κεφάλαιο

0

Επεξεργασία Στοιχείων – Προγραμματισμός

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα σιγουρέψουμε ότι εννοούμε το ίδιο πράγμα όταν αναφέρουμε μερικούς πολύ συνηθισμένους όρους της Πληροφορικής. Θα μάθουμε και μερικά πράγματα που θα μας απασχολούν συχνά στη συνέχεια.

Προσδοκώμενα αποτελέσματα:

- Θα καταλάβεις ότι όταν θέλουμε να γράψουμε ένα πρόγραμμα πρέπει πρώτα να καθορίσουμε με ακρίβεια τις προδιαγραφές του.
- Θα μάθεις τα συστατικά των προδιαγραφών.
- Θα μάθεις να διαβάζεις συντακτικούς ορισμούς όρων μιας γλώσσας προγραμματισμού.

Έννοιες κλειδιά:

- *αλγόριθμος*
- *πρόγραμμα*
- *προδιαγραφές προγράμματος*
- *προϋπόθεση (precondition)*
- *απαίτηση (postcondition)*
- *ορθότητα προγράμματος*
- *συμπερασματικοί κανόνες (inference rules)*
- *συντακτικά (syntax)*
- *νοηματικά (semantics)*
- *συμβολισμός BNF*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις – Επεξεργασία Στοιχείων:
Κατά την αλληλεπίδρασή μας με το περιβάλλον, ανάμεσα στα άλλα, δεχόμαστε και **πληροφορίες** (information) που πλουτίζουν τον γνωστικό μας κόσμο. Το μυαλό μας *επεξεργά-*

ζεται αυτές τις πληροφορίες, δηλαδή τις συσχετίζει μεταξύ τους και με γνώσεις που προϋπάρχουν, για να βγάλει συμπεράσματα που μας είναι χρήσιμα και να δημιουργήσει νέα γνώση. Η **επεξεργασία πληροφοριών** (information processing) είναι κάτι που γίνεται από τον άνθρωπο, ατομικώς ή συλλογικώς, με την βοήθεια μηχανών ή χωρίς αυτές, από καταβολής του ανθρώπινου γένους.

Όταν η επεξεργασία δεν γίνεται μόνο από τον άνθρωπο που έχει συλλέξει την πληροφορία, παρουσιάζεται η ανάγκη να *παραστήσει* την πληροφορία με κάποιο συμβολικό τρόπο για να τη μεταβιβάσει σε κάποιον άλλο ή στη μηχανή. Η παράσταση της πληροφορίας με έναν συμβολικό τρόπο μας δίνει τα **στοιχεία** ή **δεδομένα** (data). Έτσι, χρησιμοποιούμε και τον όρο **επεξεργασία στοιχείων** (data processing). Τα ακατέργαστα στοιχεία, που είναι το αντικείμενο της επεξεργασίας, λέγονται **στοιχεία εισόδου** (input data), ενώ τα αποτελέσματα της επεξεργασίας λέγονται **στοιχεία εξόδου** (output data).

Παράδειγμα 1

Μετρώντας, μερικές φορές, την τάση v στις άκρες ενός αγωγού και το ρεύμα i που τον διαρρέει, παίρνουμε μερικά (π.χ. 50) ζεύγη τιμών:

$$(v_k, i_k) \quad k = 0..49$$

Με κατάλληλη επεξεργασία (π.χ. μέθοδος ελαχίστων τετραγώνων), μπορούμε:

- να συναγάγουμε ότι το ρεύμα είναι ανάλογο της τάσης (νόμος του Ohm),
- να υπολογίσουμε την αντίσταση του αγωγού.

Τα ζεύγη τιμών (v_k, i_k) είναι τα στοιχεία εισόδου. Η τιμή της αντίστασης αλλά και ο νόμος του Ohm είναι τα στοιχεία εξόδου.

Παράδειγμα 2

Ένας εργάτης που επιβλέπει τη λειτουργία ενός ατμολέβητα, παρακολουθεί την πίεση μεσα στον λέβητα από ένα μανόμετρο. Όταν η πίεση μέσα στον λέβητα ξεπερνάει μια κρίσιμη τιμή, ανοίγει ειδικές βαλβίδες για να διαφύγει ατμός και να πέσει η πίεση.

Εδώ η ένδειξη του μανόμετρου είναι το στοιχείο εισόδου. Αλλά ποιό είναι το στοιχείο εξόδου; Μπορούμε να πούμε ότι είναι η ενέργεια του εργάτη. Τα πράγματα όμως γίνονται πιο καθαρά αν σκεφτούμε τη διαδικασία στο επίπεδο του νευρικού συστήματος και δούμε

- ως στοιχείο εισόδου το σήμα που στέλνει το μάτι πρός τον εγκέφαλο και
- ως στοιχείο εξόδου τη διαταγή που στέλνει ο εγκέφαλος προς το χέρι.

Όπως φαίνεται από το δεύτερο παράδειγμα, τα σύμβολα που χρησιμοποιούμε για την παράσταση των στοιχείων δεν χρειάζεται να είναι πάντοτε ορατά ή, γενικώς, αντιληπτά από μια αίσθηση.

Αυτό το παράδειγμα μας δείχνει πώς χρησιμοποιείται η επεξεργασία στοιχείων για **έλεγχο** (control) κάποιας διαδικασίας.

0.1 Αλγόριθμοι

Για την επίλυση των προβλημάτων μας και για την επεξεργασία στοιχείων, ειδικότερα, επινοούμε διάφορες μεθόδους. Μια τέτοια μέθοδος, που μπορεί να περιγραφεί *με πεπερασμένη ακολουθία καλά καθορισμένων βημάτων*, λέγεται **αλγόριθμος** (algorithm).

Ας πούμε, για παράδειγμα, ότι έχουμε δέκα ακέραιους αριθμούς, τους:

17 13 67 104 2 69 45 375 35 84

και θέλουμε να βρούμε ποιος είναι ο μέγιστος και τι σειρά έχει μέσα στη δεκάδα. Μια μεθοδος, που θα μπορούσες να ακολουθήσεις, περιγράφεται στο Σχ. 0-1.

Βήμα	Ενέργεια
1:	Ας πούμε ότι μέγιστος είναι ο 1ος (17)
2:	Ξεκίνα από το 2ο στοιχείο
3:	Αν τελείωσαν οι αριθμοί ΤΕΛΟΣ
	Όσο υπάρχουν και άλλοι κάνε τα εξής:
4:	Αν είναι μεγαλύτερος από αυτόν που είχες για μέγιστο τότε
5:	Θεώρησε ότι μέγιστος είναι αυτός ο αριθμός
6:	Προχώρησε στον επόμενο αριθμό
7:	Ξαναπήγαινε στο βήμα 3

Σχ. 0-1 Περιγραφή του αλγόριθμου «με λόγια».

Αυτή η περιγραφή δεν είναι και τόσο ικανοποιητική. Γιατί; Έχει πολλά λόγια! Βέβαια, είναι μεγάλη υπόθεση να περιγράφουμε έναν αλγόριθμο σε φυσική γλώσσα, αλλά, πολύ συχνά, όπως ήδη ξέρεις από την καθημερινή σου πείρα, οι ίδιες λέξεις και οι ίδιες εκφράσεις έχουν (έστω και ελαφρώς) διαφορετική σημασία για διαφορετικούς ανθρώπους. Μια λύση στο πρόβλημα αυτό είναι ο περιορισμός του λεξιλογίου και η χρήση καλά ορισμένων συμβόλων. Ας δοκιμάσουμε ξανά κάνοντας μερικές συμβάσεις για τον συμβολισμό:

- με το: **Βάλε** $X \leftarrow Y$
  εννοούμε: βάλε (αντίγραψε) την τιμή του Y ως τιμή του X,
- με το: **Όσο** <συνθήκη> **κάνε τα εξής:**
  εννοούμε: όσο ισχύει η συνθήκη να εκτελείς ξανά και ξανά όλα τα βήματα που ακολουθούν μέσα στο άγκιστρο (**{**) –με τη σειρά που γράφονται– και να ξαναελέγχεις τη συνθήκη.
- με το: **Αν** <συνθήκη> **τότε**
  εννοούμε: αν ισχύει η συνθήκη να εκτελέσεις μια φορά όλα τα βήματα που ακολουθούν μέσα στο άγκιστρο (**{**).

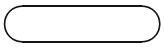
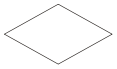
Ειδικώς για αυτήν την περίπτωση, ονομάζουμε a_k, $k = 0..9$ τους αριθμούς που έχουμε, m το μέγιστο που ψάχνουμε, k_m τη θέση του μέγιστου μέσα στη δεκάδα. Και να πώς γράφεται ο αλγόριθμός μας:

Βάλε $m \leftarrow a_0$, $k_m \leftarrow 0$, $k \leftarrow 1$
Όσο $k \le 9$ **κάνε τα εξής:**
{ **Αν** $a_k > m$ **τότε**
{ **Βάλε** $m \leftarrow a_k$, $k_m \leftarrow k$
Βάλε $k \leftarrow k + 1$

Τώρα ο αλγόριθμος έχει περιγραφεί κάπως πιο καθαρά, αλλά όχι τέλεια. Αν δεν έχεις πρόβλημα να καταλάβεις τον αλγόριθμο με τη δεύτερη ή την πρώτη διατύπωση, αυτό δεν οφείλεται στην τέλεια γλώσσα που χρησιμοποιούμε αλλά στην ανθρώπινη ευφυΐα.

Μια άλλη γλώσσα που χρησιμοποιείται ευρύτατα είναι αυτή των **λογικών διαγραμμάτων** ή **διαγραμμάτων ροής** (flowcharts). Είναι μια γλώσσα που επινοήθηκε για την περιγραφή διαδικασιών και αλγορίθμων. Για να σχεδιάσουμε ένα λογικό διάγραμμα, χρησιμοποιούμε ειδικά σύμβολα. Στον Πίν. 0-1 βλέπεις μερικά, ενώ στο Σχ. 0-2 βλέπεις πώς μπορούμε να ζωγραφίσουμε με λογικό διάγραμμα τον αλγόριθμο που δώσαμε παραπάνω.

Στόχος των παραπάνω δοκιμών ήταν να βρούμε μια γλώσσα που θα μπορεί να καταλάβει μια *μηχανή* πολύ λιγότερο ευφυής από τον άνθρωπο. Μια γλώσσα που θα μας επιτρέπει να περιγράψουμε τα βήματα ενός αλγορίθμου με *σαφήνεια* και *πληρότητα*. Οι *γλώσσες προγραμματισμού*, όπως η C++, που θα μάθουμε στο βιβλίο αυτό, είναι τέτοιες γλώσσες. Ο παραπάνω αλγόριθμος στη C++, γράφεται ως εξής:

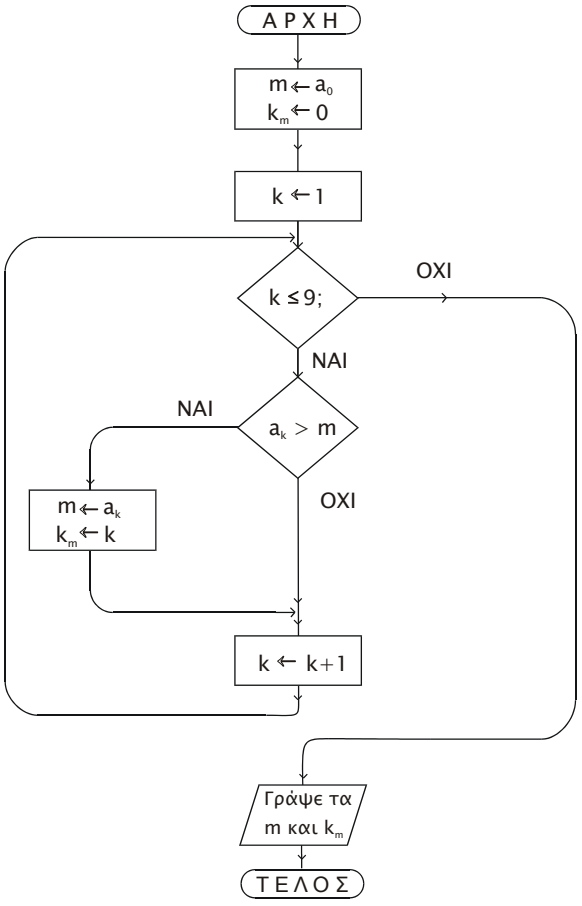
Σύμβολο	Χρήση
	για αρχή ή τέλος αλγορίθμου
	για διάβασμα (είσοδο) / γράψιμο (έξοδο) στοιχείων
	ρόμβος για αποφάσεις
	για περιγραφή επεξεργασίας
← ↑ → ↓	βέλη ροής

Πίν. 0-1: Σύμβολα για τα διαγράμματα ροής.

```
m = a[0];  km = 0;
for ( k = 1; k <= 9; k = k+1 )
   if ( a[k] > m )  {  m = a[k];   km = k;  }
```

Αυτή είναι μια περιγραφή «που μπορεί να την καταλάβει ο υπολογιστής!» Το πώς την καταλαβαίνει θα το δούμε στις επόμενες παραγράφους.

Έστω λοιπόν ότι έχουμε βρει τη γλώσσα που θα περιγράφουμε τους αλγορίθμους μας και τη μάθαμε καλά· λύθηκε το πρόβλημά μας; Όχι βέβαια! Εκτός από τη δυνατότητα δια-



Σχ. 0-2 Το λογικό διάγραμμα του αλγόριθμου για την εύρεση του μέγιστου στοιχείου.

τύπωσης του αλγόριθμου μας ενδιαφέρει και ο ίδιος ο αλγόριθμος: θα πρέπει να είναι **ορθός** (correct) και **αποδοτικός** (efficient). Με τα θέματα αυτά θα ασχοληθούμε αργότερα.

0.2 Προγραμματα και Γλώσσες Προγραμματισμού

Χρειαζόμαστε λοιπόν γλώσσες που να μπορεί να «καταλάβει» ο ηλεκτρονικός υπολογιστής (ΗΥ)![1] Ένας αλγόριθμος διατυπωμένος σε γλώσσα που μπορεί να καταλάβει ο ΗΥ είναι ένα **πρόγραμμα** (program). Κάθε βήμα του προγράμματος λέγεται **εντολή** (statement, instruction). Οι γλώσσες που χρησιμοποιούμε για να γράφουμε τα προγράμματα λέγονται **γλώσσες προγραμματισμού** (programming languages).

Όπως κάθε γλώσσα, έτσι και οι γλώσσες προγραματισμού έχουν κανόνες για την χρήση τους:

- Οι **συντακτικοί** κανόνες (syntax) καθορίζουν το *πώς πρέπει να γράφεται* ένα πρόγραμμα.
- Οι **νοηματικοί** ή **σημαντικοί** (semantics) κανόνες καθορίζουν *το νόημα αυτών που γράφονται*.

Επειδή ο ΗΥ είναι πολύ απλοϊκός σε σύγκριση με το ανθρώπινο μυαλό, *οι γλώσσες προγραμματισμού έχουν πολύ αυστηρούς κανόνες* και πολύ μικρή (ή και καθόλου) ανοχή σε λάθη. Αν κάνεις κάποιο **συντακτικό** λάθος, όσο «μικρό» κι αν είναι κατά την γνώμη σου, ο ΗΥ δεν θα καταλάβει τι του ζητάς. Αν κάνεις κάποιο **σημαντικό** λάθος, λάθος στο τι του ζητάς να κάνει, ο ΗΥ δεν μπορεί να καταλάβει ότι «άλλο ήθελες να πεις» ή ότι «προφανώς πριν από αυτό έπρεπε να κάνει εκείνο». Τελικώς θα πάρεις λάθος αποτέλεσμα. Ο ΗΥ θα είναι ένας υπάκουος αλλά κουτός υπηρέτης σου: *θα εκτελεί* με πολύ καλή απόδοση *αυτό που τον διατάζεις, αλλά όχι αυτό που θα ήθελες να τον διατάξεις*.

Ο **ψηφιακός ΗΥ** (digital computer) παριστάνει τα πάντα με τα ψηφία του δυαδικού συστήματος: το "`0`" και το "`1`". Το πρόγραμμά μας, όπως και πολλά από τα στοιχεία που χρησιμοποιεί, αποθηκεύονται στην μνήμη του ΗΥ. Επομένως... Ναι, αυτό που κατάλαβες (ή φοβήθηκες): το πρόγραμμα είναι γραμμένο με 0 και 1. Μια τέτοια γλώσσα προγραμματισμού, με 0 και 1, λέγεται **γλώσσα μηχανής** (machine language). Ο προγραμματισμός σε μια τέτοια γλώσσα είναι δύσκολος.

Παράδειγμα

Στη γλώσσα μηχανής κάποιου υπολογιστή οι εντολές:

```
0100000000000111
0001000000001000     (A)
0101000000001001
```

θα μπορούσαν να σημαίνουν τα εξής:

1. πρόσθεσε τους ακέραιους που βρίσκονται στις θέσεις 7 και 8 της μνήμης,
2. αποθήκευσε το άθροισμα στην θέση 9.

Τα τέσσερα πρώτα ψηφία της κάθε μιας από τις παραπάνω εντολές δίνουν τη διαταγή (εντολή), το τι πρέπει να γίνει. Τα υπόλοιπα δείχνουν μια θέση της μνήμης –μια **διεύθυνση** (address). Αν ξέρεις το δυαδικό σύστημα, μπορείς να «δεις» τα 7, 8 και 9 για τα οποία μιλάμε παραπάνω.

Συνήθως, για να διευκολυνθεί η δουλειά του προγραμματιστή, η εισαγωγή του προγράμματος κλπ μας δίνεται η δυνατότητα να γράφουμε τις εντολές με συμβολικά ονόματα και τις διευθύνσεις στο δεκαεξαδικό ή το οκταδικό ή το δεκαδικό σύστημα. Μας δίνεται δηλαδή η δυνατότητα να γράψουμε τις παραπάνω εντολές με πιο βολικό τρόπο, π.χ.:

---

[1] ή απλώς **υπολογιστής** (computer).

```
LOAD    7
ADD     8
STORE   9
```

Παρ' όλο που οι μεγάλες δυσκολίες δεν αντιμετωπίστηκαν, αυτές οι εντολές έχουν κάποιο νόημα, τουλάχιστον για τους αγγλομαθείς. Παρακάτω θα δεις ότι έχουμε και άλλα εργαλεία που κάνουν τα πράγματα ακόμη καλύτερα.

Μια **συμβολική γλώσσα** (assembly language) έχει κατά βάση τις εντολές της γλώσσας μηχανής του ΗΥ, αλλά επιτρέπει χρήση συμβολικών διευθύνσεων. Είναι το «επόμενο βήμα» μετά τη γλώσσα μηχανής, για την ευκολία του προγραμματιστή. Οι συμβολικές γλώσσες προσφέρουν βέβαια και άλλες ευκολίες στον προγραμματιστή και χρησιμοποιούνται από πολλούς.

Τώρα όμως τα πράγματα αλλάζουν: Ο υπολογιστής μπορεί να εκτελέσει μόνο εντολές σε γλώσσα μηχανής· έτσι, δεν μπορεί να εκτελέσει αμέσως ένα πρόγραμμα που είναι γραμμένο σε συμβολική γλώσσα. Θα πρέπει προηγουμένως να **μεταφραστεί** σε γλώσσα μηχανής. *Αυτήν τη μετάφραση την κάνει κάποιο πρόγραμμα.* Το πρόγραμμα αυτό, που λέγεται **συμβολομεταφραστής** (assembler), υπάρχει έτοιμο στον ΗΥ. Ο συμβολομεταφραστής παίρνει, ως στοιχεία εισόδου, τις εντολές του προγράμματος σε συμβολική γλώσσα και παράγει, ως αποτέλεσμα, το ισοδύναμο πρόγραμμα σε γλώσσα μηχανής.[2]

Οι συμβολικές γλώσσες, παρ' όλο που είναι πιο βολικές από την γλώσσα μηχανής, δεν παύουν να είναι πιο κοντά στον υπολογιστή παρά στο χρήστη του, τον άνθρωπο. Στο παράδειγμα που δώσαμε παραπάνω, μας πηγαίνουν από

```
        LOAD  0007          LOAD  A
το      ADD   0008    στο   ADD   B
        STORE 0009          STORE X
```

Θα ήταν όμως πολύ καλύτερο αν μπορούσαμε, αντί για τα παραπάνω να γράφουμε:

```
X ← A + B
```

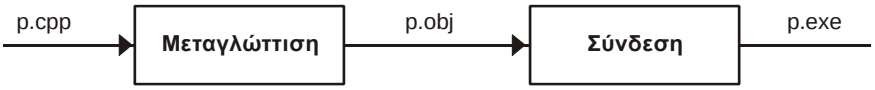
ή κάτι παρόμοιο.

Οι **γλώσσες προγραμματισμού υψηλού επιπέδου** (high level programming languages) μας δίνουν τη δυνατότητα να γράψουμε τους αλγορίθμους μας με τρόπο πιο «φιλικό», πιο οικείο στον άνθρωπο. Αυτές, είναι γλώσσες προγραμματισμού που μοιάζουν, λιγότερο ή περισσότερο, με τις ανθρώπινες γλώσσες (συνήθως τα αγγλικά) και έχουν σχέση με τα ανθρώπινα προβλήματα και τις μεθόδους που έχουμε για να τα λύνουμε. Οι περισσότερες έχουν σχεδιαστεί ώστε να ανταποκρίνονται καλύτερα σε προβλήματα ορισμένου τύπου. Για επιστημονικά προβλήματα έγιναν οι γλώσσες FORTRAN, ALGOL, APL και άλλες. Οι γλώσσες COBOL, RPG είναι σχεδιασμένες για εμπορικές εφαρμογές. Η PL/I, και η Ada μπορούν να δουλέψουν με προβλήματα πολλών τύπων. Η BASIC σχεδιάστηκε το 1965 για διδασκαλία προγραμματισμού και γνώρισε μεγάλη επιτυχία στους mini –και στους μικρο– υπολογιστές, αλλά χρησιμοποιείται απ' όλους για όλα! Η Pascal, που είναι και αυτή μια γλώσσα υψηλού επιπέδου, σχεδιάστηκε επίσης για διδασκαλία προγραμματισμού, αλλά χρησιμοποιείται στην παραγωγή.

Η C++ είναι μια γλώσσα που έχει γίνει πολύ δημοφιλής στους επιστήμονες της Πληροφορικής –αλλά και των άλλων θετικών επιστημών– και στους τεχνικούς· φαίνεται να αντικαθιστά τη FORTRAN στα επιστημονικά και τεχνικά προγράμματα. Το μεγαλύτερο μέρος του λογισμικού συστημάτων γράφεται σήμερα στη γλώσσα αυτή· το ίδιο ισχύει και για το λογισμικό εφαρμογών. Θα μας απασχολήσει σε επόμενη παράγραφο (και στα υπόλοιπα κεφάλαια του βιβλίου).

---

[2] Αν εξαιρέσουμε τις περιπτώσεις που έχουμε πολύ απλόν υπολογιστή, το πιο πιθανό είναι ότι το «ισοδύναμο πρόγραμμα σε γλώσσα μηχανής» θα παραχθεί κατά τη *φόρτωση* από τον **φορτωτή** (loader), αλλά ας τα αφήσουμε αυτά για μιαν άλλη φορά…



Σχ. 0-3 Από το αρχικό πρόγραμμα (p.cpp) στο τελικό ή εκτελέσιμο (p.exe).

Όπως το πρόγραμμα που γράφεται σε συμβολική γλώσσα έτσι και το πρόγραμμα που γράφεται σε γλώσσα υψηλού επιπέδου δεν μπορεί να εκτελεστεί από τον ΗΥ. Πρέπει πρώτα να γίνει η μετάφρασή του σε γλώσσα μηχανής. Η μετάφραση αυτή γίνεται από κάποιο άλλο πρόγραμμα που γενικώς λέγεται **μεταφραστής.**

Ένα είδος μεταφραστή είναι ο **μεταγλωττιστής** (compiler). Ο μεταγλωττιστής είναι ένα *πρόγραμμα* που παίρνει –ως στοιχεία εισόδου– το πρόγραμμά μας, γραμμένο σε γλώσσα υψηλού επιπέδου· από την επεξεργασία του βγάζει –ως αποτέλεσμα– το «ισοδύναμο» πρόγραμμα (πιθανότατα με κάποιες «βελτιώσεις») σε γλώσσα μηχανής. Αυτό που γράφει ο προγραμματιστής είναι το **αρχικό** ή **πηγαίο πρόγραμμα** (source program) και αυτό που παράγει ο μεταγλωττιστής είναι το **αντικειμενικό πρόγραμμα** (object program).

Συνήθως, το πρόγραμμα που βγάζει ο μεταγλωττιστής, δεν είναι έτοιμο για εκτέλεση. Ο μεταγλωττιστής βάζει μέσα στο πρόγραμμα «οδηγίες» για εκτέλεση μερικών προγραμμάτων, που υπάρχουν έτοιμα σε «βιβλιοθήκες» του ΗΥ. Αυτά τα προγράμματα θα πρέπει να «συνδεθούν» με το αντικειμενικό πρόγραμμα ώστε να προκύψει το τελικό **εκτελέσιμο** (executable) πρόγραμμα. Αυτή τη δουλειά την κάνει ένα άλλο πρόγραμμα που λέγεται **συνδέτης** (linker ή linkage editor). Αυτό φαίνεται διαγραμματικώς στο Σχ. 0-3.

Ένας άλλος τύπος μεταφραστή είναι ο **ερμηνευτής** (interpreter). Ο ερμηνευτής δεν παράγει αντικειμενικό πρόγραμμα, αλλά μεταφράζει μια προς μια τις εντολές και τις δίνει για εκτέλεση.

0.3 Το Σωστό Πρόγραμμα

Ας πούμε ότι γράφεις το πρόγραμμα που είδαμε στην §0.2

```
      . . .
m = a[0];  km = 0;
fot ( k = 1; k <= 9; k = k+1 )
   if ( a[k] > m) then {  m = a[k]  km = k;  }
      . . .
```

Αλλά, κάνεις μερικά λαθάκια[3]: αντί για "**for**" έβαλες "**fot**", μετά τη συνθήκη της "**if**" έβαλες το "**then**" και μεταξύ των εντολών "`m = a[k]`" και "`km = k`" δεν έβαλες ένα "**;**". Αυτά είναι παραδείγματα *συντακτικών λαθών*, που, γενικώς, δεν δημιουργούν μεγάλο πρόβλημα: θα τα βρει ο μεταγλωττιστής, θα σου δείξει πού (περίπου) είναι και θα τα διορθώσεις.

Ας δούμε τώρα κάτι άλλο:

```
      . . .
m = a[0];  km = 0;
for ( k = 11; k <= 9; k = k+1 )
   if ( a[k] > m )  {  m = a[k];  km = k;  }
      . . .
```

Εδώ δεν υπάρχουν συντακτικά λάθη. Αλλά υπάρχει κάτι άλλο, χειρότερο: ζητάμε το *k* –αυξανόμενο ξανά και ξανά κατά 1– να πάρει τιμές από 11 μέχρι 9. Αυτό δεν γίνεται και το αποτέλεσμα είναι: η "`if (a[k] > m) { m = a[k]; km = k; }`" δεν θα εκτελεσθεί ούτε μια φορά και έτσι οι *m* και *km* θα μείνουν με τις αρχικές τιμές τους.

Στην περίπτωση αυτή τα πράγματα είναι άσχημα: Οι εντολές περνούν χωρίς πρόβλημα από τον μεταγλωττιστή. Η μόνη ένδειξη ότι κάτι δεν πάει καλά είναι τα παράλογα αποτελέσματα. Αυτό εδώ είναι ένα *νοηματικό* λάθος και είναι σαφώς σοβαρότερο από το προη-

[3] Το γιατί είναι λάθη θα το μάθεις αργότερα· προς το παρόν δέξου ότι αυτό που δώσαμε στην §0.2 ήταν σωστό και δες τις διαφορές που υπάρχουν.

γούμενο. Θα μπορούσε κανείς να πει: «Σιγά το λάθος! Από απροσεξία, έμεινε το πλήκτρο πατημένο και πέρασαν δύο άσοι αντί για ένας. Είναι προφανές ότι εννοείται 1.» Τίποτε δεν είναι προφανές· όπως είπαμε, ο υπολογιστής *θα υπακούσει με ακρίβεια σε αυτό που έγραψες και όχι σε αυτό που θα ήθελες* να γράψεις.

Δες και το παρακάτω:

```
   . . .
m = a[0];  km = 0;
for ( k = 11; k <= 9; k = k+1 )
  if ( a[k] > M )  {  m = a[m];  km = k;  }
   . . .
```

Εδώ υπάρχει ένα άλλο πολύ σοβαρό λάθος: "`m = a[m]`". Το "`a[m]`" δεν έχει νόημα, διότι βάζουμε στο *m* ως τιμή έναν από τους αριθμούς που εξετάζουμε, ενώ –όπως θα έχεις καταλάβει– μέσα στις αγκύλες πρέπει να υπάρχει η σειρά του αριθμού.

Αυτό το λάθος μπορεί να γίνει από κάποιον που δεν έχει καταλάβει τι ακριβώς παριστάνουν οι μεταβλητές ή τον τρόπο που δουλεύει ο αλγόριθμος. Θα μπορούσαμε να συνεχίσουμε δίνοντας και άλλα τέτοια παραδείγματα, αλλά καλύτερα να σταματήσουμε και να ρωτήσουμε: τι θα πει «*το πρόγραμμα έχει λάθος;*»

Θα πεις: «Ωραία ερώτηση! Καλά, κουβέντα θ' ανοίξουμε τώρα; Αν δεν μου δώσει αποτέλεσμα: «μέγιστο το 375 και στη θέση 7» τότε ο αλγόριθμος έχει λάθος!» Ναι, αλλά η φιλοδοξία μας είναι να κάνουμε έναν αλγόριθμο που να δουλεύει γενικώς και όχι μόνο για τη συγκεκριμένη δεκάδα. Η απάντηση στην ερώτησή μας είναι:

- ***Ένα πρόγραμμα χαρακτηρίζεται ορθό ή λαθεμένο σε σχέση με συγκεκριμένες προδιαγραφές.***

Ποιες είναι οι προδιαγραφές για το παράδειγμά μας; Ας τις δούμε:

1. «Έχουμε δέκα ακέραιους αριθμούς» Εδώ υπάρχει μια σημαντική πληροφορία: Το ότι έχουμε να εξετάσουμε ακέραιους αριθμούς σημαίνει ότι μπορούμε να τους συγκρίνουμε με τον "`>`", πράγμα πολύ ουσιώδες για τη μέθοδο που περιγράψαμε. Αν είχαμε δέκα υπολογιστές ή δέκα δίσκους μουσικής ή δέκα μιγαδικούς αριθμούς δεν θα μπορούσαμε να κάνουμε σύγκριση με τον "`>`" και δεν θα είχε νόημα «ο μέγιστος».
2. «Ποιος είναι ο μέγιστος (από αυτούς)» Αυτό σημαίνει ότι θέλουμε να βρούμε έναν αριθμό που να είναι μεγαλύτερος από (ή ίσος με) οποιονδήποτε από τους αριθμούς που μας δόθηκαν. Μια καλύτερη διατύπωση είναι: να μην είναι μικρότερος από οποιονδήποτε από τους αριθμούς που μας δόθηκαν. Α, έτσι. Αν πάρουμε τον 500 μας κάνει; Όχι! Θα πρέπει να είναι ένας από τους αριθμούς που μας δόθηκαν.
3. «Τι σειρά έχει (ο μέγιστος) μέσα στη δεκάδα» Δηλαδή η απάντηση στην ερώτηση αυτή θα πρέπει να είναι ένας ακέραιος αριθμός μεταξύ 0 και 9. Και ακόμη, αν *m* ο μέγιστος και k_m η σειρά του στη δεκάδα θα πρέπει να έχουμε $m = a[k_m]$.

Η 1. είναι μια ιδιότητα των στοιχείων που έχουμε να επεξεργαστούμε και μπορούμε να τη χρησιμοποιήσουμε στον αλγόριθμό μας· λέγεται **συνθήκη** ή **κατηγόρημα εισόδου** (input condition/predicate) ή **προϋπόθεση** (precondition). Οι 2. και 3. είναι ιδιότητες που χαρακτηρίζουν τα αποτελέσματα της επεξεργασίας· αποτελούν τη **συνθήκη** ή **κατηγόρημα εξόδου** (output condition ή predicate) ή **απαίτηση** (postcondition).

Παρόμοια πράγματα έχουμε στην τεχνολογία και στα μαθηματικά. Σύγκρινε με τα παρακάτω:

- «Να κατασκευασθεί θερμαντικό σώμα που να αποδίδει 2000 *kcal/h* όταν τροφοδοτηθεί με τάση 220 *V* (ενεργή τιμή).»
- «Να βρεθούν πραγματικοί x_1 και x_2 τέτοιοι ώστε $ax_i^2 + bx_i + c = 0$, όπου *a*, *b*, *c* πραγματικοί τέτοιοι ώστε: $a \neq 0$ και $b^2 - 4ac \geq 0$.»
- «Δίνεται το ευθύγραμμο τμήμα *AB* και μια ευθεία *ε* που δεν είναι κάθετη στο *AB*. Να βρεθεί σημείο *M* της *ε* που ισαπέχει από τα *A* και *B*.»

Στο πρώτο παράδειγμα προϋπόθεση είναι $V_{εν} = 220\ V$, ενώ η απαίτηση είναι: απόδοση 2000 *kcal/h*.

Στο δεύτερο η προϋπόθεση είναι: $a, b, c \in \mathbb{R}$ και $a \neq 0$ και $b^2 - 4ac \geq 0$. Απαίτηση: $x_1, x_2 \in \mathbb{R}$ και $ax_1^2 + bx_1 + c = 0$ και $ax_2^2 + bx_2 + c = 0$.

Στο τρίτο η προϋπόθεση είναι: το ευθύγραμμο τμήμα *AB* δεν είναι κάθετο στην ευθεία· και η απαίτηση: σημείο *M* της *ε* τέτοιο ώστε: *MA* = *MB*.

Ο ηλεκτρολόγος που θα αναλάβει να κατασκευάσει το θερμαντικό σώμα δεν θα κάνει οτιδήποτε στην τύχη. Από τις προδιαγραφές θα υπολογίσει την αντίσταση που χρειάζεται· υπολογίζοντας τη μέγιστη ένταση ρεύματος θα επιλέξει τα σωστά καλώδια και την κατάλληλη ασφάλεια κ.ο.κ. Ο ηλεκτρολόγος θα μπορεί –με οδηγό τους υπολογισμούς που έκανε με βάση τις προδιαγραφές– να αποδείξει ότι η συσκευή του συμμορφώνεται με αυτές (τις προδιαγραφές).

Οποιοσδήποτε (σχεδόν) προσπαθήσει να λύσει το δεύτερο πρόβλημα, δεν πρόκειται να αρχίσει να δοκιμάζει αριθμούς στην τύχη. Θα χρησιμοποιήσει τους γνωστούς τύπους, που ισχύουν με την προϋπόθεση των προδιαγραφών. Αν ζητήσουμε, θα μπορεί να αποδείξει ότι οι τύποι είναι σωστοί.

Στο τρίτο πρόβλημα, δεν πρόκειται φυσικά να δοκιμάζουμε σημεία της *ε* με την ελπίδα ότι θα βρούμε κάποιο που να ισαπέχει των *A* και *B*. Θα πάρουμε τη μεσοκάθετο στο *AB* και θα βρούμε το σημείο τομής της με την *ε*. Εύκολα μπορούμε να αποδείξουμε ότι αυτό είναι το σημείο που ψάχνουμε.

Διάλεξε όποια αντιστοιχία θέλεις, αλλά –όπως οποιαδήποτε λύση σε κάποιο κατασκευαστικό πρόβλημα–

- ***το πρόγραμμα είναι ένα προϊόν για το οποίο πρέπει να υπάρχουν προδιαγραφές,***
- ***το πρόγραμμα γράφεται έτσι ώστε να συμμορφώνεται με τις προδιαγραφές,***
- ***πρέπει να αποδεικνύεται η ορθότητα του προγράμματος (δηλαδή η συμμόρφωση με τις προδιαγραφές).***

Πώς θα γράφονται οι προδιαγραφές; Αφού οι προδιαγραφές θα χρησιμοποιηθούν για την απόδειξη ορθότητας του προγράμματος θα πρέπει να διατυπώνονται με μαθηματική αυστηρότητα. Και με μαθηματικό συμβολισμό; Συνήθως αυτά πάνε μαζί.

Πώς μπορούμε να γράψουμε προδιαγραφές για το παράδειγμα του μέγιστου; Βασιζόμαστε στο ξεκαθάρισμα που κάναμε:

Προϋπόθεση: $\mathrm{a}[k] \in \mathbb{Z}$, για κάθε $k \in [\,0\,..\,9\,]$.

Απαίτηση: Υπάρχουν $m \in \mathbb{Z}$ και $k_m \in [\,0\,..\,9\,]$ τέτοια ώστε:

$(m = \mathrm{a}[k_m])$ και $(m \geq \mathrm{a}[k]$ για κάθε $k \in [\,0\,..\,9\,])$.

Κάναμε την εξής σύμβαση: με το $[\,\nu_1\,..\,\nu_2\,]$ $(\nu_1, \nu_2 \in \mathbb{Z},\ \nu_1 \leq \nu_2)$ συμβολίζουμε το υποσύνολο των ακεραίων που έχουν τιμές $\geq \nu_1$ και $\leq \nu_2$. Παρομοίως ορίζονται και τα $(\,\nu_1\,..\,\nu_2\,]$, $[\,\nu_1\,..\,\nu_2\,)$ και $(\,\nu_1\,..\,\nu_2\,)$ σε αντιστοιχία με τα υποσύνολα (διαστήματα) $(\alpha_1, \alpha_2]$, $[\alpha_1, \alpha_2)$ και (α_1, α_2) της ευθείας των πραγματικών.

Το πρόβλημα με τις ρίζες της δευτεροβάθμιας εξίσωσης, όπως το δώσαμε παραπάνω, είναι ένα *θεώρημα* που πρέπει να αποδειχτεί. Παρομοίως, ένα θεώρημα που πρέπει να αποδειχτεί είναι και το πρόβλημα για την εύρεση του μεγίστου. *Το πρόγραμμα είναι μια κατασκευαστική απόδειξη ύπαρξης*. Γενικώς, το πρόβλημα του προγραμματισμού μπορεί να διατυπωθεί ως εξής:

- ***Δίνονται τα στοιχεία $x_1,\ldots x_m$, που ικανοποιούν τη συνθήκη $\phi(x_1,\ldots x_m)$. Απόδειξε ότι υπάρχουν $y_1,\ldots y_n$ που ικανοποιούν τη συνθήκη $\psi(x_1,\ldots x_m, y_1,\ldots y_n)$.***

Ή αλλιώς:

- ***Γράψε ένα πρόγραμμα που θα παίρνει τα στοιχεία $x_1,\ldots x_m$, που ικανοποιούν τη συνθήκη $\phi(x_1,\ldots x_m)$ και θα υπολογίζει τα $y_1,\ldots y_n$ που θα πρέπει να ικανοποιούν τη συνθήκη $\psi(x_1,\ldots x_m, y_1,\ldots y_n)$.***

Εδώ θα πρέπει να αναγνώρισες στη $\phi(x_1,... x_m)$ την προϋπόθεση και στην $\psi(x_1,... x_m, y_1,... y_n)$ την απαίτηση.

Στη σύγχρονη Τεχνολογία Λογισμικού θα βρεις τον όρο **προγραμματισμός με συμβόλαιο** (programming by contract) για το αυτονόητο: τη σύνθεση προγράμματος με βάση συγκεκριμένες προδιαγραφές (που είναι το «συμβόλαιο»).

Για να βρούμε το σημείο που ισαπέχει από τα άκρα ευθύγραμμου τμήματος, χρησιμοποιούμε τον κανόνα και το διαβήτη για να φέρουμε τη μεσοκάθετο. Παραλλήλως αποδεικνύουμε ότι το σημείο που θα βρούμε είναι αυτό που ζητάμε. Το ίδιο πρέπει να κάνουμε και στο πρόγραμμά μας: δίνουμε εντολές στον υπολογιστή και με την εκτέλεσή τους προχωρούμε στον καθορισμό των τιμών των m και k_m από επεξεργασία των τιμών των a[k]. Θα πρέπει όμως να αποδείξουμε ότι τελικώς: (m = a[k_m]) και ($m \geq$ a[k] για κάθε $k \in$ [0 .. 9]) και $k_m \in$ [0 .. 9].

Για την απόδειξη ορθότητας της γεωμετρικής κατασκευής χρησιμοποιούμε θεωρήματα, αξιώματα και συμπερασματικούς κανόνες που σε ορισμένες περιπτώσεις έχουν σχέση με τη χρήση των οργάνων, π.χ.: «με τον κανόνα γράφω τη μοναδική ευθεία που περνάει από δύο σημεία», «όλα τα σημεία της γραμμής που γράφει το ένα σκέλος του διαβήτη ισαπέχουν από σημείο που βρίσκεται το άλλο σκέλος», «αφού κάθε σημείο της μεσοκαθέτου ισαπέχει από τα άκρα και το σημείο που η μεσοκάθετος τέμνει την ε ισαπέχει από τα άκρα» κλπ.

Για την απόδειξη της ορθότητας του προγράμματος θα χρησιμοποιούμε κατ' αρχήν αξιώματα και συμπερασματικούς κανόνες που έχουν σχέση με τις εντολές που ζητούμε να εκτελεστούν. Με άλλα λόγια: Κάθε εντολή έχει συγκεκριμένο νόημα, κάνει συγκεκριμένα πράγματα που μπορούμε να τα περιγράψουμε αυστηρώς με *αξιώματα* και *συμπερασματικούς κανόνες*. Με βάση αυτά μπορούμε να αποδείξουμε ότι το πρόγραμμά μας είναι σωστό.

Αν αποδείξουμε ότι

- αν ένα πρόγραμμα E αρχίσει να εκτελείται με στοιχεία εισόδου $x_1,... x_m$ που ικανοποιούν την $\phi(x_1,... x_m)$ (προϋπόθεση) και ότι
- αν η εκτέλεσή του τελειώσει κανονικώς θα μας δώσει στοιχεία εξόδου $y_1,... y_n$ που να ικανοποιούν την $\psi(x_1,... x_m, y_1,.... y_n)$ (απαίτηση)

τότε λέμε ότι το πρόγραμμά μας είναι **μερικώς ορθό** (partially correct) και γράφουμε συμβολικώς[4]:

$$\phi(x_1,... x_m) \{ E \} \psi(x_1,... x_m, y_1,.... y_n)$$

Αν τώρα έχεις αποδείξει τη μερική ορθότητα του προγράμματός σου και αν ακόμη αποδείξεις ότι *η εκτέλεση του προγράμματος θα τερματισθεί*, έχεις αποδείξει ότι το πρόγραμμά σου είναι **ολικώς ορθό** (totally correct). Συμβολικώς γράφουμε:

$$\phi(x_1,... x_m) < E > \psi(x_1,... x_m, y_1,.... y_n)$$

Όπως θα δεις στη συνέχεια, το πρόβλημα του τερματισμού έχει σχέση με τον τερματισμό εκτέλεσης των επαναληπτικών εντολών: σε ένα πρόγραμμα C++ θα πρέπει να αποδειχτεί ότι όλες οι **while**, **for** και **dowhile** καθώς και οι *αναδρομικές κλήσεις* υποπρογραμμάτων, που υπάρχουν στο πρόγραμμα, *δεν θα εκτελούνται επ' άπειρον*.

Για να κάνεις τις αποδείξεις ορθότητας προγράμματος σου χρειάζεται ο Κατηγορηματικός Λογισμός 1ης Τάξης (First Order Predicate Calculus)[5]. Εδώ, σε ένα βιβλίο εισαγωγής στον προγραμματισμό, δεν θα σου ζητήσουμε τόσα πολλά. Θα περιοριστούμε στον Προτασιακό Λογισμό (Propositional Calculus) με τις έννοιες και τα σύμβολα του οποίου θα πρέπει να είσαι πιο εξοικειωμένος. Σε κάθε περίπτωση, διάβασε το Παρ. Α για να δεις και τον συμ-

---

[4] Αλλού θα βρεις το ίδιο πράγμα γραμμένο ως εξής:

$$\{ \phi(x_1,... x_m) \} E \{ \psi(x_1,... x_m, y_1,.... y_n) \}$$

Όπως θα δεις, ο συμβολισμός που χρησιμοποιούμε είναι βολικός για τη C++.

[5] ή λογικές ανώτερης τάξης.

βολισμό που θα χρησιμοποιούμε στη συνέχεια. Φυσικά, δεν μπορούμε να αποφύγουμε τα «*για κάθε ...*» και «*υπάρχει ... τέτοιο ώστε ...*», που υπάρχουν άλλωστε και στο παράδειγμα με το μέγιστο· βάλαμε λοιπόν και μερικά πράγματα για αυτά, αλλά ελπίζουμε ότι καταλαβαίνεις σωστά τα παραπάνω με το νόημα που έχουν στην ελληνική γλώσσα.

0.3.1 Συμπερασματικοί Κανόνες

Για αξιώματα και θεωρήματα έχεις δει αρκετά στα μαθηματικά. Εδώ ας κάνουμε μια παρένθεση για να πούμε δύο λόγια για τους **συμπερασματικούς κανόνες** (inference rules). Όπως λέει και το όνομά τους, είναι κανόνες που σου λένε πώς μπορείς να βγάζεις συμπεράσματα όταν κάνεις τις αποδείξεις σου. Να ένα παράδειγμα από τον Κατηγορηματικό Λογισμό:

- Αν για κάθε x ισχύει η $P(x)$

τότε, μπορούμε να συμπεράνουμε ότι

- Ισχύει η $P(a)$ για τυχόν a.

Αυτό γράφεται συμβολικώς ως εξής:

$$\frac{\forall x \bullet P(x)}{P(a)}$$

Εδώ θα παρατηρήσεις ότι α) αυτό είναι αυτονόητο και τετριμμένο β) το χρησιμοποιείς χωρίς να ξέρεις ότι υπάρχει. Και τα δύο είναι σωστά: α) όπως τα αξιώματα, έτσι και οι συμπερασματικοί κανόνες είναι αυτονόητοι και τετριμμένοι (ή τουλάχιστον έτσι μας φαίνεται), β) ήδη αναφέραμε τη χρήση αυτού του κανόνα στο γεωμετρικό παράδειγμα που δώσαμε παραπάνω.

Ας δούμε ακόμη έναν κανόνα από τον Προτασιακό Λογισμό:

- Αν ισχύει η πρόταση P και
- Αν ισχύει η πρόταση $P \Rightarrow Q$, δηλαδή η P συνεπάγεται την Q,

τότε, μπορούμε να συμπεράνουμε ότι

- Ισχύει η Q.

Συμβολικώς (modus ponens):

$$\frac{P, P \Rightarrow Q}{Q}$$

Τώρα θα δούμε δύο συμπερασματικούς κανόνες που είναι χρήσιμοι για την απόδειξη ορθότητας προγραμμάτων.

Ας πούμε ότι έχεις να λύσεις το εξής πρόβλημα (θα το δούμε παρακάτω)[6]: Γράψε πρόγραμμα E τέτοιο ώστε:

$$(a == a_0) \;\&\&\; (b == b_0)\ \{\ E\ \}\ (a == b_0) \;\&\&\; (b == a_0)$$

Γράφεις το πρόγραμμα, αλλά στην απόδειξή του παίρνεις:

$$(a == a_0) \;\&\&\; (b == b_0)\ \{\ E\ \}\ (a == b_0) \;\&\&\; (b == a_0) \;\&\&\; (S == a_0)$$

Είναι σωστό το πρόγραμμα E ή όχι;

Είναι σωστό, διότι αφού απόδειξες ότι ισχύει η $(a == b_0) \;\&\&\; (b == a_0) \;\&\&\; (S == a_0)$ θα ισχύει και η $(a == b_0) \;\&\&\; (b == a_0)$, αφού ισχύει η $(A \;\&\&\; B) \Rightarrow A$.

Αυτό μας λέει ο παρακάτω συμπερασματικός **κανόνας του επακόλουθου** (rule of consequence):

- Αν μετά την εκτέλεση του προγράμματος E με προϋπόθεση P ισχύει η Q και
- Αν από η Q συνεπάγεται την R

τότε

[6] "==" σημαίνει «είναι ίσο με» και "&&" σημαίνει «και». Δες το Παράρτ. Α.

- Αν το πρόγραμμα E εκτελεσθεί με προϋπόθεση P μετά την εκτέλεση ισχύει η R.

Συμβολικώς:

$$\frac{P\{E\}\,Q,\ Q \Rightarrow R}{P\,\{E\}\,R} \qquad (E1)$$

Όπως θα δεις στη συνέχεια, πιο χρήσιμος θα μας είναι ένας άλλος παρόμοιος συμπερασματικός κανόνας:

- Αν η P συνεπάγεται την Q και
- Αν μετά την εκτέλεση του προγράμματος E με προϋπόθεση Q ισχύει η R

τότε

- Αν το πρόγραμμα E εκτελεσθεί με προϋπόθεση P μετά την εκτέλεση ισχύει η R.

Συμβολικώς:

$$\frac{P \Rightarrow Q,\ \ Q\,\{E\}\,R}{P\,\{E\}\,R} \qquad (E2)$$

Στις αποδείξεις μας θα χρησιμοποιούμε τους δύο αυτούς κανόνες –κυρίως τον ($E2$)– χωρίς να τους αναφέρουμε πάντοτε.

0.4 Από τις Προδιαγραφές στο Πρόγραμμα

Αν ενδιαφέρεσαι σοβαρώς για την Πληροφορική και την Τεχνολογία Λογισμικού, θα πρέπει να δώσεις ιδιαίτερη προσοχή στο θέμα «προδιαγραφές». Πολύ γρήγορα θα καταλάβεις ότι αυτό που λέμε «προγραμματισμός» ή «ανάπτυξη εφαρμογής» είναι στην πραγματικότητα επεξεργασία προδιαγραφών.

Πώς δουλεύει ο Μηχανικός Λογισμικού (software engineer); Από την ανάλυση των απαιτήσεων της εφαρμογής που έχει να αναπτύξει, διατυπώνει προδιαγραφές για τα προγράμματα και τις βάσεις δεδομένων που πρέπει να γίνουν.

Ας πάρουμε τώρα ένα από αυτά τα προγράμματα. Αρχικώς προδιαγράφεται όπως είδαμε παραπάνω:

$$\phi(x_1,\ldots x_m) < E > \psi(x_1,\ldots x_m, y_1,\ldots y_n)$$

Στη συνέχεια το αρχικό πρόβλημα διασπάται σε δύο ή περισσότερα υποπροβλήματα E_1, ..., E_N:

$\phi(x_1,\ldots x_m)$
$< E_1 >$
$\omega_1(x_1,\ldots x_m, y_1,\ldots y_n, z_1,\ldots, z_p)$
:
$\omega_{N-1}(x_1,\ldots x_m, y_1,\ldots y_n, z_1,\ldots, z_p)$
$< E_N >$
$\psi(x_1,\ldots x_m, y_1,\ldots y_n)$

Τα $z_1,\ldots, z_p$ είναι βοηθητικά αντικείμενα (μεταβλητές) που χρειάζονται για την ανάπτυξη του προγράμματος.

Όπως βλέπεις κάθε υποπρόβλημα έχει προϋπόθεση την απαίτηση του προηγούμενου[7]. Αυτή η διαδικασία συνεχίζεται με τη διάσπαση των E_1, ..., E_N σε άλλα «μικρότερα» κ.ο.κ. μέχρι να φτάσουμε –κατ' αρχήν– σε προβλήματα που λύνονται με μια εντολή (στην πραγματικότητα σταματούμε όταν καταλήξουμε σε προβλήματα που η λύση τους είναι απλή). Τότε έρχεται η ώρα της **κωδικοποίησης** (coding) σε κάποια γλώσσα προγραμματισμού.

Πώς καταλαβαίνουμε ότι η διαδικασία της διάσπασης των προβλημάτων σε απλούστερα έφτασε σε προβλήματα που λύνονται με μια εντολή; Σύμφωνα με όσα λέμε, αυτό θα πρέπει να φαίνεται από τις προδιαγραφές των «υποπροβλημάτων»· άρα θα πρέπει να

[7] Η διάσπαση που βλέπεις εδώ είναι *σειριακή*· μπορεί όμως να γίνει και *παράλληλη*.

έχουμε προδιαγραφές για τις εντολές. Πράγματι, το νοηματικό μέρος της γλώσσας μας δίνει ακριβώς αυτές τις προδιαγραφές των εντολών και τις χρησιμοποιούμε για να γράφουμε τα προγράμματά μας και για να αποδεικνύουμε ότι είναι σωστά.

Με τον καιρό, όσο θα γράφεις μεγαλύτερα και δυσκολότερα προγράμματα, θα καταλάβεις ότι το σημαντικό κομμάτι του προγραμματισμού είναι η διατύπωση των προδιαγραφών. Οι νεόκοποι προγραμματιστές κάνουν το λάθος να μην ξεχωρίζουν τη διατύπωση προδιαγραφών από την κωδικοποίηση πράγμα που έχει πολύ κακές επιπτώσεις στα προγράμματα που γράφουν. Οι παλιοί προγραμματιστές είχαν μια συνταγή για να τονίζουν αυτήν την ανάγκη: «*think first, code later*» ή, στα ελληνικά: «*πρώτα σκέψου και άφησε για αργότερα το γράψιμο των εντολών*».

Η διαδικασία που περιγράψαμε παραπάνω, λέγεται **βήμα-προς-βήμα ανάλυση** (step-by-step refinement) και στηρίζεται στο συμπερασματικό **κανόνα της σύνθεσης** (rule of composition):

- Αν μετά την εκτέλεση του προγράμματος E_1 με προϋπόθεση P ισχύει η Q και
- Αν μετά την εκτέλεση του προγράμματος E_2 με προϋπόθεση Q ισχύει η R

τότε

- Αν με προϋπόθεση P εκτελεσθούν πρώτα το E_1 και μετά το E_2, μετά την εκτέλεση ισχύει η R.

Συμβολικώς:

$$\frac{P\{E_1\}Q,\ Q\{E_2\}R}{P\{E_1;E_2\}R} \qquad (\Sigma)$$

Τελειώνοντας (προς το παρόν) να πούμε ότι υπάρχουν ειδικές γλώσσες για τη διατύπωση προδιαγραφών, όπως είναι η Z (Diller 1990), (Spivey 1988), η VDM (Andrews & Ince 1991), (Jones 1990) κ.ά., που στηρίζονται στη Μαθηματική Λογική (Κατηγορηματικό Λογισμό). Η Z είναι μια γλώσσα που δημιουργήθηκε αποκλειστικώς για τη διατύπωση προδιαγραφών λογισμικού (αλλά και υλικού). Η VDM (Vienna Development Method) είναι μέθοδος ανάπτυξης λογισμικού που περιλαμβάνει και εργαλεία για την αυστηρή διατύπωση προδιαγραφών. Για ειδικές κατηγορίες προβλημάτων έχουν αναπτυχθεί ειδικές γλώσσες.

0.5 Αποδοτικότητα Προγράμματος

Δες πώς αλλιώς μπορούμε να γράψουμε τον αλγόριθμο του μεγίστου:

```
km = 0;
for ( k = 1; k <= 9; k = k+1 )
   if ( a[k] > a[km] )  {  km = k;  }
m = a[km];
```

Εδώ ψάχνουμε τη θέση του μεγίστου· όταν τη βρούμε μπορούμε εύκολα να πάρουμε και το μέγιστο. Βλέπεις ότι αλγόριθμος είναι λίγο πιο «οικονομικός» από τον αρχικό (σύγκρινέ τους για να βρεις τις διαφορές.)

Γενικώς: ένα καλό πρόγραμμα δεν θα πρέπει να χρησιμοποιεί περισσότερη μνήμη ούτε να ζητάει την εκτέλεση περισσότερων εντολών από όσο χρειάζεται. Πρέπει δηλαδή να μην κάνει σπατάλη στη χρήση μνήμης και υπολογιστικού χρόνου.

Πολλοί προγραμματιστές, θέλοντας να συμμορφωθούν με αυτήν την αρχή, καταφεύγουν σε διάφορα *τεχνάσματα* με αποτέλεσμα: ένα *πρόγραμμα που δεν φαίνεται εύκολα πώς δουλεύει* και επομένως:

- δεν μπορεί να αποδειχτεί ότι είναι σωστό (και συχνά δεν είναι σωστό),
- δεν μπορεί να τροποποιηθεί.

Το πρόβλημα της βελτίστοποίησης του προγράμματος δεν λύνεται με τεχνάσματα αλλά *με έναν καλύτερο αλγόριθμο* που:

- αποδεικνύεται η ορθότητά του και
- δεν είναι σπάταλος σε υπολογιστικό χρόνο και χρήση μνήμης.

Βέβαια, όταν έχεις τον αλγόριθμο, υπάρχουν πολλοί τρόποι για να τον γράψεις σε κάποια γλώσσα προγραμματισμού. Οι τρόποι αυτοί διαφέρουν μεταξύ τους στις λεπτομέρειες. Ο καλός προγραμματιστής θα προσέξει τις λεπτομέρειες ώστε να αποφύγει διάφορες σπατάλες· αυτό λέγεται **μικροβελτιστοποίηση** (microoptimization). Η μικροβελτιστοποίηση είναι επιθυμητή *αλλά δεν έχει πρώτη προτεραιότητα*. Η πρώτη προτεραιότητα είναι το σωστό πρόγραμμα, δηλαδή το πρόγραμμα που αποδειγμένα ανταποκρίνεται στις προδιαγραφές του!

Σε μια εισαγωγή στον προγραμματισμό δεν ασχολείται κανείς και πολύ με τέτοια θέματα. Πάντως στη συνέχεια θα δεις μερικά απλά παραδείγματα που θα σου δείχνουν τι θα πει «καλύτερος αλγόριθμος» και τι θα πει «καλύτερη υλοποίηση» στις λεπτομέρειες.

0.6 Η Γλώσσα C++

Η γλώσσα C++ σχεδιάστηκε από τον B. Stroustrup με στόχο να δοθεί η δυνατότητα **αντικειμενοστρεφούς προγραμματισμού** (object-oriented programming) στους χρήστες της C. Είναι, κατ' αρχήν, υπερσύνολο της C (Kernighan & Ritsie 1978).

Η περιγραφή της γλώσσας δίνεται στο (Stroustrup 1997). Για την τυποποίησή της εργάσθηκαν σε συνεργασία οι επιτροπές ANSI X3J16 (του οργανισμού τυποποίησης των ΗΠΑ) και ISO WG21 (του διεθνούς οργανισμού τυποποίησης). Το πρότυπο (ISO/IEC 14882:1998) δόθηκε στη δημοσιότητα το 1998· θα δεις να αναφέρεται ως C++98. Επειδή το 2003 εγκρίθηκε μια τροποποίηση του προτύπου (ISO/ IEC 14882:2003) θα το δεις και ως C++03. Τον Σεπτέμβριο 2011 εγκρίθηκε το πρότυπο ISO/IEC 14882: 2011 και έχει το ανεπίσημο όνομα C++11.[8]

Η C++ έχει όλη την αποδοτικότητα της C όταν διαχειρίζεται δομές του υλικού (δυαδικό ψηφίο, ψηφιολέξη κλπ) ενώ από την άλλη μεριά δίνει στον προγραμματιστή τη δυνατότητα να υλοποιήσει **κλάσεις**. Ακόμη παρέχει και άλλες δυνατότητες που ευκολύνουν τη δουλειά του.

Στο C++11 υπάρχει πλήρες οπλοστάτιο για *πολυνηματικό σύνδρομο* προγραμματισμό ώστε να μπορούμε να γράψουμε προγράμματα που να εκμεταλλεύονται τις δυνατότητες που δίνουν τα σύγχρονα ΛΣ και οι σύγχρονοι επεξεργαστές.

Ήδη υπάρχουν αρκετοί μεταγλωττιστές που συμμορφώνονται σε μεγάλο ποσοστό με το πρότυπο C++03. Δύο από αυτούς δίνονται δωρεάν:

- Η Borland προσφέρει δωρεάν[9] τον μεταγλωττιστή (v.5.5) που χρησιμοποιεί στον C++ Builder. Συμμορφώνεται σε μεγάλο βαθμό με το πρότυπο της C++. Το πρόβλημα είναι ότι δεν έχει περιβάλλον ανάπτυξης και θα πρέπει να γράφεις τα προγράμματά σου στο Notepad και να τα περνάς με command line.
- Ο **g++** της GNU: Μπορείς να τον βρεις ενσωματωμένο σε περιβάλλον ανάπτυξης εφαρμογών (IDE) επίσης δωρεάν: α) Dev C++[10], β) Code::Blocks Studio[11]. Οι πειραματικές εκδόσεις του g++ που συμμορφώνονται με το C++11 υπάρχουν ήδη στο διαδίκτυο.[12]

---

[8] Το C++11 με ορισμένες τροποποιήσεις-συμπληρώσεις αναφέρεται ήδη ως C++14.

[9] `http://dn.codegear.com/article/20633`

[10] `http://www.bloodshed.net/`

[11] `http://www.codeblocks.org/`

[12] `http://gcc.gnu.org/projects/cxx0x.html`

0.7 Ο Συμβολισμός BNF

Για να έχουμε περισσότερη καθαρότητα και ακρίβεια στην περιγραφή του συντακτικού της γλώσσας, σε ορισμένες περιπτώσεις, θα χρησιμοποιούμε μια μορφή του γνωστού **συμβολισμού BNF** (Backus - Naur Form).

Ας ξεκινήσουμε με ένα

Παράδειγμα 1

Οι ακόλουθοι κανόνες δημιουργούν (ορίζουν) μια οντότητα με το όνομα *ψηφίο*, που μπορεί να είναι οποιοδήποτε από τα σύμβολα 0 ... 9· στη συνέχεια χρησιμοποιούμε το *ψηφίο* για να ορίσουμε την οντότητα *δεκαεξαδικό ψηφίο*.

ψηφίο = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
δεκαεξαδικό ψηφίο = *ψηφίο* |
"a" | "b" | "c" | "d" | "e" | "f" |
"A" | "B" | "C" | "D" | "E" | "F";

Όπως βλέπεις, ένας ορισμός οντότητας γράφεται ως εξής:

- γράφουμε το όνομα της οντότητας, π.χ. *ψηφίο*,
- μετά παραθέτουμε το "=" και
- τέλος γράφουμε τον κανόνα που ορίζει τη συντακτική δομή και τερματίζεται με ένα ";".

Ο κανόνας γράφεται ως εξής:

- όποτε εμφανίζεται ένα σύμβολο από το αλφάβητο της γλώσσας –**τελικό σύμβολο** (terminal symbol)– περικλείεται σε εισαγωγικά "...", π.χ. "2" ή "3",
- όταν κάποια οντότητα μπορεί να δημιουργηθεί με διάφορους τρόπους, οι διαφορετικές επιλογές διαχωρίζονται με το σύμβολο "|",
- για να καθορίσουμε ότι κάποια αντικείμενα εμφανίζονται με κάποια συγκεκριμένη σειρά τα παραθέτουμε χωρισμένα με κόμματα (",").

Παράδειγμα 2

Ο ακόλουθος κανόνας περιγράφει οποιονδήποτε από τους ορμαθούς "00", "01", ... "99", με βάση το ψηφίο που ορίσαμε παραπάνω.

διψήφιος αριθμός = *ψηφίο*, *ψηφίο*;

Για να μην ταλαιπωρούμαστε με πολύ γράψιμο χωρίς νόημα θα χρησιμοποιούμε και την παρακάτω σύμβαση συντόμευσης:

- αν δεν υπάρχει κίνδυνος παρεξήγησης τότε αν έχουμε μια ακολουθία (τελικών) συμβόλων που διαχωρίζονται με το σύμβολο "|" θα γράφουμε: το πρώτο, αποσιωποιητικά και το τελευταίο.

Έτσι, οι κανόνες του παραδ. 1 γράφονται:

ψηφίο = "0" | ... | "9";
δεκαεξαδικό ψηφίο = *ψηφίο* | "a" | ... | "f" | "A" | ... | "F";

Στους κανόνες μπορεί να υπάρχει και **αναδρομή** (recursion), δηλαδή στον ορισμό να χρησιμοποιούμε το οριζόμενο. Δες το:

Παράδειγμα 3

Να, τώρα, οι συντακτικοί κανόνες για να γράφουμε δεκαδικούς αριθμούς χωρίς πρόσημο:

δεκαδικός χωρίς πρόσημο = *ακέραιος χωρίς πρόσημο* |
κλασματικό μέρος |
ακέραιος χωρίς πρόσημο, *κλασματικό μέρος*;
κλασματικό μέρος = ".", *ακέραιος χωρίς πρόσημο*;

ακέραιος χωρίς πρόσημο = *ψηφίο* | *ακέραιος χωρίς πρόσημο*, *ψηφίο*;
ψηφίο = "**0**" | ... | "**9**";

Όπως βλέπεις, λέμε ότι ένας *ακέραιος χωρίς πρόσημο* μπορεί να είναι *ψηφίο* ή *ακέραιος χωρίς πρόσημο* ακολουθούμενος από *ψηφίο*. Δηλαδή: το "**133**" είναι *ακέραιος χωρίς πρόσημο*; Για να δούμε:

- Κατ' αρχάς δεν είναι *ψηφίο*. Θα πρέπει λοιπόν να είναι της μορφής *ακέραιος χωρίς πρόσημο*, *ψηφίο*. Πράγματι, το "**3**" είναι *ψηφίο*. Αν το υπόλοιπο ("**13**") είναι *ακέραιος χωρίς πρόσημο* είμαστε εντάξει.
- Και πάλι, το "**13**" δεν είναι *ψηφίο*. Θα πρέπει να είναι της μορφής *ακέραιος χωρίς πρόσημο*, *ψηφίο*. Το "**3**" είναι *ψηφίο*. Αν το υπόλοιπο ("**1**") είναι *ακέραιος χωρίς πρόσημο* είμαστε εντάξει.
- Το "**1**" όμως είναι *ψηφίο* άρα είναι *ακέραιος χωρίς πρόσημο*. Άρα και το "**13**" είναι *ακέραιος χωρίς πρόσημο*, επομένως και το "**133**" είναι *ακέραιος χωρίς πρόσημο*.

Αυτό είναι ένα παράδειγμα *αναδρομής στα αριστερά*. Αλλού μπορεί να δεις τον παραπάνω αναδρομικό ορισμό ως εξής:

ακέραιος χωρίς πρόσημο = *ψηφίο* | *ψηφίο*, *ακέραιος χωρίς πρόσημο*;

Αυτή είναι *αναδρομή στα δεξιά*. Εδώ θα χρησιμοποιούμε συνήθως αναδρομή στα αριστερά.

Τέλος, μέσα σε αγκύλες βάζουμε κάτι που μπορεί να υπάρχει μπορεί και όχι.

Παράδειγμα 4

Αντί να γράψουμε:

δεκαδικός = *πρόσημο*, *δεκαδικός χωρίς πρόσημο* |
δεκαδικός χωρίς πρόσημο;
πρόσημο = "**+**" | "**-**";

γράφουμε:

δεκαδικός = [*πρόσημο*] *δεκαδικός χωρίς πρόσημο*;
πρόσημο = "**+**" | "**-**";

Συχνά θα βλέπεις συντακτικούς ορισμούς οντοτήτων που δεν έχουν συγκεκριμένο φυσικό νόημα. Θα πρόκειται για ενδιάμεσους ορισμούς που χρησιμεύουν μόνον για να ολοκληρωθεί μια ακολουθία συντακτικών ορισμών. Μην τρομάζεις λοιπόν.