδή αρχείων στα οποία τα bytes του αρχείου κωδικοποιούν χαρακτήρες, π.χ. σύμφωνα με την κωδικοποίηση ASCII. Όμως, για κάθε τύπο δεδομένων είναι δυνατή η κωδικοποίηση αυτών με bytes τα οποία μπορούν να αποθηκευθούν σε αρχεία, με ευθύνη του προγραμματιστή.

Ένας τρόπος προσπέλασης των αρχείων είναι ο *σειριακός* (sequential), δηλαδή για να προσπελάσουμε κάποιο στοιχείο πρέπει να ξεκινήσουμε από την αρχή του αρχείου και να διατρέξουμε όλα τα προηγούμενα. Ιστορικά, τα αρχεία αντικατοπτρίζουν τη δυνατότητα αποθήκευσης πληροφοριών σε μαγνητικά μέσα, π.χ. ταινία, για τα οποία μόνο ο σειριακός τρόπος προσπέλασης είναι εφικτός. Ένας άλλος τρόπος προσπέλασης, εφικτός για αρχεία που αποθηκεύονται στους σύγχρονους υπολογιστές, είναι με *τυχαία προσπέλαση* (random access), όταν υπάρχει η δυνατότητα να μεταβούμε σε οποιοδήποτε στοιχείο ενός αρχείου σε σταθερό χρόνο — όπως συμβαίνει και με τους πίνακες. Στη συνέχεια θα επικεντρωθούμε στα αρχεία σειριακής προσπέλασης, τα οποία είναι και τα απλούστερα στη χρήση.

13.4.1 Αρχεία εισόδου και εξόδου

Τα αρχεία που χρησιμοποιεί ένα πρόγραμμα διακρίνονται σε αρχεία εισόδου και αρχεία εξόδου. Με τον όρο **είσοδος από κάποιο αρχείο** εννοούμε το διάβασμα πληροφοριών που είναι αποθηκευμένες στο αρχείο και την αποθήκευση αυτών σε μεταβλητές του προγράμματος. Με τον όρο **έξοδος σε κάποιο αρχείο** εννοούμε το γράψιμο στο αρχείο πληροφοριών, που αρχικά βρίσκονται σε μεταβλητές του προγράμματος. Επομένως, από τα αρχεία εισόδου επιτρέπεται μόνο η είσοδος στοιχείων στο πρόγραμμα, ενώ στα αρχεία εξόδου επιτρέπεται μόνο η έξοδος.

Η χρήση ενός αρχείου από ένα πρόγραμμα γίνεται μέσω μίας μεταβλητής με τύπο `FILE *` (προσοχή, όχι `FILE` αλλά δείκτης σε `FILE`).

Παράδειγμα δήλωσης:

```
#include <stdio.h>

...

FILE *f;
```

Πριν χρησιμοποιηθεί ένα αρχείο, είτε για διάβασμα είτε για γράψιμο, πρέπει να "ανοιχθεί", δηλαδή να αρχικοποιηθεί με κατάλληλο τρόπο ώστε να μπορούν να χρησιμοποιηθούν οι πληροφορίες που περιέχει. Αυτό γίνεται με τη συνάρτηση `fopen` που δέχεται δύο συμβολοσειρές ως παραμέτρους. Η πρώτη είναι το όνομα του αρχείου που θέλουμε να ανοίξουμε. Η δεύτερη καθορίζει δύο πράγματα:

1. Αν πρόκειται για αρχείο εισόδου (`"r"`) ή για αρχείο εξόδου (`"w"`). Στην πρώτη περίπτωση, μετά το άνοιγμα του αρχείου μπορούν να αρχίσουν να διαβάζονται στοιχεία ξεκινώντας από την αρχή. Στη δεύτερη περίπτωση, αν το αρχείο δεν υπάρχει τότε δημιουργείται, ενώ αν υπάρχει και περιέχει κάποια στοιχεία τότε αυτά σβήνονται και το αρχείο θεωρείται αρχικά άδειο. Υπάρχει επίσης η δυνατότητα ανοίγματος ενός αρχείου εξόδου διατηρώντας τα υπάρχοντα περιεχόμενά του και προσθέτοντας στο τέλος (append, `"a"`).

2. Αν πρόκειται για αρχείο κειμένου (`"t"`) ή για δυαδικό αρχείο (binary file), δηλαδή για αρχείο που κωδικοποιεί δεδομένα οποιουδήποτε τύπου με ευθύνη του προγραμματιστή (`"b"`).

Η συνάρτηση `fopen` επιστρέφει ένα `FILE *` που αντιστοιχεί στο αρχείο που μόλις ανοίξαμε. Αν το άνοιγμα δεν επιτύχει (π.χ. γιατί ένα αρχείο εισόδου δεν υπάρχει ή γιατί απαγορεύεται η πρόσβαση σε αυτό), τότε η `fopen` επιστρέφει `NULL`.

Αφού επεξεργαστούμε τα στοιχεία ενός αρχείου, πρέπει να καλέσουμε τη συνάρτηση `fclose` για να το κλείσουμε. Ειδικά για τα αρχεία εξόδου, αυτό είναι πολύ σημαντικό καθώς αν δεν κλείσουμε ένα αρχείο ενδέχεται να χαθούν κάποια από τα δεδομένα που έχουμε γράψει σε αυτό.

Παράδειγμα ανοίγματος και κλεισίματος:

```
#include <stdio.h>

...

FILE *f, *g;

f = fopen("some-input-text-file.txt", "rt");
g = fopen("some-output-binary-file.txt", "wb");

if (f == NULL OR g == NULL) {
  printf("Some of the files could not be opened!\n");
  return;
}
```

```
...

fclose(f);  fclose(g);
```

13.4.2 Χρήση των αρχείων

Για το γράψιμο σε αρχεία και το διάβασμα από αρχεία, αντίστοιχα, μπορούν να χρησιμοποιηθούν οι παρακάτω συναρτήσεις, αναλόγως τι θέλουμε να γράψουμε ή να διαβάσουμε:

- `fputc` και `fgetc`: για χαρακτήρες
- `fputs` και `fgets`: για συμβολοσειρές
- `fprintf` και `fscanf`: για οτιδήποτε μπορεί να μορφοποιηθεί, όπως με τις `printf` και `scanf`.
- `fwrite` και `fread`: για ακολουθίες byte

Επίσης, με τη συνάρτηση `feof` μπορούμε να ελέγξουμε αν κατά τη προηγούμενη εντολή διαβάσματος από κάποιο αρχείο εισόδου βρέθηκε το τέλος του αρχείου.

Παράδειγμα 1

Το πρόγραμμα που ακολουθεί (σε C) χρησιμοποιεί δύο αρχεία κειμένου. Το `f` είναι αρχείο εισόδου ενώ το `g` είναι αρχείο εξόδου. Το πρόγραμμα διαβάζει διαδοχικά τα περιεχόμενα του `f` και τα αποθηκεύει (αντιγράφει) στο `g`, μετατρέποντας τα μικρά γράμματα σε κεφαλαία. Για το σκοπό αυτό χρησιμοποιείται η συνάρτηση βιβλιοθήκης `toupper` που ορίζεται στο `ctype.h` (βλ. και σελ. 146).

```
#include <stdio.h>
#include <ctype.h>

int main ()
{
  FILE *f, *g;
  int c;

  if ((f = fopen("some-input-file.txt", "rt")) == NULL) {
    printf("The input file could not be opened!\n");
    return 1;
  }
  if ((g = fopen("some-output-file.txt", "wt")) == NULL) {
    printf("The output file could not be opened!\n");
    return 1;
  }

  while ((c = fgetc(f)) != EOF) fputc(c, g);
```

```
  fclose(f); fclose(g);
  return 0;
}
```

Στην αρχή του προγράμματος ανοίγουμε τα αρχεία `f` και `g`, ενώ στο τέλος τα κλείνουμε. Το κύριο μέρος του προγράμματος είναι ένας βρόχος **`while`** που επαναλαμβάνεται μέχρι να εξαντληθούν τα στοιχεία του `f`.

Παράδειγμα 2

Μια ειδική κατηγορία αρχείων είναι τα **αρχεία κειμένου** (text files). Τα αρχεία κειμένου είναι ουσιαστικά αρχεία χαρακτήρων. Κάθε αρχείο κειμένου αποτελείται από γραμμές. Στο τέλος κάθε γραμμής (συμπεριλαμβανομένης φυσικά και της τελευταίας) υπάρχει ο ειδικός χαρακτήρας `'\n'`.

Το πρόγραμμα που ακολουθεί μετρά τον αριθμό των γραμμών και των χαρακτήρων πολλών αρχείων, τα ονόματα των οποίων δίνονται από το χρήστη στη **γραμμή εντολών** κατά την κλήση του προγράμματος. Αυτό γίνεται με τις παραμέτρους `argc` και `argv` της κύριας συνάρτησης (`main`).

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int i;

  for (i=1; i<argc; i++) {
    // επεξεργασία του αρχείου argv[i]

    FILE *f;
    int chars = 0, lines = 0, c;

    if ((f = fopen(argv[i], "rt")) == NULL) return 1;

    while ((c = fgetc(f)) != EOF) {
      chars++;
      if (c == '\n') lines++;
    }
    fclose(f);
    printf("%d chars, %d lines, %s\n", chars, lines, argv[i]);
  }

  return 0;
}
```

Παράδειγμα 3

Το πρόγραμμα που ακολουθεί χρησιμοποιεί δυο δυαδικά αρχεία και αντιγράφει τα περιεχόμενα του ενός στο άλλο, διαβάζοντας (το πολύ) 1000 bytes κάθε φορά. Ανατρέξτε στην περιγραφή των συναρτήσεων `fread` και `fwrite` (στα man pages του υπολογιστή σας ή στο internet) για να δείτε πώς λειτουργεί.

```
#include <stdio.h>

int main (int argc, char * argv[])
{
  FILE *fin, *fout;

  fin = fopen(argv[1], ”rb”);
  if (fin == NULL) return 1;

  fout = fopen(argv[2], ”wb”);
  if (fout == NULL) return 2;

  while (!feof(fin)) {
    unsigned char buffer[1000];
    unsigned int count;

    count = fread(buffer, 1, 1000, fin);
    fwrite(buffer, 1, count, fout);
  }

  fclose(fin);
  fclose(fout);
  return 0;
}
```

Κεφάλαιο 14

Δείκτες και συνδεδεμένες λίστες

Επειδή συχνά κατά την παράλληλη εκτέλεση πολλών εργασιών (tasks) στον υπολογιστή ο κατακερματισμός της μνήμης είναι αναπόφευκτος, καθίσταται αναγκαία η "σύνδεση" των μη συνεχόμενων περιοχών της, για την καλύτερη αξιοποίησή τους.

Αυτό επιτυγχάνεται με τη δημιουργία συνδεδεμένης λίστας που με τη σειρά της υλοποιείται με τη χρήση του τύπου pointer (δείκτη).

Οι συνδεδεμένες λίστες λύνουν δύο από τα προβλήματα που συναντήσαμε με τους γραμμικούς πίνακες:

Πρώτον, οι πίνακες έχουν προκαθορισμένο μέγεθος, ενώ οι λίστες μπορούν να επεκταθούν απεριόριστα (όπως και τα αρχεία).

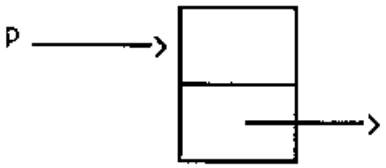
Δεύτερον στους πίνακες είναι δύσκολη η προσθήκη νέων στοιχείων σε κάποια ενδιάμεση θέση, διότι για να μην αλλαχτεί η σειρά των παλιών στοιχείων πρέπει να γίνει κάποια ολίσθηση (π.χ. προς τα δεξιά). Το ίδιο ισχύει και στην περίπτωση διαγραφής στοιχείων: δε θέλουμε κενές θέσεις στη δομή μας. Στις λίστες αντιθέτως η προσθήκη ή διαγραφή στοιχείων είναι αρκετά εύκολη.

Η τιμή μιας μεταβλητής τύπου pointer είναι η διεύθυνση μιας περιοχής της μνήμης όπου βρίσκεται η τιμή κάποιου τύπου. Η τιμή αυτή είναι συνήθως μία δομή που μπορεί να περιέχει, εκτός των άλλων, και κάποιον pointer.

Παράδειγμα:

```
struct node_t {
  int info;
  struct node_t *next;
};

typedef struct node_t node, *nodeptr;
```

Η μεταβλητή `p` τύπου `nodeptr` δείχνει τη "διεύθυνση" μιας μεταβλητής τύπου `node`:



Να πώς γίνεται η ανάθεση τιμών:

```
p->info = 17;
p->next = NULL;
```

Ο τελεστής `->` είναι συντομογραφία: `p->x` ισοδυναμεί με `(*p).x`. Η τιμή `NULL` είναι μια ειδική τιμή δείκτη που σημαίνει "πουθενά".

Άλλο παράδειγμα:

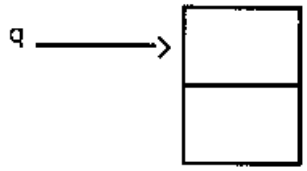
```
typedef REAL * ptr;
ptr p;
```

Ανάθεση: `*p = 7.628` (και όχι `p = 7.628`!). Ανάθεση του "πουθενά" `p = NULL` (και όχι `*p = NULL`!).

14.1 Δυναμική παραχώρηση μνήμης

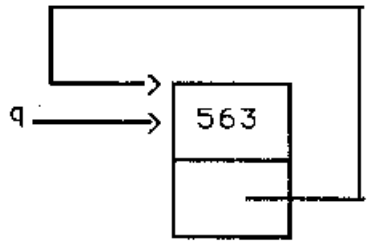
Δημιουργία (δέσμευση) νέου "κουτιού" γίνεται με χρήση του τελεστή `NEW(t)`, όπου `t` ο τύπος δεδομένων που θα αποθηκεύονται στο νέο κουτί. Π.χ.

```
nodeptr q;
q = NEW(nodeptr);
```

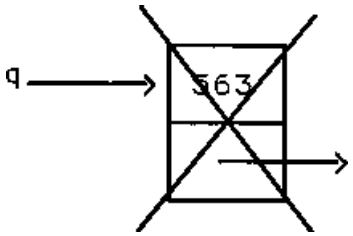


Το νέο κουτί που δημιουργείται έχει ακόμη απροσδιόριστο (undefined) περιεχόμενο. Ορίζουμε το περιεχόμενο με αναθέσεις, π.χ.

```
q->info = 563;
q->next = NULL;
```

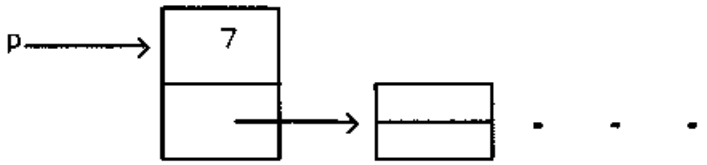


Αποδέσμευση (επιστροφή) "κουτιού" γίνεται με χρήση της συνάρτησης `DELETE(q)`, που παραχωρεί το κουτί στη διοίκηση της μνήμης ώστε να μπορεί να χρησιμοποιηθεί το κομμάτι αυτό της μνήμης από άλλο χρήστη.

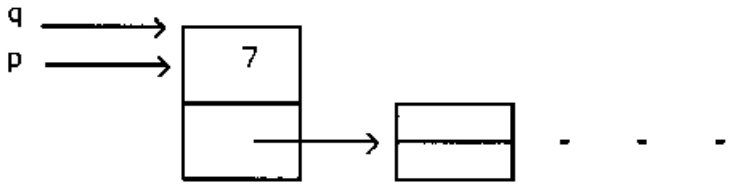


ΠΡΟΣΟΧΗ: Στην ανάθεση να συμφωνούν οι τύποι.

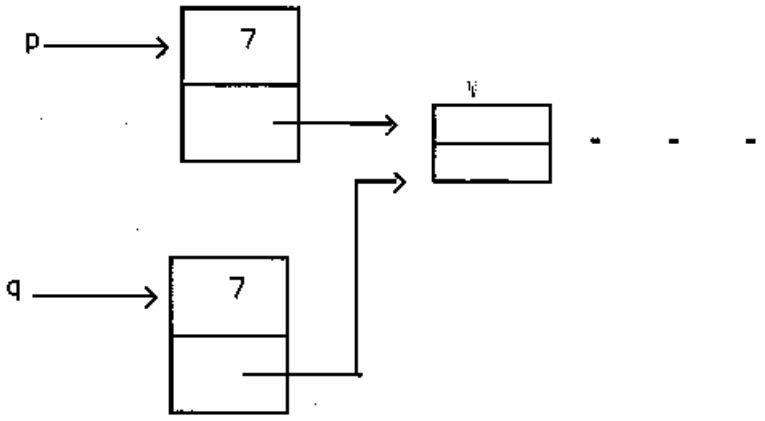
Έστω p δείκτης, άρα *p περιεχόμενο θέσης που "δείχνει" ο δείκτης p.



Τότε q = p σημαίνει ο δείκτης q να δείχνει εκεί που δείχνει ο δείκτης p,



ενώ *q = *p σημαίνει το περιεχόμενο της θέσης που δείχνει ο δείκτης p να αντιγραφεί και στη θέση που δείχνει ο δείκτης q.



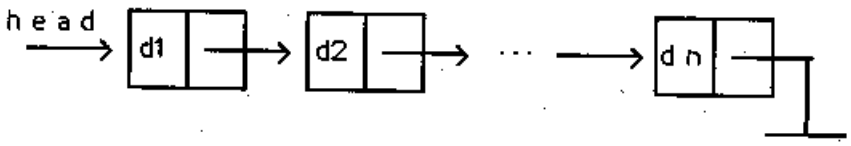
Πριν αναθέσουμε όμως q = p θα πρέπει να φροντίσουμε να επιστραφεί στη μνήμη, με DELETE(q), η θέση που έδειχνε ο q, αλλιώς θα έχουμε δημιουργήσει ένα κουτί-σκουπίδι που απλώς καταλαμβάνει χώρο μνήμης και είναι άχρηστο, δηλαδή μη προσπελάσιμο. Υπάρχουν λειτουργικά συστήματα με συλλογή σκουπιδιών (garbage collection).

Ακόμη όμως χειρότερο λάθος είναι να κάνουμε DELETE(q) και μετά να ξεχάσουμε να δώσουμε νέα τιμή στο q (π.χ. q = p). Τότε θα έχουμε ένα "ξεκρέμαστο δείκτη" (dangling pointer) που δείχνει σε περιοχή μνήμης που δεν μας ανήκει — μπορεί επομένως να μπει κάτι στη θέση αυτή εν αγνοία μας (π.χ. από άλλο χρήστη).

Ακόμη πιο επικίνδυνο είναι το πρόβλημα ξεκρέμαστου δείκτη που δημιουργείται αν κάνουμε DELETE(q), δώσουμε νέα τιμή στο q, αλλά υπάρχει και άλλος δείκτης r που δείχνει στη θέση που επιστρέψαμε στη μνήμη με DELETE(q).

14.2 Συνδεδεμένες λίστες

Συνδεδεμένη λίστα (linked list) είναι μια διαδοχή κόμβων (κουτιών) που υλοποιούνται με structs. Κάθε ένας από τους κόμβους αυτούς περιέχει τις πληροφορίες που θέλουμε να αποθηκεύσουμε σε αυτόν, καθώς επίσης και ένα δείκτη στην περιοχή της μνήμης που περιέχει τον επόμενο κόμβο. Πρέπει να υπάρχει κάποιος δείκτης π.χ. `head` (κεφαλή) που να δείχνει στον πρώτο κόμβο. Ο δείκτης που περιέχεται στον τελευταίο κόμβο έχει την τιμή `NULL`.



όπου `d1`, `d2`, ..., `dn` τα περιεχόμενα.

Στις συνδεδεμένες λίστες η προσθήκη και η διαγραφή στοιχείων είναι πιο απλή από ότι στους πίνακες. Όλα τα στοιχεία της λίστας είναι του ιδίου τύπου, όπως συμβαίνει και στους πίνακες.

Οι θέσεις αποθήκευσης για μια συνδεδεμένη λίστα συνήθως δημιουργούνται “δυναμικά” (κατά τη διάρκεια εκτέλεσης του προγράμματος).

Ξαναθυμίζουμε τον τύπο `nodetype` που περιέχει δύο πεδία: ένα για ακέραιο και ένα για δείκτη σε διεύθυνση όπου έχει αποθηκευτεί struct του ίδιου τύπου.

```
struct node_t {
  int info;
  struct node_t *next;
};

typedef struct node_t node, *nodeptr;
```

Κάθε τιμή τύπου `nodeptr` θα είναι ένας δείκτης σε μια μη-προκαθορισμένη περιοχή μνήμης που περιέχει πληροφορίες του τύπου `node`.

Κάθε κόμβος μέσα στη συνδεδεμένη λίστα δείχνει στον επόμενο του, εκτός από τον τελευταίο που δείχνει στο `NULL`. Χρειάζεται δήλωση για το δείκτη `head` που θα δείχνει στον πρώτο κόμβο:

```
nodeptr head;
```

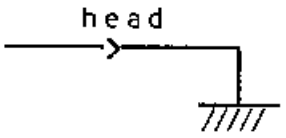
π.χ. `WRITELN(head->info);` τυπώνει τον αριθμό που περιέχεται στον πρώτο κόμβο της λίστας και `head = head->next;` μετακινεί το δείκτη στο δεύτερο στοιχείο της λίστας.

Ακολουθεί ένα παράδειγμα δημιουργίας συνδεδεμένης λίστας. Η παρακάτω συνάρτηση διαβάζει ακέραιους αριθμούς από το πληκτρολόγιο και τους αποθηκεύει στο πεδίο `info` των κόμβων μίας συνδεδεμένης λίστας. Στο τέλος, η λίστα περιέχει τους αριθμούς που δόθηκαν σε αντίστροφη σειρά.

```
FUNC nodeptr readListReversed ()
{
  nodeptr head = NULL, n;
  int data;

  while (scanf("%d", &data) == 1) {
    n = NEW(node);
    n->info = data;
    n->next = head;
    head = n;
  }
  return head;
}
```

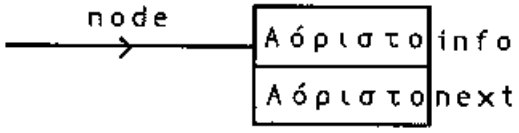
Εκτέλεση με το χέρι:

```
head = NULL;
```

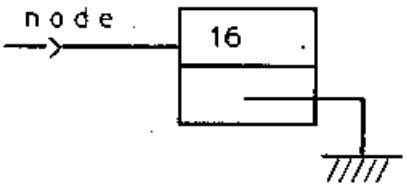



```
scanf("%d", &data);
```

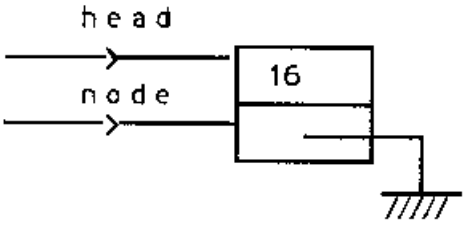
```
n = NEW(node);
```




```
n->info = data;
n->next = head;
```



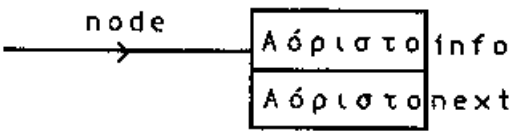

```
head = n;
```



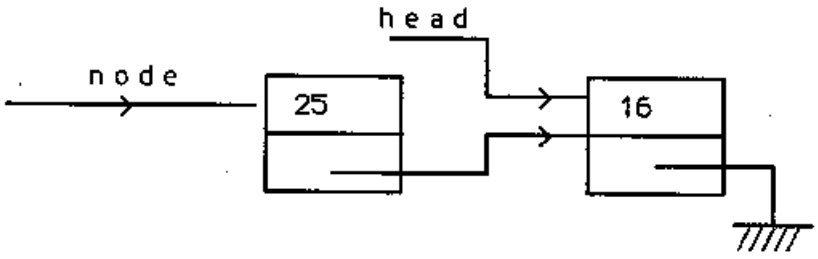

```
scanf("%d", &data);
```

διαβάζεται π.χ. ο αριθμός 25

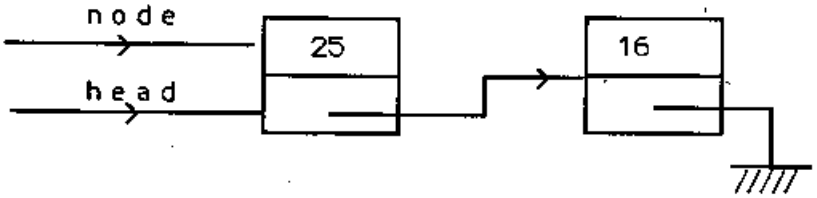
```
n = NEW(node)
```



```
n->info = data;
n->next = head;
```




```
head = n;
```



κ.ο.κ. μέχρι να ολοκληρωθεί ο βρόχος **while**.

Παράδειγμα για τύπωμα λίστας:

```
PROC print (nodeptr p)
{
  while (p != NULL) {
     WRITELN(p->info);
     p = p->next;
  }
}
```

ή αναδρομικά:

```
PROC print (nodeptr p)
{
  if (p != NULL) {
    WRITELN(p->info);
    print(p->next);
  }
}
```

Παράδειγμα για ανάποδο τύπωμα λίστας (με αναδρομή):

```
PROC printBack (nodeptr p)
{
  if (p != NULL) {
    printBack(p->next);
    WRITELN(p->info);
  }
}
```

Κεφάλαιο 15

Δομές δεδομένων

15.1 Πολυπλοκότητα. Τάξη μεγέθους: O, Ω, Θ

Οι συναρτήσεις που μας ενδιαφέρουν είναι τέτοιες που να μπορούν να εκφράσουν την πολυπλοκότητα κάποιου αλγορίθμου (κόστος αριθμού υπολογιστικών βημάτων: χρόνος, ή κόστος μεγέθους μνήμης που χρειάζεται: χώρος). Έτσι π.χ. μπορούμε να υποθέσουμε ότι οι συναρτήσεις μας (f, g, κ.λπ.) είναι θετικές και (μονότονα, όχι όμως κατ' ανάγκη γνησίως) αύξουσες. Ορισμός:

1. $O(f) = \{g \mid \exists c, \exists n_0, \forall n > n_0 : g(n) < c \cdot f(n)\}$
2. $\Omega(f) = \{g \mid \exists c, \exists n_0, \forall n > n_0 : g(n) > c \cdot f(n)\}$
3. $\Theta(f) = \{g \mid \exists c_1, c_2, \exists n_0, \forall n > n_0 : c_1 < \frac{g(n)}{f(n)} < c_2\}$

Παράδειγμα: $5n^2 + 4n - 2n \log n + 7 = \Theta(n^2)$

Σημείωση:

1. Γράφουμε $g = O(f)$ αντί $g \in O(f)$.
2. $\Theta(f) = O(f) \cap \Omega(f)$.
3. Επειδή δεν έχουμε "παθολογικές" συναρτήσεις (π.χ. $1/n$ δεν είναι δυνατόν να είναι πολυπλοκότητα αλγορίθμου) χρησιμοποιούμε $O(1)$ για να υποδηλώσουμε μια σταθερή συνάρτηση.

Ισχύουν:

1. Αν $p(n)$ πολυώνυμο και an^b είναι ο παράγων με το μεγαλύτερο εκθέτη, τότε $O(p(n)) = n^b$.
2. $O(1) < 0(\log * n) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^2 \log^5 n) < O(n^3) < \ldots <$ Poly $< O(2^n) < O(n!) < O(n^n) < O(2 \text{ ^^ } n) < ...$

όπου Poly $= n^{O(1)}$ και $2 \text{ ^^ } n = 2^{2^{2^{\cdots^2}}}$ (n φορές), η υπερεκθετική συνάρτηση, $\log * n$ η αντίστροφη της υπερεκθετικής.

Εάν υπάρχει αλγόριθμος που λύνει κάποιο πρόβλημα σε χρόνο $O(f(n))$ τότε λέμε ότι η (χρονική) πολυπλοκότητα του προβλήματος είναι $O(f(n))$ (άνω φράγμα). Εάν υπάρχει απόδειξη ότι το πρόβλημα χρειάζεται τουλάχιστον $\Omega(g(n))$ χρόνο για να λυθεί, τότε λέμε ότι η πολυπλοκότητα του προβλήματος είναι $\Omega(g(n))$ (κάτω φράγμα). Εάν έχουμε αλγόριθμο και απόδειξη με την ίδια συνάρτηση — $(f(n))$ και $\Omega(f(n))$ — τότε λέμε ότι η πολυπλοκότητα του προβλήματος είναι $\Theta(f(n))$. Παράδειγμα: η πολυπλοκότητα ταξινόμησης (με συγκρίσεις) είναι $\Theta(n \log n)$.

15.2 Γραμμικές δομές δεδομένων

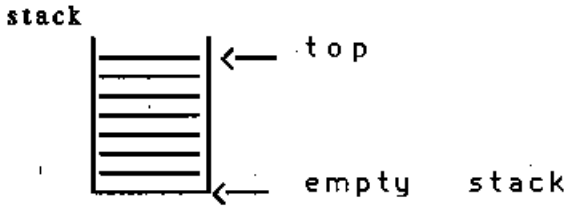
Ηδη γνωστές δομές:

- πίνακας (array)
- δομή (struct)
- ένωση (union)
- αρχείο (file)

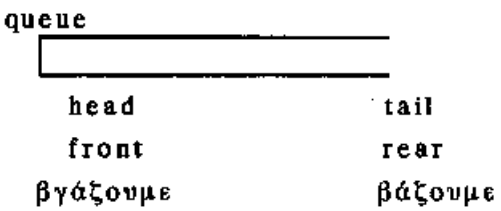
Νέες αφηρημένες δομές:

- στοίβα (stack, LIFO: last in - first out)
- ουρά (queue, FIFO: first in - first out)
- γραμμική λίστα (linear list)

15.3 Στοίβες και ουρές



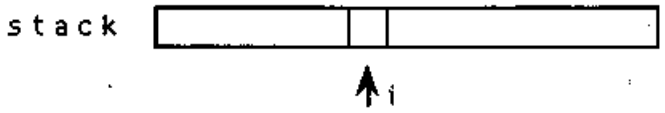
βάζουμε με push — βγάζουμε με pop



Δεν χρειάζεται οι χρήστες να ξέρουν ακριβώς πώς υλοποιείται π.χ. η στοίβα ή η ουρά (με πίνακα ή συνδεδεμένη λίστα). Αυτή η φιλοσοφία λέγεται **αφαίρεση τύπων δεδομένων** (abstract data type). Χρειάζονται μόνο τα ακόλουθα:

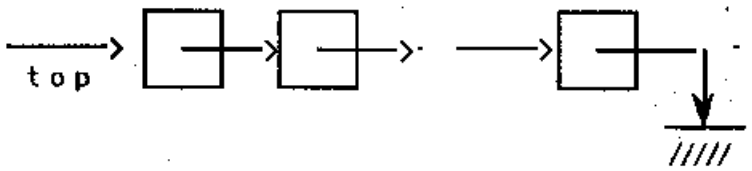
- Για στοίβα:
  - ο τύπος `stack`
  - οι συναρτήσεις `push`, `pop` και `empty`.
- Για ουρά:
  - ο τύπος `queue`
  - οι συναρτήσεις `enqueue`, `dequeue` και `empty`.

15.3.1 Υλοποίηση στοίβας με πίνακα



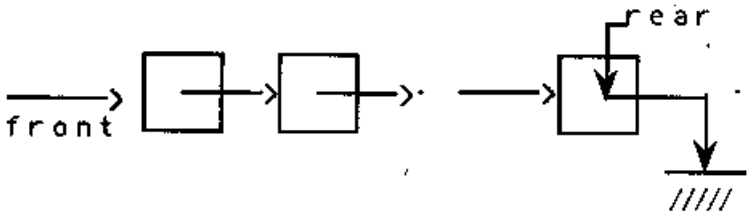

```
const int size = 100;

struct stack_t {
  int arr[size], top;
};

typedef struct stack_t stack;

FUNC stack empty ()
{
  stack result;
  result.top = 0;
  return result;
}

PROC push (stack *s, int data)
{
  s->arr[s->top] = data;
  s->top++;
}

FUNC bool pop (stack *s, int *dataptr)
{
  if (s->top == 0) return false;
  s->top--;
  *dataptr = s->arr[s->top];
  return true;
}
```

15.3.2 Υλοποίηση στοίβας με δείκτες




```
struct node_t {
  int info;
  struct node_t *next;
};

typedef struct node_t node, *stack;

FUNC stack empty ()
{
  return NULL;
}

PROC push (stack *s, int data)
{
  node *p;
  p = NEW(node);
  p->info = data;
  p->next = *s;
  *s = p;
}

FUNC bool pop (stack *s, int *dataptr)
{
  node *p;
  if (s == NULL) return false;
  p = *s;
  *dataptr = (*s)->info;
  *s = (*s)->next;
  DELETE(p);
  return true;
}
```

15.3.3 Υλοποίηση ουράς με δείκτες



```
struct node_t {
  int info;
  struct node_t *next;
};

typedef struct node_t node;

struct queue_t {
  node *front, *rear;
};

typedef struct queue_t queue;

FUNC queue empty ()
{
  queue result;
  result.front = NULL;
  result.rear = NULL;
  return result;
}

PROC enqueue (queue *q, int data)
{
  node *p;
  p = NEW(node);
  p->info = data;
  p->next = NULL;
  if (q->front == NULL) q->front = p;
  else                  q->rear->next = p;
  q->rear = p;
}

FUNC bool dequeue (queue *q, int *dataptr)
{
  node *p;
  if (q->front == NULL) return false;
  p = q->front;
  *dataptr = q->front->info;
  if (q->front == q->rear) q->rear = NULL;
  q->front = q->front->next;
  DELETE(p);
  return true;
}
```

15.3.4 Γραμμικές λίστες

Εισαγωγή στο τέλος

```
PROC insertAtRear (list *l, int data)
{
  node *p, *q;
  p = NEW(node);
  p->info = data; p->next = NULL;
  if (*l == NULL) *l = p;
  else { q = *l;
         while (q->next != NULL)
           q = q->next;
         q->next = p;
       }
} // O(n)
```

Εισαγωγή μετά από το δοθέντα κόμβο

```
PROC insertAfter (node *p, int data)
{
  node *q;
  if (p != NULL) {
    q = NEW(node);
    q->info = data;
    q->next = p->next;
    p->next = q;
  }
} // O(1)
```

Διαγραφή μετά από το δοθέντα κόμβο

```
FUNC bool deleteAfter (node *p, int *dataptr)
{
  node *q;
  if (p ==  NULL OR p->next == NULL) return false;
  q = p->next;
  *dataptr = q->info;
  p->next = q->next;
  DELETE(q);
  return true;
} // O(1)
```

Αν αντί να δίνεται το προηγούμενο του στοιχείου που θέλω να διαγράψω, δίνεται το ίδιο το στοιχείο, τότε: διαγράφω το επόμενο αφού όμως πρώτα αντιγράψω το περιεχόμενο του στο τρέχον. Ετσι το περιεχόμενο του τρέχοντος χάνεται (δηλαδή η διαγραφή πέτυχε) ενώ το περιεχόμενο του επομένου μένει ανέπαφο. Για τη διαγραφή του τελευταίου στοιχείου χρησιμοποιούμε φρουρό (δες λίγο παρακάτω).

Εύρεση τιμής

```
FUNC node* search (list l, int data)
{
  node *p;
  for (p = l; p != NULL; p = p->next)
    if (p->info == data) return p;
  return NULL;
} // O(n)
```

Για να αποφύγω το διπλό έλεγχο (δηλ. τους ελέγχους `p != NULL` και `p->info == data`) θα μπορούσα να είχα ένα παραπανίσιο κουτί στο τέλος της λίστας, με ενδεικτική τιμή. Αυτός ο επιπλέον κόμβος με την ψευδοτιμή (dummy value) λέγεται φρουρός (sentinel).

Αντιστροφή λίστας

```
PROC reverse (list *l)
{
  node *p, *q;
  q = NULL;
  while (*l != NULL) {
    p = *l;
    *l = p->next;
    p->next = q;
    q = p;
  }
  *l = q;
} // O(n)
```

Εκτέλεσε με το χέρι την παραπάνω διαδικασία!

Συνένωση δύο λιστών

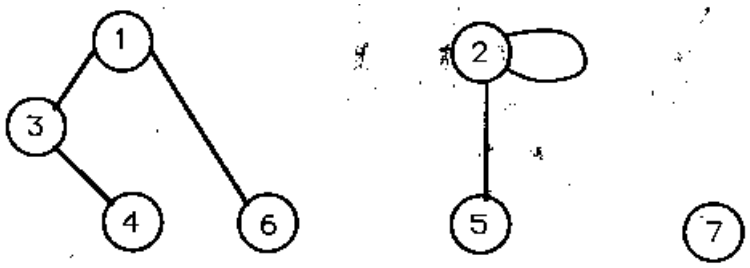
```
PROC concat (list *l1, list l2)
{
  node *p;
  if (l2 == NULL) return;
  if (*l1 == NULL) *l1 = l2;
  else {
    p = *l1;
    while (p->next != NULL) p = p->next;
    p->next = l2;
  }
} // O(n1)
```

15.4 Γράφοι και δέντρα

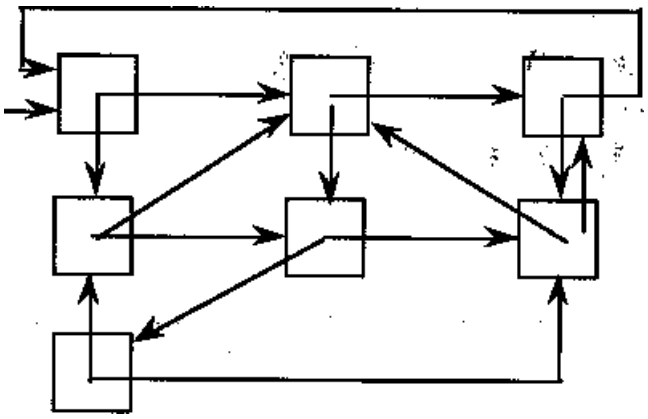
Γράφος ή **γράφημα** (graph) $G = (V, E)$ όπου V πεπερασμένο σύνολο (κόμβων) και σύνολο ζευγών από το V (ακμών).

Παράδειγμα: $V = \{1, 2, 3, 4, 5, 6, 7\}$, $= \{\{x, y\} \mid x, y \in V, x + y = 4 \text{ή} x + y = 7\}$

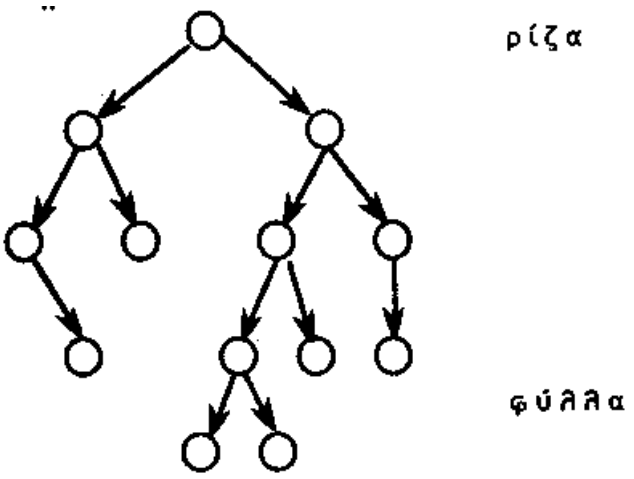
Γραφική Παράσταση:



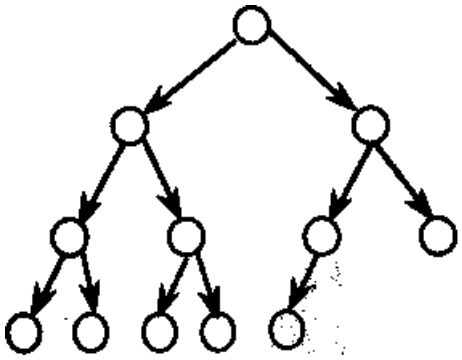
Αν οι ακμές είναι **διατεταγμένα ζεύγη** τότε μιλάμε για **κατευθυνόμενο γράφο** (directed graph). Με δείκτες μπορούμε να κατασκευάσουμε οποιαδήποτε περίπλοκη δομή δεδομένων που να μοιάζει με κατευθυνόμενο γράφο, π.χ. με struct που έχει 2 δείκτες:



Δυαδικά δέντρα (binary trees) είναι ειδικοί γράφοι της μορφής:

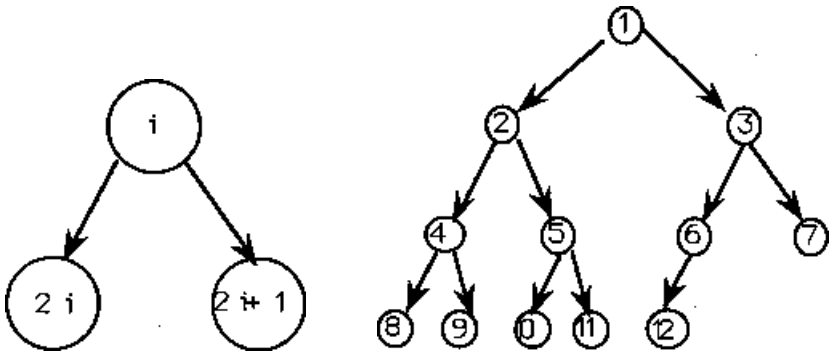


Πλήρη δυαδικά δέντρα λέγονται τα δέντρα της μορφής: π.χ.

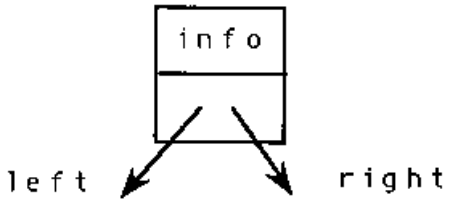


δηλαδή μόνο το κατώτατο επίπεδο μπορεί να μην είναι πλήρες.

Εάν το πλήθος των κόμβων στην περίπτωση αυτή είναι n τότε το ύψος είναι $O(\log n)$. Τα δυαδικά δέντρα μπορούν να υλοποιηθούν με array όπου οι κόμβοι αποθηκεύονται με αυτή την αρχή:



Μπορούν όμως να υλοποιηθούν και με pointers:




```
struct node_t {
  int info;
  struct node_t *left, *right;
};

typedef struct node_t node, *tree;
```

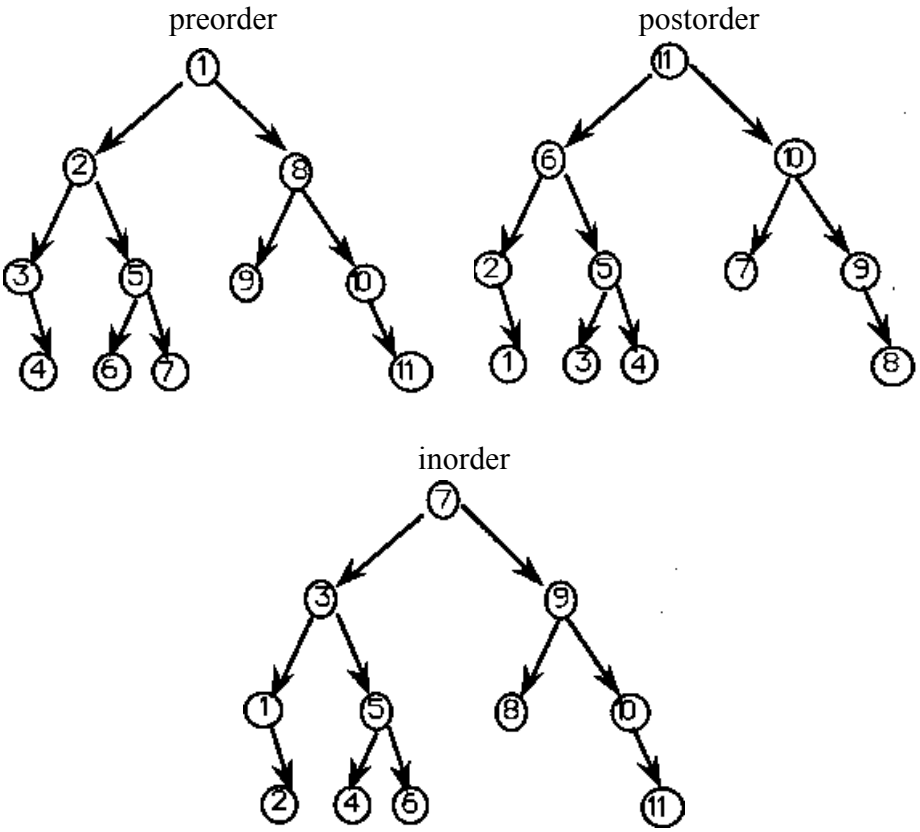
Αν θέλουμε να διατρέξουμε (π.χ. για αναζήτηση στοιχείου) όλους τους κόμβους ενός δυαδικού δέντρου, μπορούμε να το κάνουμε με διάφορους τρόπους (αναδρομικά):

- **προθεματικά** (preorder): για κάθε υποδέντρο πρώτα η ρίζα, μετά το αριστερό του υποδέντρο και μετά το δεξιό.

- **επιθεματικά** (postorder): για κάθε υποδέντρο πρώτα το αριστερό υποδέντρο, μετά το δεξιό και μετά η ρίζα.

- **ενθεματικά** (inorder): για κάθε υποδέντρο πρώτα το αριστερό υποδέντρο, μετά η ρίζα και μετά το δεξιό.

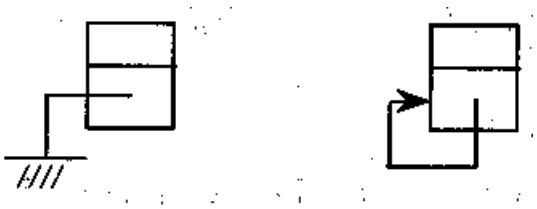
Π.χ.



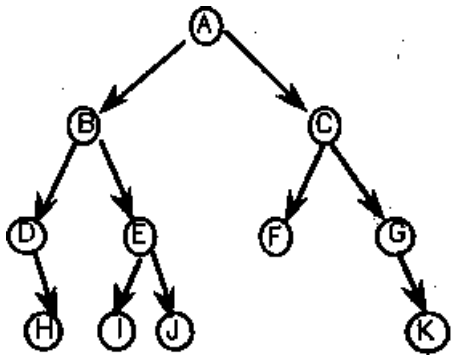

```
PROC preorder (tree t)
{
  if (t != NULL) {
    WRITELN(t->info);
    preorder(t->left);
    preorder(t->right);
  }
}
```

Παρακάτω διατρέχουμε δυαδικό δέντρο **χωρίς αναδρομή** (με threading). Περνάμε κάθε κόμβο τρεις φορές (συνδυασμός preorder, inorder, και postorder).

Προσοχή: Υποθέτουμε ότι οι κάτω-κάτω δείκτες δεν δείχνουν στο `NULL` αλλά στον "εαυτό" τους.



```
PROC threading (tree p)
{
  nodeptr q, r;

  r = NULL;
  while (p != NULL) {
    WRITELN(p->info);
    q = p->left;
    p->left = p->right;
    p->right = r;
    r = p;
    p = q;
  }
}
```



Διατρέχεται ως εξής:

A B D D H H H D B E I I I E J J J E B A C F F F C G G K K K G C A

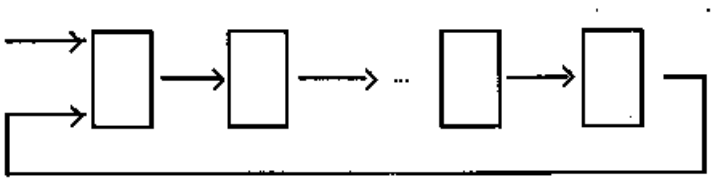
Εκτέλεσε με το χέρι την παραπάνω διαδικασία.

15.5 Άλλες λίστες

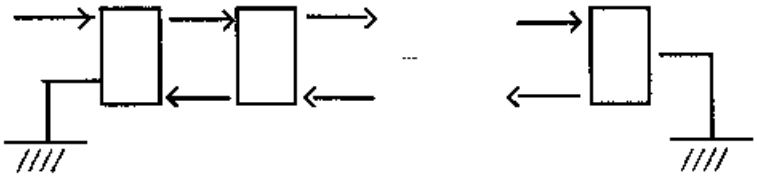
Αυτοοργανωνόμενες λίστες

Η δομή περιέχει μετρητή που μετράει πόσες φορές χρειαζόμαστε αυτό το στοιχείο. Ανάλογα με αυτόν τον αριθμό χρησιμότητας επανατοποθετούμε το στοιχείο πιο μπροστά ή πιο πίσω, έτσι ώστε η προσπέλαση αργότερα να είναι πιο αποδοτική.

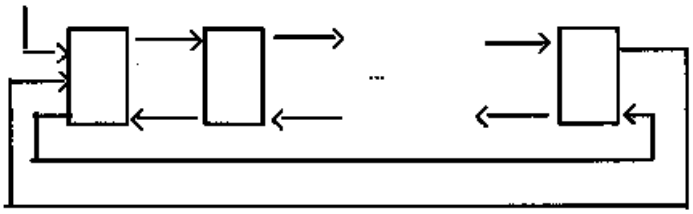
Κυκλική λίστα



Διπλά συνδεδεμένη λίστα



Διπλά συνδεδεμένη κυκλική λίστα



15.6 Ουρά προτεραιότητας

Ουρά προτεραιότητας (priority queue) είναι μια αφηρημένη δομή δεδομένων στην οποία μας ενδιαφέρει να βάζουμε (γρήγορα) νέα στοιχεία και να βγάζουμε (γρήγορα) το στοιχείο με την (εκάστοτε) μεγαλύτερη προτεραιότητα. Τέτοια σειρά προτεραιότητας χρησιμοποιούν π.χ. τα λειτουργικά συστήματα για την αποδοτική χρονοδρομολόγηση (scheduling) διαφόρων διεργασιών (process).

Αν χρησιμοποιήσουμε αταξινόμητη λίστα ή πίνακα τότε φυσικά βάζουμε γρήγορα, δηλ. σε χρόνο $O(1)$, βγάζουμε όμως αργά, δηλ. σε χρόνο $O(n)$, μετά από γραμμική αναζήτηση. Αν, από την άλλη μεριά, ταξινομήσουμε τα στοιχεία, τότε βάζουμε γρήγορα ((1)) και βγάζουμε σχετικά γρήγορα ($O(\log n)$) με δυαδική αναζήτηση, κοστίζει όμως η ταξινόμηση ($O(n \log n)$).

Μας ενδιαφέρει και οι δύο πράξεις να γίνονται αποδοτικά, χωρίς πρόσθετο κόστος ταξινόμησης (καλό trade-off, ισοζύγιο, αντιστάθμισμα).

Αν χρησιμοποιήσουμε τη δομή **heap** (σωρός) τότε βάζουμε και βγάζουμε σχετικά αποδοτικά: $O(\log n)$. Σωρός είναι ένα πλήρες δυαδικό δέντρο έτσι ώστε η προτεραιότητα κάθε κόμβου να είναι μεγαλύτερη από την προτεραιότητα όλων των κόμβων που είναι απόγονοι του. Ο σωρός υλοποιείται εύκολα με γραμμικό πίνακα (array).

Η προσθήκη νέου στοιχείου είναι τώρα σχετικά απλή: μπαίνει στο πρώτο ελεύθερο φύλλο του δέντρου και μετακινείται (στο κλαδί) προς τη ρίζα με συγκρίσεις προτεραιότητας και εναλλαγές ώσπου να βρει τη θέση του. Υψος δέντρου (μήκος κλαδιού): $O(\log n)$.

Η εξαγωγή του στοιχείου με τη μεγαλύτερη προτεραιότητα είναι άμεση, αλλά χρειάζεται μετά τη διαγραφή της ρίζας να επανασυγκολληθεί ο σωρός. Αυτό γίνεται επίσης σε χρόνο $O(\log n)$: το τελευταίο στοιχείο μπαίνει στη ρίζα και μετακινείται προς τα φύλλα, συγκρίνοντας προτεραιότητες, ώσπου να βρει τη θέση του.

Με χρήση σωρού μπορεί να γίνει και αποδοτική ταξινόμηση: heapsort: $O(n \log n)$.

Κεφάλαιο 16

Το λειτουργικό σύστημα Unix

Κεφάλαιο 17

Internet

Ένα **δίκτυο υπολογιστών** (computer network) αποτελείται από ένα σύνολο υπολογιστών που συνδέονται με ειδικές συσκευές και προγράμματα επικοινωνίας με σκοπό την επικοινωνία μεταξύ τους, δηλαδή την ανταλλαγή πληροφοριών ή δεδομένων. Η δημιουργία δικτύων υπολογιστών θεωρείται ένα μεγάλο βήμα εμπρός στον τομέα της πιο αποδοτικής χρήσης τους. Ολόκληρα δίκτυα υπολογιστών μπορούν να συνδέονται μεταξύ τους, δημιουργώντας έτσι μεγαλύτερα δίκτυα. Το μεγαλύτερο δίκτυο υπολογιστών είναι το Internet (με κεφαλαίο I), που συνδέει σήμερα χιλιάδες μικρότερα δίκτυα υπολογιστών, εκατομμύρια υπολογιστών και δεκάδες εκατομμυρίων χρήστες, σε ολόκληρο τον κόσμο.

Το Internet δεν έχει ιδιοκτήτη. Υπάρχουν εταιρίες που συνεισφέρουν στη διαχείρισή του αλλά δεν υπάρχει κάποιος κεντρικός φορέας ή οργανισμός που να το ελέγχει. Αντίθετα, τα τοπικά και εθνικά δίκτυα που αποτελούν το Internet χρηματοδοτούνται και πιθανώς ελέγχονται από τοπικούς φορείς, ακολουθώντας τους τοπικούς νόμους και πολιτικές. Αν και ξεκίνησε καθαρά ως ακαδημαϊκό και ερευνητικό δίκτυο, το Internet γνώρισε μεγάλη άνθηση προς τα τέλη της δεκαετίας του 1990 και σήμερα χρησιμοποιείται περισσότερο για εμπορικές εφαρμογές.

17.1 Ονόματα και διευθύνσεις υπολογιστών

Οι υπολογιστές που είναι συνδεδεμένοι στο Internet χαρακτηρίζονται από μια μοναδική διεύθυνση, που ονομάζεται **διεύθυνση IP** (IP address). Η διεύθυνση αυτή αποτελείται από τέσσερις αριθμούς μεταξύ 0 και 255, χωρισμένων από τελείες, π.χ. 147.102.1.4 ή 210.0.12.99. Οι υπολογιστές που ανήκουν στο δίκτυο υπολογιστών του Ε.Μ.Π. έχουν διευθύνσεις που αρχίζουν από 147.102.

Οι διευθύνσεις IP, παρότι μοναδικές, δεν είναι ιδιαίτερα εύχρηστες για να περιγράφονται οι υπολογιστές του Internet. Για το λόγο αυτό, οι υπολογιστές ονομάζονται συνηθέστερα με ακολουθίες συμβολικών ονομάτων, που πάλι διαχωρίζονται από τελείες. Για παράδειγμα, ο υπολογιστής που εξυπηρετεί το ηλεκτρονικό ταχυδρομείο του Ε.Μ.Π. ονομάζεται `mail.ntua.gr`. Σε κάθε όνομα υπολογιστή αντιστοιχεί μια μοναδική διεύθυνση IP.

Από τα συμβολικά ονόματα που αποτελούν το πλήρες όνομα ενός υπολογιστή, το πρώτο είναι το κύριο όνομά του ενώ τα επόμενα περιγράφουν **επικράτειες** (domains), δηλαδή μικρά ή μεγαλύτερα δίκτυα υπολογιστών στα οποία αυτός ο υπολογιστής ανήκει. Στο προηγούμενο παράδειγμα, το κύριο όνομα του υπολογιστή είναι `mail`, η επικράτεια `ntua` αντιστοιχεί στο

συνολικό δίκτυο υπολογιστών του Ε.Μ.Π. ενώ η επικράτεια gr αντιστοιχεί σε όλα τα δίκτυα υπολογιστών που βρίσκονται στην Ελλάδα. Κατ' αυτό τον τρόπο, στο πλήρες όνομα ενός υπολογιστή αναφέρεται κατ' αρχήν το κύριο όνομά του και στη συνέχεια οι επικράτειες στις οποίες ανήκει, από τη στενότερη προς την ευρύτερη.

Οι επικράτειες και οι ιεραρχίες υπολογιστών που ορίζονται με αυτό τον τρόπο δεν είναι πάντα γεωγραφικές. Για παράδειγμα, τα πανεπιστήμια των Η.Π.Α. βρίσκονται στην επικράτεια edu και οι περισσότερες ιδιωτικές εταιρίες στην επικράτεια com.

17.2 Ηλεκτρονικό ταχυδρομείο (e-mail)

Το ηλεκτρονικό ταχυδρομείο (e-mail) είναι για πολλούς η βασικότερη υπηρεσία του Internet. Δίνει τη δυνατότητα σε κάποιον να αποστέλλει ηλεκτρονικά μηνύματα μέσω του Internet σε έναν ή περισσότερους παραλήπτες, οι οποίοι εξυπακούεται ότι πρέπει να έχουν πρόσβαση σε έναν από τους υπολογιστές που είναι συνδεδεμένοι στο Internet. Σε αυτά τα μηνύματα, εκτός από απλό κείμενο, μπορεί κανείς να ενσωματώσει ηλεκτρονικά έγγραφα διαφόρων μορφών, όπως π.χ. εικόνες, ήχους, μορφοποιημένο κείμενο, εκτελέσιμα προγράμματα, κ.λπ.

Κάθε χρήστης του Internet που έχει πρόσβαση στο ηλεκτρονικό ταχυδρομείο διαθέτει μια μοναδική **ηλεκτρονική ταχυδρομική διεύθυνση** (e-mail address). Οι διευθύνσεις αυτές αποτελούνται από το όνομα του χρήστη, που συνήθως ταυτίζεται με το όνομα πρόσβασης (user name ή login name) στον υπολογιστή που χρησιμοποιεί, τον ειδικό χαρακτήρα @ που διαβάζεται "at" και το όνομα του υπολογιστή που ενεργεί ως παραλήπτης του ηλεκτρονικού ταχυδρομείου για το συγκεκριμένο χρήστη. Για παράδειγμα, στην ηλεκτρονική ταχυδρομική διεύθυνση:

```
zachos@cs.ntua.gr
```

το όνομα του χρήστη είναι zachos και το όνομα του υπολογιστή που διαχειρίζεται το ηλεκτρονικό του ταχυδρομείο είναι cs.ntua.gr.

Υπάρχει πληθώρα εφαρμογών που μπορεί κανείς να χρησιμοποιήσει για να βλέπει το ηλεκτρονικό του ταχυδρομείο.

17.3 Πρόσβαση σε απομακρυσμένους υπολογιστές (telnet)

Με αυτή την υπηρεσία μπορεί κανείς να χρησιμοποιήσει από μακριά υπολογιστές που συνδέονται στο Internet. Στην περίπτωση αυτή, το τερματικό που χρησιμοποιεί συμπεριφέρεται ουσιαστικά σαν ένα τερματικό του απομακρυσμένου υπολογιστή. Με αυτό τον τρόπο μπορεί κανείς να χρησιμοποιήσει τα προγράμματα εκείνου του υπολογιστή, να προσπελάσει τα αρχεία του που βρίσκονται σε εκείνο τον υπολογιστή, κ.ο.κ. Φυσικά, για να χρησιμοποιήσει κανείς το telnet θα πρέπει αφενός να γνωρίζει το πλήρες όνομα ή τη διεύθυνση IP του απομακρυσμένου υπολογιστή, αφετέρου να διαθέτει εκεί έναν έγκυρο κωδικό πρόσβασης (login) και την κατάλληλη λέξη-κλειδί (password).

17.4 Μεταφορά αρχείων (FTP)

Τα αρχικά FTP είναι συντομογραφία για το File Transfer Protocol, δηλαδή το βασικό πρωτόκολλο που χρησιμοποιείται στο Internet για τη μεταφορά ηλεκτρονικών αρχείων μεταξύ υπολο-

γιστών. Το FTP χρησιμοποιείται όπως περίπου και το telnet: επιτρέπει σε κάποιον να συνδεθεί με κάποιον απομακρυσμένο υπολογιστή και να προσπελάσει τα αρχεία του εκεί. Η μεταφορά αρχείων από τον απομακρυσμένο υπολογιστή προς τον τοπικό αποκαλείται συχνά "κατέβασμα" αρχείων (download) ενώ η αντίστροφη μεταφορά αποκαλείται "ανέβασμα" αρχείων (upload).

Όπως και στο telnet, για να χρησιμοποιήσει κανείς το FTP θα πρέπει να γνωρίζει το πλήρες όνομα ή τη διεύθυνση IP του απομακρυσμένου υπολογιστή και να διαθέτει εκεί έναν έγκυρο κωδικό πρόσβασης (login) και την κατάλληλη λέξη-κλειδί (password). Εξαίρεση σε αυτό αποτελεί η περίπτωση του **ανώνυμου FTP** (anonymous FTP), για την οποία δε χρειάζεται κανείς να διαθέτει κωδικό πρόσβασης. Οι υπολογιστές που παρέχουν αυτή την υπηρεσία επιτρέπουν σε όλους τους χρήστες του Internet να προσπελαύνουν τα αρχεία που βρίσκονται αποθηκευμένα σε αυτούς (συνήθως μόνο για κατέβασμα) δίνοντας τον ειδικό κωδικό πρόσβασης `anonymous` και, αντί λέξης-κλειδιού, την ηλεκτρονική ταχυδρομική διεύθυνση του χρήστη. Τέτοιοι υπολογιστές, όπως π.χ. ο `ftp.ntua.gr`, διαθέτουν συνήθως συλλογές προγραμμάτων η διακίνηση των οποίων είναι ελεύθερη.

17.5 Ηλεκτρονικά νέα (news)

Η υπηρεσία ηλεκτρονικών νέων δίνει τη δυνατότητα στους χρήστες του Internet να παρακολουθήσουν και να συμμετάσχουν σε συζητήσεις που περιστρέφονται γύρω από θέματα που τους ενδιαφέρουν. Υπάρχει μεγάλος αριθμός τέτοιων θεμάτων, για κάθε ένα από τα οποία έχει δημιουργηθεί μια **ομάδα συζήτησης** (newsgroup). Για παράδειγμα, στην ομάδα

```
comp.lang.pascal
```

συζητούνται θέματα που αφορούν στη γλώσσα προγραμματισμού Pascal.

Κάθε ομάδα συζήτησης λειτουργεί σαν ένας πίνακας ανακοινώσεων. Σε αυτόν μπορεί κανείς να βρει, όποτε το επιθυμεί, μηνύματα που έχουν τοποθετήσει εκεί άλλοι συμμετέχοντες στην ομάδα. Επίσης, αν το επιθυμεί, μπορεί να γράψει ένα δικό του μήνυμα και να το ανακοινώσει στην ομάδα, περιμένοντας τις απαντήσεις των υπολοίπων. Κατ' αυτό τον τρόπο, οι ομάδες συζήτησης είναι ανοικτές προς οποιονδήποτε θέλει να τις παρακολουθήσει.

17.6 Κουτσομπολιό (chat)

Το "κουτσομπολιό" (chat ή, πληρέστερα, Internet Relay Chat — IRC) έχει αρκετές ομοιότητες με τα ηλεκτρονικά νέα. Και στις δυο περιπτώσεις, υπάρχει μια ομάδα χρηστών του Internet που συμμετέχουν σε μια συζήτηση πάνω σε κάποιο θέμα κοινού ενδιαφέροντος. Η βασική διαφορά του από τα ηλεκτρονικά νέα είναι ο τρόπος διεξαγωγής της συζήτησης. Στο IRC, η συζήτηση πραγματοποιείται σε συγκεκριμένο χρόνο και τα λεγόμενα δεν παραμένουν διαθέσιμα μετά τη λήξη της. Για να συμμετάσχει κάποιος στη συζήτηση, πρέπει να συνδεθεί στον υπολογιστή του ακριβώς αυτή την ώρα και να επιλέξει την αντίστοιχη ομάδα, που στην ορολογία του IRC ονομάζεται "κανάλι" (channel). Τότε θα μπορεί να "ακούει" τα λεγόμενα των άλλων ή και να "μιλά", όπως ακριβώς θα έκανε αν βρισκόταν μαζί με τους συνομιλητές του στον ίδιο χώρο.

17.7 Παγκόσμιος ιστός (WWW)

Ο παγκόσμιος ιστός, η συνηθέστερα WWW (World-Wide Web), περιγράφεται επίσημα ως "ένα σύστημα αναζήτησης υπερμεσικών πληροφοριών, που αποσκοπεί να δώσει παγκόσμια πρόσβαση σε ένα μεγάλο όγκο πληροφοριών". Το WWW δίνει στους χρήστες του Internet τη δυνατότητα να προσπελαύνουν μια πληθώρα πληροφοριών με συνέπεια και απλότητα.

Από την εμφάνισή του, το 1989, το WWW έχει αλλάξει κατά πολύ τον τρόπο με τον οποίο οι χρήστες του Internet διαχειρίζονται τις πληροφορίες. Ο αρχικός στόχος του ήταν η προώθηση των επιστημών και της εκπαίδευσης, όμως πολύ σύντομα έφερε επανάσταση σε πολλές άλλες δραστηριότητες της κοινωνίας, συμπεριλαμβανομένου του εμπορίου, της πολιτικής και της τέχνης.

17.7.1 Υπερμέσα και σύνδεσμοι

Η λειτουργία του WWW βασίζεται κυρίως στη χρήση υπερμέσων και τον τρόπο με τον οποίο οι χρήστες αλληλεπιδρούν με αυτά. Τα **υπερμέσα** (hypermedia) είναι ηλεκτρονικά έγγραφά που περιέχουν πληροφορίες σε πολλές διαφορετικές μορφές, όπως π.χ. κείμενο, εικόνες, ήχο, βίντεο. Αυτά τα έγγραφα μπορεί κανείς να τα αποθηκεύσει, να τα διαβάσει, να τα διορθώσει και να τα αναζητήσει. Το σημαντικό τους όμως πλεονέκτημα είναι ότι τα έγγραφα αυτά μπορούν να περιέχει συνδέσμους προς άλλα ηλεκτρονικά έγγραφα. Οι σύνδεσμοι αυτοί ονομάζονται **υπερσύνδεσμοι** (hyperlinks).

Βλέποντας ένα υπερμεσικό ηλεκτρονικό έγγραφο στην οθόνη του υπολογιστή του, μπορεί κανείς να επιλέξει (χρησιμοποιώντας συνήθως το δείκτη του ποντικιού) έναν υπερσύνδεσμο και να μεταφερθεί στο ηλεκτρονικό έγγραφο όπου αυτός οδηγεί. Από εκεί μπορεί να ακολουθήσει την ίδια διαδικασία, κ.ο.κ. Το σύνολο όλων των υπερμεσικών εγγράφων που υπάρχουν στο WWW, μαζί με τους υπερσυνδέσμους που τα διασυνδέουν, είναι αυτά που δίνουν στο WWW τη νοερή εικόνα ενός τεράστιου ιστού αράχνης (web), από την οποία προκύπτει και το όνομά του.

17.7.2 Διευθύνσεις στον παγκόσμιο ιστό (URL)

Λόγω του μεγάλου όγκου των πληροφοριών που υπάρχουν κατανεμημένες στον παγκόσμιο ιστό, αλλά και λόγω της ποικιλίας μορφής αυτών των πληροφοριών, είναι πολύ σημαντικό να υπάρχει μια ενιαία σύμβαση για τις διευθύνσεις των πληροφοριών που να αποκαλύπτει τη θέση αλλά εν μέρει και το είδος της πληροφοριάς. Οι διευθύνσεις αυτές ονομάζονται **URL** (Uniform Resource Locators). Υπάρχουν πολλές μορφές URL, ανάλογα με το είδος της πληροφορίας στην οποία δείχνουν. Η γενική μορφή πάντως ενός URL —με μικρές αποκλίσεις— είναι η εξής:

τύπος`://`*υπολογιστής*`/`*θέση*

Ο **τύπος** ενός URL είναι συνήθως μια μικρή λέξη, ενδεικτική του είδους της πληροφορίας στην οποία δείχνει και του τρόπου με τον οποίο θα γίνει η προσπέλαση σε αυτήν. Για παράδειγμα, ο τύπος `http` υποδηλώνει ότι η πληροφορία είναι ένα υπερμεσικό ηλεκτρονικό έγγραφο που βρίσκεται στον παγκόσμιο ιστό, ενώ ο τύπος `news` υποδηλώνει ότι η πληροφορία είναι μια ομάδα συζήτησης. Ο **υπολογιστής** υποδεικνύει τη διεύθυνση του υπολογιστικού συστήματος

μέσα στο δίκτυο που περιέχει την πληροφορία. Στη θέση αυτή μπορεί κανείς να γράψει οποιαδήποτε έγκυρη διεύθυνση υπολογιστή στο Internet, π.χ. `www.softlab.ntua.gr` ή `147.102.1.1`. Τέλος, η **θέση** υποδεικνύει την τοπική διεύθυνση της πληροφορίας στον υπολογιστή που προσδιορίζεται. Συνήθως πρόκειται για τη διεύθυνση ενός αρχείου στο τοπικό σύστημα αρχείων. Παραδείγματα URL είναι:

```
http://www.ntua.gr/
http://www.corelab.ntua.gr/courses/
ftp://ftp.softlab.ntua.gr/users/nickie/papers/thesis.ps.gz
```