Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης
και Υπολογιστων

Υλοποίηση Μεταγλωττιστή Γλώσσας Προστακτικού Προγραμματισμού με Στοιχεία Ταυτοχρονισμού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΗΣΤΟΣ Ν. ΓΡΑΒΒΑΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Αθήνα, Οκτώβριος 2005

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης
και Υπολογιστων

Υλοποίηση Μεταγλωττιστή Γλώσσας Προστακτικού Προγραμματισμού με Στοιχεία Ταυτοχρονισμού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΗΣΤΟΣ Ν. ΓΡΑΒΒΑΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26η Οκτωβρίου 2005.

.....	.....	.....
Νικόλαος Παπασπύρου Λέκτορας Ε.Μ.Π.	Ανδρέας-Γεώργιος Σταφυλοπάτης Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2005

Χρήστος Ν. Γραββάνης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σκοπός της εργασίας είναι η μελέτη των στοιχείων ταυτοχρονισμού που συναντώνται στον προγραμματισμό και η κατασκευή ενός μεταγλωττιστή για μια γλώσσα προστακτικού προγραμματισμού που υποστηρίζει κάποια από αυτά τα χαρακτηριστικά. Η αρχική γλώσσα, που ονομάζεται CoPCL, είναι ένα υποσύνολο της Pascal εμπλουτισμένο με ταυτόχρονη εκτέλεση, συγχρονισμό και επικοινωνία. Η τελική γλώσσα είναι η συμβολική γλώσσα (assembly) των 32-bit επεξεργαστών της Intel.

Ο προγραμματισμός με στοιχεία ταυτοχρονισμού (concurrent programming) αποτελεί ένα σύνολο από σημειολογίες και τεχνικές προγραμματισμού για την έκφραση πιθανού ταυτοχρονισμού εργασιών και για την επίλυση των συνεπαγόμενων προβλημάτων του συγχρονισμού και της επικοινωνίας μεταξύ τους. Ο ταυτοχρονισμός είναι ανεξάρτητος από τα συστήματα παράλληλης επεξεργασίας, με τα οποία ασχολούνται τα θέματα υπολογιστικών συστημάτων υλικού και λογισμικού. Αποτελεί ένα σημαντικό είδος προγραμματισμού γιατί προσφέρει μια αφαιρετική εικόνα για την μελέτη του παραλληλισμού χωρίς την εμπλοκή σε λεπτομέρειες υλοποίησης. Η αφαιρετικότητα που προσφέρει αποτελεί πολύ χρήσιμο χαρακτηριστικό για την κατασκευή δομημένων και ορθών προγραμμάτων. Έτσι, οι σύγχρονες γλώσσες προγραμματισμού επιβάλλεται να υποστηρίζουν χαρακτηριστικά ταυτοχρονισμού για την ανάπτυξη πολύπλοκων εφαρμογών που ζητούν την αξιοποίηση των τεχνικών που παρέχει ο ταυτοχρονισμός.

Κατά την υλοποίηση του μεταγλωττιστή ακολουθήθηκαν τα στάδια της λεκτικής, συντακτικής, σημασιολογικής ανάλυσης, παραγωγής ενδιάμεσου και τελικού κώδικα. Δόθηκε έμφαση στην ενσωμάτωση των χαρακτηριστικών ταυτοχρονισμού που υποστηρίζονται από τις σύγχρονες γλώσσες προγραμματισμού, όπως είναι η επικοινωνία και ο συγχρονισμός των ταυτόχρονα εκτελούμενων εργασιών. Για την υλοποίηση των στοιχείων αυτών χρησιμοποιήθηκε η προτυποποιημένη βιβλιοθήκη συναρτήσεων πολυνηματισμού POSIX threads.

Λέξεις κλειδιά

Υλοποίηση γλωσσών προγραμματισμού, μεταγλωττιστές, ταυτοχρονισμός.

Abstract

The purpose of this diploma dissertation is the study of elements of concurrent programming and the implementation of a compiler for an imperative concurrent programming language, which implements some of the these elements. The source language, which is called CoPCL, is a subset of Pascal enhanced with concurrent execution, synchronization and communication. The target language is the assembly language for Intel's series of 32-bit processors.

Concurrent programming is a set of programming notations and techniques for expressing potential concurrency of tasks and for solving the resulting problems of synchronization and communication between them. Concurrency is independent of parallel processing systems, which is a topic of computing systems hardware and software research. It is an important paradigm of programming, offering an abstract view for studying parallelism without involving implementation details. The abstraction provided is a useful feature for the development of structured and correct programs. Hence, modern programming languages should support concurrency features for the development of complex applications, which can be best implemented by techniques provided by concurrency.

The compiler's implementation follows several stages: lexical, syntactic and semantic analysis, generation of intermediate and final code. Emphasis has been given to incorporate most of the characteristics that are supported by modern concurrent programming languages, such as communication and synchronization of the concurrent executed tasks. For the implementation of these features, the standard multithreading function library of POSIX threads was used.

Key words

Programming language implementation, compilers, concurrent programming.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου, κ. Νίκο Παπασπύρου για το ενδιαφέρον που έδειξε και για τον χρόνο που αφιέρωσε κατά την εκπόνηση αυτής της εργασίας. Θα ήθελα επίσης να ευχαριστήσω τους γονείς μου για την υποστήριξη και τη συμπαράσταση που μου προσέφεραν κατά τη διάρκεια των σπουδών μου, καθώς και τον αδελφό μου, Γιώργο, για την πρωτοβουλία που είχε πριν από 12 χρόνια, να ζητήσουμε από τους γονείς μας να αποκτήσουμε τον πρώτο μας προσωπικό υπολογιστή.

Χρήστος Ν. Γραββάνης,

Αθήνα, Οκτώβριος 2005.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-7-05, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2005.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Πίνακες

Σχήματα

Κεφάλαιο 1

Εισαγωγή

Ο προγραμματισμός με στοιχεία ταυτοχρονισμού (concurrent programming) αποτελεί ένα σύνολο από σημειολογίες και τεχνικές προγραμματισμού για την έκφραση πιθανού ταυτοχρονισμού εργασιών και για την επίλυση των συνεπαγόμενων προβλημάτων του συγχρονισμού και της επικοινωνίας μεταξύ τους. Ο ταυτοχρονισμός είναι ανεξάρτητος από τα συστήματα παράλληλης επεξεργασίας, με τα οποίο ασχολούνται τα θέματα υπολογιστικών συστημάτων υλικού και λογισμικού. Αποτελεί ένα σημαντικό είδος προγραμματισμού γιατί προσφέρει μια αφαιρετική εικόνα για την μελέτη του παραλληλισμού χωρίς την εμπλοκή σε λεπτομέρειες υλοποίησης. Η αφαιρετικότητα που προσφέρει αποτελεί πολύ χρήσιμο χαρακτηριστικό για την κατασκευή δομημένων και ορθών προγραμμάτων. Έτσι, οι σύγχρονες γλώσσες προγραμματισμού επιβάλλεται να υποστηρίζουν χαρακτηριστικά ταυτοχρονισμού για την ανάπτυξη πολύπλοκων εφαρμογών που ζητούν την αξιοποίηση των τεχνικών που παρέχει ο ταυτοχρονισμός.

1.1 Σκοπός

Σκοπός της εργασίας είναι η μελέτη των βασικών χαρακτηριστικών και τεχνικών του ταυτοχρονισμού και η κατασκευή ενός μεταγλωττιστή για μια γλώσσα προστακτικού προγραμματισμού που θα υποστηρίζει κάποια στοιχεία ταυτοχρονισμού. Η γλώσσα που δημιουργήθηκε ονομάστηκε CoPCL, που σημαίνει Concurrent PC Langugage Αποτελεί επέκταση της εκπαιδευτικής γλώσσας PCL η οποία περιγράφεται στο βιβλίο αναφοράς [2] και αποτελεί μία απλή γλώσσα χρήσιμη για την μελέτη των σταδίων κατασκευής μεταγλωττιστών.

1.2 Σύνοψη της εργασίας

Στο πλαίσιο της υλοποίησης του μεταγλωττιστή για τη γλώσσα CoPCL ακολουθήθηκαν τα στάδια της λεκτικής, συντακτικής, σημασιολογικής ανάλυσης, καθώς και της παραγωγής ενδιάμεσου και τελικού κώδικα. Η συνέχεια της διπλωματικής εργασίας έχει την ακόλουθη δομή:

Κεφάλαιο 2. Περιγραφή του προγραμματισμού με στοιχεία ταυτοχρονισμού.

Κεφάλαιο 3. Ορισμός και περιγραφή της γλώσσας CoPCL.

Κεφάλαιο 4. Περιγραφή της υλοποίησης και μεταγλώττισης της CoPCL.

Κεφάλαιο 5. Παραδείγματα προγραμμάτων της CoPCL με ενδιάμεσο και τελικό κώδικα.

Κεφάλαιο 6. Συμπεράσματα της εργασίας και προτάσεις μελλοντικής χρήσης και επέκτασης του μεταγλωττιστή.

Κεφάλαιο 2

Προγραμματισμός με στοιχεία ταυτοχρονισμού

Στο *διαδικαστικό μοντέλο* προγραμματισμού κάθε πρόγραμμα αντιμετωπίζεται ως μια σειρά εντολών που εκτελούνται διαδοχικά. Με τη χρήση των στοιχείων του *ταυτοχρονισμού* επιτυγχάνεται ο προγραμματισμός ταυτόχρονων εργασιών με άμεσο πλεονέκτημα την ευκολότερη ανάπτυξη πηγαίου κώδικα για αλγορίθμους και εφαρμογές που εμπεριέχουν την έννοια της ταυτόχρονης εξέλιξης των διαδικασιών που τις αποτελούν.

Ένα σειριακό πρόγραμμα ορίζει την ακολουθία εκτέλεσης των εντολών μίας *εργασίας*, ενώ η εκτέλεσή τους αποτελεί μία *διεργασία* (process). Στην περίπτωση του ταυτοχρονισμού ορίζονται περισσότερες από μία ταυτόχρονες εργασίες, καθεμία από τις οποίες εκτελείται είτε ως μία ξεχωριστή διεργασία είτε ως ένα επιπλέον νήμα εκτέλεσης (thread) που ανήκει στο συνολικό χώρο της κύριας διεργασίας. Για να αξιοποιηθεί αποτελεσματικά η έννοια του ταυτοχρονισμού απαιτείται η χρήση των κατάλληλων δομών για τον *έλεγχο* (control structures) των εργασιών που προγραμματίζονται, καθώς και η εξασφάλιση της μεταξύ τους *επικοινωνίας* (communication) και του *συγχρονισμού* (synchronization) της εκτέλεσής τους, είτε μέσω *κοινής μνήμης* (shared memory) είτε με *πέρασμα μηνυμάτων* (message passing). Δημοφιλείς γλώσσες που ικανοποιούν αυτές τις προδιαγραφές είναι η Java και η Ada, ενώ υπάρχει μια πληθώρα από βιβλιοθήκες που έχουν αναπτυχθεί για την κάλυψη των αναγκών του μοντέλου του ταυτοχρονισμού, όπως για παράδειγμα τα POSIX Threads για την υλοποίηση πολυνηματισμού. Στο κεφάλαιο αυτό θα παρουσιαστούν διάφορες εκδοχές για τα βασικά ζητήματα και τις τεχνικές του μοντέλου του ταυτοχρονισμού που συναντώνται συχνά και προσδίδουν ταυτόχρονο χαρακτήρα στον προγραμματισμό.

2.1 Δομές ελέγχου ταυτόχρονης εκτέλεσης εργασιών

Κατά την εξέλιξη του μοντέλου του ταυτοχρονισμού έχουν προταθεί διάφορες *δομές ελέγχου* (control structures) σκοπός των οποίων είναι να δώσουν τη δυνατότητα στον προγραμματιστή να καταστρώσει την ταυτόχρονη εκτέλεση των εργασιών από τις οποίες αποτελείται το πρόγραμμά του. Τρεις από αυτές είναι τα ζεύγη εντολών fork-join και cobegin-coend, καθώς και η ενσωμάτωση των εντολών κάθε εργασίας σε μία δική της ξεχωριστή υπομονάδα. Μία υπομονάδα της τρίτης περίπτωσης συνήθως αποκαλείται process.

2.1.1 Εντολές fork-join

Οι εντολές **fork** και **join** προτάθηκαν από τον Conway το 1963 και τους Dennis και Van Horn το 1966. Με την εντολή fork καθορίζεται η έναρξη της εκτέλεσης μίας ρουτίνας κώδικα, όπως ακριβώς η κλήση ενός υποπρογράμματος σε ένα σειριακό πρόγραμμα. Παρόλ' αυτά τόσο η κληθείσα όσο και η καλούσα ρουτίνα συνεχίζουν την εκτέλεσή τους ταυτόχρονα. Για να εξασφαλισθεί ο συγχρονισμός τους, μετά το πέρας της κληθείσας ρουτίνας, δίνεται η δυνατότητα στην καλούσα ρουτίνα να χρησιμοποιήσει την εντολή join. Σε περίπτωση που η εκτέλεση της κληθείσας ρουτίνας δεν έχει ολοκληρωθεί, τότε η εκτέλεση της ρουτίνας που εκτέλεσε το join μπλοκάρεται έως ότου ειδοποιηθεί ότι η κληθείσα τερματίστηκε.

```
program P1;          program P2;
...                    ...
fork P2;               ...
...                    ...
join P2;               end
...
```

Σχήμα 2.1: Παράδειγμα *fork-join*.

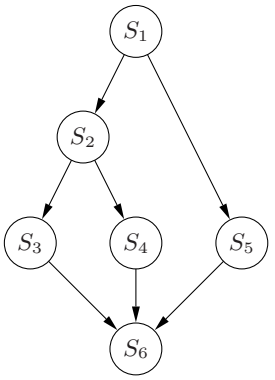
Στο σχήμα 2.1 φαίνεται ένα παράδειγμα χρήσης των fork και join. Η ετέλεση της ρουτίνας P_2 ξεκινάει όταν εκτελείται η εντολή fork στο σώμα της ρουτίνας P_1. Οι δύο ρουτίνες εκτελούνται ταυτόχρονα μέχρι είτε να τερματιστεί η P_2 είτε να εκτελέσει η P_1 την εντολή join, με την οποία θα μπλοκαριστεί έως ότου η P_2 φτάσει στο τέλος της. Αν και σε προγράμματα με πολλές διακλαδώσεις και βρόχους απαιτείται προσεκτική ανάλυση της ροής ελέγχου για την κατανόηση των ταυτόχρονων εκτελέσεων, η χρήση των fork και join αποτελεί ένα ισχυρό εργαλείο για την δυναμική δημιουργία διεργασιών, ενώ επιτρέπει και πολλαπλές εκτελέσεις του κώδικα της ίδιας ρουτίνας ταυτόχρονα. Οι εντολές αυτές είναι υλοποιημένες στις κλήσεις συστήματος του λειτουργικού συστήματος Unix, ενώ είναι ενσωματωμένες σε διάφορες γλώσσες προγραμματισμού, όπως η PL/I.

2.1.2 Δομή cobegin...coend

Ένας δεύτερος τρόπος για να καθορίσουμε την ταυτόχρονη εκτέλεση εργασιών είναι με τη χρήση της δομής **cobegin**...**coend**. Οι εντολές cobegin (**co**ncurrent **begin**) και coend (**co**ncurrent **end**), που προτάθηκαν αρχικά από τον Dijkstra[1], υποδεικνύουν την ταυτόχρονη εκτέλεση ενός συνόλου εντολών. Η εκτέλεση της εντολής:

$$\textbf{cobegin } S_1 \parallel S_2 \parallel \cdots \parallel S_n \textbf{ coend}$$

προκαλεί την ταυτόχρονη εκτέλεση των εντολών S_1, S_2, ..., S_n. Μετά τον τερματισμό όλων των εντολών S_i $(1 \leq i \leq n)$ η εκτέλεση συνεχίζει με την εντολή που ακολουθεί.




```
...
S1;
cobegin
   begin
      S2; cobegin S3 || S4 coend
   end ||
   S5
coend;
S6;
...
```

Σχήμα 2.2: Παράδειγμα *cobegin-coend*.

Στο σχήμα 2.2 φαίνεται ένα παράδειγμα χρήσης της δομής cobegin, μαζί με τον ισοδύναμο *γράφο προήγησης* (precedence graph). Αρχικά εκτελείται η εντολή S_1 και μετά το τέλος της δημιουργούνται δύο διεργασίες οι οποίες ξεκινούν να εκτελούνται ταυτόχρονα. Η πρώτη εκτελεί τη φωλιασμένη δομή cobegin, ενώ η δεύτερη εκτελεί την εντολή S_5. Στη φωλιασμένη δομή cobegin τίθενται σε ταυτόχρονη εκτέλεση οι εντολές S_3 και S_4. Αφού τερματιστούν οι S_3, S_4

[1] Η ονομασία που δόθηκε στις εντολές αρχικά από τον Dijkstra ήταν **parbegin** (**par**allel **begin**) και **parend** (**par**allel **end**).

και S_5, τότε συνεχίζεται η εκτέλεση με την εντολή S_6. Όπως γίνεται αντιληπτό, η δομή cobegin περιορίζεται σε μία είσοδο και έξοδο σε αυτήν, με αποτέλεσμα να προσφέρει τη δυνατότητα να εκφραστεί με σαφήνεια ποια κομμάτια κώδικα εκτελούνται ταυτόχρονα. Παρόλο που δε διαθέτει το δυναμικό χαρακτήρα των fork και join, αποτελεί εύχρηστο και αποτελεσματικό εργαλείο για την υλοποίηση πολλών ταυτόχρονων υπολογισμών και έχει ενσωματωθεί από διάφορες γλώσσες προγραμματισμού στο παρελθόν, όπως στην ALGOL68.

2.1.3 Δήλωση διεργασιών

Μία τρίτη μέθοδος για αποτελεσματικότερο έλεγχο της ταυτόχρονης εκτέλεσης των κομματιών κώδικα ενός προγράμματος είναι η δόμηση μεγάλων προγραμμάτων σε ειδικές σειριακές ρουτίνες οι οποίες πρόκειται να εκετελεστούν ταυτόχρονα. Παρόλο που τέτοιου είδους ρουτίνες θα μπορούσαν να οριστούν ως συνηθισμένα υποπρογράμματα και στη συνέχεια να εκτελεστούν με τη χρήση των δομών fork-join ή cobegin, η εξειδικευμένη δήλωσή τους επιτυγχάνει καλύτερη δομική σαφήνεια στο συνολικό πρόγραμμα και καθορίζεται αυστηρά σε ποια κομμάτια κώδικα πρόκειται να ζητηθεί ταυτόχρονη εκτέλεση. Μία τέτοια ρουτίνα ονομάζεται *διεργασία* (process).

Η *δήλωση διεργασιών* βρίσκεται ενσωματωμένη σε διάφορες γλώσσες προγραμματισμού, όπως οι Concurrent Pascal, Modula και Ada. Σε ορισμένες περιπτώσεις καθεμία από τις δηλωμένες διεργασίες έχει κατά τη διάρκεια της εκτέλεσης ένα μοναδικό στιγμειότυπο το οποίο δημιουργείται στην αρχή εκτέλεσης του προγράμματος. Εναλλακτικά, σε άλλες περιπτώσεις δίνεται η δυνατότητα μέσω της εντολής fork να υπάρχουν πολλαπλά στιγμειότυπα της ίδιας διεργασίας σε ένα πρόγραμμα, τα οποία δημιουργούνται άλλοτε κατά την αρχικοποίηση της εκτέλεσης του προγράμματος και άλλοτε με δυναμικό τρόπο κατά τη διάρκεια της εκτέλεσης.

Στο σχήμα 2.3 φαίνεται ένα παράδειγμα δήλωσης διεργασιών σε ένα πρόγραμμα συστήματος παραγωγού-καταναλωτή. Τόσο η μία όσο και η άλλη υπορουτίνα δηλώνονται ως διεργασίες με αποτέλεσμα να καθορίζεται ο ταυτόχρονος χαρακτήρας εκτέλεσής τους. Με τον τρόπο αυτό, κάποια άλλη υπορουτίνα που δεν έχει δηλωθεί ως διεργασία δεν έχει δυνατότητα να εκτελεστεί ταυτόχρονα με κάποιο άλλο μέρος του προγράμματος.

```
program system;
  var x : data;

  process producer;
  begin
    loop
      produce x
    end
  end;

  process consumer;
  begin
    loop
      consume x
    end
  end;

  ...
```

Σχήμα 2.3: Παράδειγμα *process declaration*.

2.2 Επικοινωνία διεργασιών

Κατά την ταυτόχρονη εκτέλεση δύο ή περισσότερων εργασιών στο ίδιο πρόγραμμα προκύπτουν ανάγκες επικοινωνίας μεταξύ τους. Οι κυριότεροι λόγοι για την επικοινωνία τους είναι:

- Η ανταλλαγή πληροφοριών για τους υπολογισμούς που υλοποιούνται
- Ο συγχρονισμός των λειτουργιών που εκτελούνται

Η *ανταλλαγή πληροφοριών* μεταξύ των ταυτόχρονων διεργασιών καλύπτει την ανάγκη που προκύπτει όταν κατά την εκτέλεση ενός αλγορίθμου απαιτούνται από μια διεργασία αποτελέσματα υπολογισμών που έχουν προκύψει από κάποια άλλη. Παράλληλα, ανακύπτει η ανάγκη του *συγχρονισμού* των υπολογισμών των διεργασιών με σκοπό τη σωστή συνεργασία τους στην εκτέλεση ενός προγράμματος. Στο παράδειγμα του παραγωγού-καταναλωτή που παρουσιάστηκε στο σχήμα 2.3 ο παραγωγός παράγει δεδομένα σε μία ενδιάμεση μεταβλητή και ο καταναλωτής λαμβάνει τα δεδομένα και τα χρησιμοποιεί. Αν ο απομονωτής δεν έχει προλάβει να γράψει τα παραγόμενα δεδομένα, τότε δεν πρέπει να επιτραπεί στον καταναλωτή να διαβάσει τα δεδομένα. Στην περίπτωση αυτή απαιτείται να υπάρχει μία μέθοδος επικοινωνίας για την ανταλλαγή δεδομένων, καθώς και ένα μοντέλο συγχρονισμού για να εκτελούνται οι λειτουργίες σε ορθή σειρά. Δύο μέθοδοι επικοινωνίας και συγχρονισμού μεταξύ ταυτόχρονων διεργασιών είναι η χρήση κοινής μνήμης (shared memory) και το πέρασμα μηνυμάτων (message passing).

2.2.1 Κοινή μνήμη

Στην περίπτωση χρήσης κοινών μεταβλητών για την επικοινωνία μεταξύ των διεργασιών η ανταλλαγή πληροφοριών μεταξύ τους επιτυγχάνεται μέσω εγγραφής και ανάκτησης δεδομένων από την κοινή μνήμη. Παράλληλα για την ορθότητα των προγραμμάτων υπάρχουν δύο τύποι συγχρονισμού που πρέπει να εξασφαλιστούν. Πρόκειται για τον αμοιβαίο αποκλεισμό (mutual exclusion) και τον συγχρονισμό συνθήκης (condition synchronization). Ο *αμοιβαίος αποκλεισμός* επιτρέπει σε στοιχειώδεις λειτουργίες να εκτελούνται αδιαίρετα. Μία τέτοια σειρά λειτουργιών που πρέπει να εκτελεστούν χωρίς διακοπή αποκαλείται *κρίσιμο τμήμα* (critical section). Αν διεργασίες εκτελούν ταυτόχρονες λειτουργίες χρησιμοποιώντας κοινές μεταβλητές, τότε μπορεί να προκύψουν απρόβλεπτα αποτελέσματα. Με τον αμοιβαίο αποκλεισμό των κρίσιμων τμημάτων ενός προγράμματος επιτυγχάνεται η ακέραια εκτέλεσή τους και η ορθότητα των αποτελεσμάτων των υπολογισμών τους.

Εξάλλου, υπάρχουν περιπτώσεις στις οποίες κάποια επιμέρους λειτουργία επιτρέπεται να εκτελεστεί μόνο αν ισχύει κάποια ορισμένη συνθήκη για τα δεδομένα που μοιράζεται με τις άλλες διεργασίες. Στην περίπτωση αυτή η διεργασία πρέπει να περιμένει να επαληθευθεί η απαραίτητη συνθήκη για να προχωρήσει στην εκτέλεση της λειτουργίας της. Η ανάγκη αυτή για συγχρονισμό είναι ο *συγχρονισμός συνθήκης*. Η περίπτωση που εξετάστηκε στο παράδειγμα παραγωγού-καταναλωτή απαιτεί και τα δύο είδη συγχρονισμού. Αφενός πρέπει να αποφευχθεί η ταυτόχρονη εκτέλεση εγγραφής και ανάκτησης της κοινής μεταβλητής, αφετέρου δεν πρέπει να επιτρέπεται εγγραφή αν δεν έχει ανακτηθεί η τιμή της προηγούμενης εγγραφής και ανάκτηση αν δεν έχει προηγηθεί εγγραφή.

Απασχόληση-Αναμονή

Μία μέθοδος για την υλοποίηση του συγχρονισμού κοινής μνήμης είναι με τον έλεγχο των κοινών μεταβλητών. Η διεργασία που αποστέλλει ειδοποίηση θέτει την τιμή μίας κοινής μεταβλητής, ενώ η διεργασία που περιμένει να ειδοποιηθεί ελέγχει συνεχώς το περιεχόμενό της έως ότου αποκτήσει την επιθυμητή τιμή. Η μέθοδος αυτή ονομάζεται *απασχόληση-αναμονή* (busy-wait) και ενώ είναι ικανή για να επιτευχθεί συγχρονισμός συνθήκης, δεν είναι αρκετή για τον αμοιβαίο αποκλεισμό κρίσιμων τμημάτων. Για τον αμοιβαίο αποκλεισμό χρησιμοποιείται σε

```
cobegin
  loop (* Διεργασία 1 *)
    enter1 := true;
    turn := 2;
    while enter2 and (turn = 2) do skip;
    (* Κρίσιμο τμήμα διεργασίας 1 *)
    enter1 := false;
    (* Μη κρίσιμο τμήμα διεργασίας 1 *)
  end ||
  loop (* Διεργασία 2 *)
    enter2 := true;
    turn := 1;
    while enter1 and (turn = 1) do skip;
    (* Κρίσιμο τμήμα διεργασίας 2 *)
    enter2 := false;
    (* Μη κρίσιμο τμήμα διεργασίας 2 *)
  end
coend
```

Σχήμα 2.4: Αμοιβαίος αποκλεισμός με *αναμονή-απασχόληση*.

συνεργασία με συγκεκριμένα πρωτόκολλα εισόδου και εξόδου από το κρίσιμο τμήμα. Η αρχική λύση δόθηκε από τον Dekker και στη συνέχεια απλοποιήθκε από τον Peterson (σχήμα 2.4).

Όπως φαίνεται στον αλγόριθμο του σχήματος 2.4 χρησιμοποιούνται τρεις κοινές μεταβλητές. Οι λογικές μεταβλητές enter1 και enter2 είναι true κατά τη διάρκεια της εκτέλεσης του πρωτοκόλλου εισόδου και του κρίσιμου τμήματος των διεργασιών 1 και 2, αντίστοιχα. Η τρίτη κοινή μεταβλητή turn καταγράφει την επόμενη διεργασία που έχει άδεια να εκτελέσει το κρίσιμο τμήμα της, ενώ εξασφαλίζει τον συγχρονισμό στην περίπτωση που και οι δύο διεργασίες εκτελούν το πρωτόκολλο εισόδου τους την ίδια στιγμή. Ενώ ο αλγόριθμος είναι αποτελεσματικός, παρουσιάζει μειονεκτήματα απόδοσης επειδή η απασχόληση-αναμονή καταναλώνει άσκοπα πόρους για την αναμονή, ενώ καθιστά το πρόγραμμα δύσκολο σε σχεδιασμό και κατανόηση. Τα προβλήματα αυτά προσπερνώνται με τη χρήση του σχήματος των σηματοφορέων.

Σηματοφορείς

Για να αποφευχθεί η σπατάλη χρόνου που προκαλεί η μέθοδος αναμονή-απασχόληση ο Dijkstra εισήγαγε τη χρήση των *σηματοφορέων* (semaphores). Ένας *σηματοφορέας* αποτελείται από μια δομή που περιλαμβάνει μια ακέραια μεταβλητή n και υποστηρίζει δύο πράξεις τις οποίες ο Djkstra ονόμασε **p** και **v**[2]. Οι δύο αυτές πράξεις πάνω σε ένα σηματοφορέα ορίζονται ως εξής:

- Όταν μια διεργασία επιθυμεί να εκτελέσει την πράξη p, ελέγχει την τιμή του n και αν $n \geq 1$, η πράξη p εκτελείται ελαττώνοντας την τιμή του n κατά 1, αλλιώς η διεργασία περιμένει έως ότου η τιμή του n γίνει μεγαλύτερη ή ίση του 1.

- Μια διεργασία εκτελεί την πράξη v στο σηματοφορέα αυξάνοντας την τιμή του n κατά 1.

Στο σχήμα 2.5 φαίνεται η απλοποίηση του αμοιβαίου αποκλεισμού με τη χρήση του σηματοφορέα mutex. Η αρχική τιμή του σηματοφορέα είναι 1 και όταν κάποια διεργασία μπει στο κρίσιμο τμήμα της εκτελώντας την πράξη p, η τιμή του γίνεται 0, με αποτέλεσμα αν προσπαθήσει η άλλη διεργασία να μπει στο κρίσιμο τμήμα της να μπλοκαριστεί έως ότου εκτελεστεί η εντολή v και γίνει η τιμή του σηματοφορέα 1. Με τους σηματοφορείς επίσης αποφεύγεται το μειονέκτημα της απασχόλησης-αναμονής, διότι οι πράξεις p και v μπορούν να υλοποιηθούν σαν ανεξάρτητες ατομικές λειτουργίες χωρίς απαίτηση κατανάλωσης πόρων κατά την αναμονή.

[2] Ο Dijkstra ονόμασε τις πράξεις p και v από τις ολλανδικές λέξεις passeren και vrygeven που σημαίνουν «περνώ» και «απελευθερώνω» αντίστοιχα.

```
mutex.n := 1;
cobegin
  loop (* Διεργασία 1 *)
    p(mutex);
    (* Κρίσιμο τμήμα διεργασίας 1 *)
    v(mutex);
    (* Μη κρίσιμο τμήμα διεργασίας 1 *)
  end ||
  loop (* Διεργασία 2 *)
    p(mutex);
    (* Κρίσιμο τμήμα διεργασίας 2 *)
    v(mutex);
    (* Μη κρίσιμο τμήμα διεργασίας 2 *)
  end
coend
```

Σχήμα 2.5: Αμοιβαίος αποκλεισμός με χρήση *σηματοφορέα*.

Εντούτοις οι σηματοφορείς μπορούν να οδηγήσουν σε δυσνόητο κώδικα πολύπλοκων προγραμμάτων, οπότε σε γλώσσες υψηλού επιπέδου χρησιμοποιούνται συχνά η δομή «κρίσιμο τμήμα» και οι επόπτες.

Η δομή «κρίσιμο τμήμα»

Η δομή *κρίσιμο τμήμα* (critical region) εισήχθη από τον Hoare για την ενσωμάτωσή της σε γλώσσες υψηλού επιπέδου. Το όνομά της προκύπει από τη χρήση της για τον αμοιβαίο αποκλεισμό διεργασιών, ενώ κατά την μεταγλώττιση της δομής χρησιμοποιούνται οι προηγούμενες τεχνικές. Η δομή κρίσιμο τμήμα έχει τη μορφή:

region *resource* **do** s;

όπου *resource* (πόρος) είναι μία κοινή δομή δεδομένων και s οι εντολές του κρίσιμου τμήματος κάποιας από τις διεργασίες που χρησιμοποιούν τον κοινό πόρο. Με τον τρόπο αυτό εξασφαλίζεται ότι κάθε στιγμή εκτελείται το κρίσιμο τμήμα μίας μόνο διεργασίας από αυτές που χρησιμοποιούν τους κοινούς πόρους.

Ο Hoare εισήγαγε και τη δομή *κρίσιμο τμήμα υπό συνθήκη* (conditional critical region) για το συγχρονισμό διεργασιών. Η μορφή της είναι

region *resource* **when** c **do** s;

όπου c είναι μια λογική έκφραση. Η είσοδος στο κρίσιμο τμήμα s μπορεί να γίνει μόνο εφόσον η συνθήκη c είναι αληθής. Αν είναι ψευδής τότε η διεργασία μπλοκάρεται έως ότου γίνει αληθής, μέσω κάποιας άλλης διεργασίας που έχει πρόσβαση στις κοινές μεταβλητές που επηρεάζουν την τιμή της λογικής έκφρασης c.

Στο σχήμα 2.6 φαίνεται το παράδειγμα παραγωγού-καταναλωτή με τη χρήση της δομής «κρίσημο τμήμα υπό συνθήκη». Αρχικά δηλώνεται ένα resource το οποίο περιλαμβάνει τις κοινές μεταβλητές. Στα σημεία που πρέπει να εκτελεστεί κρίσιμο τμήμα χρησιμοποιούνται κατάλληλες συνθήκες έτσι ώστε να εξασφαλιστεί ότι ούτε διαβάζονται δεδομένα από άδεια μεταβλητή, αλλά και ούτε εγγράφονται δεδομένα αν είναι ήδη γεμάτη η μεταβλητή.

Επόπτες

Ο *επόπτης* (monitor) έχει προταθεί ως τεχνική για να ξεπεραστούν τα μειονεκτήματα δυσνόητου κώδικα που υπάρχουν στις προηγούμενες τεχνικές. Αποτελεί μια συλλογή από κοινές μεταβλητές και διαδικασίες που υλοποιούν τις λειτουργίες που προσπελαύνουν τις μεταβλητές

```
program system;
  var x : data;
  resource r : x;

  process producer;
  begin
    loop
      region r when x.nonfull do
        produce x
    end
  end;

  process consumer;
  begin
    loop
      region r when x.nonempty do
        consume x
    end
  end;
```

Σχήμα 2.6: Παράδειγμα χρήσης δομών *«κρίσημο τμήμα υπό συνθήκη»*.

αυτές. Συνεπώς, ο επόπτης αποτελεί μια μορφή *ενθυλάκωσης* όπως αυτή εφαρμόζεται στο αντικειμενοστρεφές μοντέλο προγραμματισμού. Με την τεχνική αυτή απομονώνονται τα κομμάτια κώδικα που πρόκειται να εκτελεστούν ταυτόχρονα και προσδίδεται αυστηρότητα στο σχεδιασμό του προγράμματος και σαφήνεια κατά την κατανοησή του.

Ο αμοιβαίος αποκλεισμός εξασφαλίζεται με το να επιτρέπεται κάθε φορά μόνο μία διεργασία να εκτελεί διαδικασίες που ανήκουν στον ίδιο επόπτη. Ο συγχρονισμός της εκτέλεσης των διαδικασιών εξασφαλίζεται μέσω των λειτουργιών *wait* και *signal* που εφαρμόζονται πάνω σε ειδικές για το σκοπό αυτό μεταβλητές που ονομάζονται *μεταβλητές συνθήκης* (condition variables) και οι οποίες ορίζονται μέσα στον επόπτη. Η εκτέλεση της εντολής wait πάνω σε μία μεταβλητή συνθήκης μέσα από τον επόπτη έχει σαν αποτέλεσμα η διεργασία που την εκτελεί να απελευθερώνει τον επόπτη και να μπλοκάρεται έως ότου κάποια άλλη διεργασία να δεσμεύσει τον επόπτη και να εκτελέσει την εντολή signal πάνω στην μεταβλητή αυτή. Η εντολή signal έχει ως αποτέλεσμα την επανεκκίνηση της εκτέλεσης μιας διεργασίας που περίμενε στη μεταβλητή συνθήκης μετά από εκτέλεση της wait πάνω σε αυτή.

```
monitor resource;
  var busy : boolean := false;
      nonbusy : condition;

  procedure acquire;
  begin
    if busy then wait nonbusy;
    busy := true
  end;

  procedure release;
  begin
    busy := false;
    signal nonbusy
  end;
```

Σχήμα 2.7: Παράδειγμα χρήσης της δομής του *επόπτη*.

Στο σχήμα 2.7 παρουσιάζεται ένα απλό παράδειγμα στο οποίο διεργασίες μοιράζονται έναν πόρο δυναμικά. Όταν οι διεργασίες επιθυμούν να δεσμεύσουν τον πόρο καλούν τη διαδικασία *acquire* και όταν δεν το χρειάζονται πια, καλούν τη διαδικασία *release*. Μία λογική μεταβλητή busy χρησιμοποιείται για να περιγραφεί η κατάσταση του πόρου. Αν η τιμή της είναι true αυτό σημαίνει ότι είναι δεσμευμένος, αλλιώς αν είναι ελεύθερος η τιμή της είναι false. Όταν μια διεργασία καλέσει την acquire και ο πόρος είναι ελεύθερος τότε η διεργασία θα αδρανοποιηθεί και θα περιμένει σε μια ουρά της μεταβλητής nonbusy έως ότου ενεργοποιηθεί μέσω της εντολής signal. Αυτό θα γίνει όταν κάποια άλλη διεργασία δεν χρειάζεται πλέον τον πόρο και καλέσει την release. Από το παράδειγμα φαίνεται η αντιστοιχία των εντολών wait και signal με τις εντολές p και v στους σηματοφορείς. Οι επόπτες έχουν ενσωματωθεί με τη μορφή αυτή σε διάφορες γλώσσες υψηλού επιπέδου, όπως η Modula και η Concurrent Pascal, ενώ τα χαρακτηριστικά των εποπτών βρίσκονται ενσωματωμένα στις δομές κλάσεων που χρησιμοποιούνται από δημοφιλείς αντικειμενοστρεφείς γλώσσες, όπως η Java.

2.2.2 Πέρασμα μηνυμάτων

Όλες οι τεχνικές επικοινωνίας που παρουσιάστηκαν ως τώρα βασίζονται στη χρήση κοινών μεταβλητών οπότε είναι κατάλληλες για υλοποίηση σε αρχιτεκτονικές που διαθέτουν κοινή μνήμη. Αντίθετα το πέρασμα μηνυμάτων (message passing) δεν προϋποθέτει τη χρήση κοινής μνήμης, οπότε αποτελεί πολύ χρήσιμη τεχνική για την επικοινωνία σε παράλληλες αρχιτεκτονικές με κατανεμημένη μνήμη. Οι σηματοφορείς και οι επόπτες, που εξετάστηκαν, στηρίζονται στην ιδέα ότι η ανταλλαγή πληροφοριών και ο συγχρονισμός μεταξύ διεργασιών μπορούν να υλοποιηθούν με τη βοήθεια κοινών μεταβλητών που μπορούν να προσπελαύνονται από όλες τις διεργασίες. Μια διαφορετική ιδέα είναι η επικοινωνία να γίνεται με το *πέρασμα μηνυμάτων*.

Η *ανταλλαγή πληροφοριών* επιτυγχάνεται με το πέρασμα τιμών από τη διεργασία που στέλνει το μήνυμα (αποστολέας) στη διεργασία που λαμβάνει το μήνυμα (παραλήπτης). Ο *συγχρονισμός* επιτυγχάνεται επειδή η παραλαβή ενός μηνύματος μπορεί να γίνει μόνο αφού έχει γίνει η αποστολή του. Έτσι ο παραλήπτης πρέπει να περιμένει για τη λήψη του μηνύματος έως ότου το αποστείλει ο αποστολέας, τεχνική που συγχρονίζει τις δύο διεργασίες.

Μία μέθοδος επίτευξης *αμοιβαίου αποκλεισμού* με πέρασμα μηνυμάτων αφορά στη χρήση μιας διεργασίας επόπτη η οποία έχει ένα σηματοφορέα με αρχική τιμή 1. Ο επόπτης διαδραματίζει μεσάζοντα ρόλο ανάμεσα στις διεργασίες και στον κοινό πόρο που αντιπροσωπεύει και για κάθε χρήστη δημιουργεί μια υποδιεργασία επικοινωνίας μαζί του. Μια διεργασία που επιθυμεί να χρησιμοποιήσει έναν κοινό πόρο αποστέλλει μήνυμα-αίτηση προς τον επόπτη του πόρου, ο οποίος με τη σειρά του εκτελεί την πράξη p στο σηματοφορέα του και επιστρέφει στη διεργασία ένα μήνυμα ότι έχει την άδεια να χρησιμοποιήσει τον κοινό πόρο.

Για να αποσταλεί ένα μήνυμα πρέπει να εκτελεσθεί μια εντολή της μορφής:

send *λίστα_εκφράσεων* **to** *παραλήπτης*;

όπου η *λίστα_εκφράσεων* περιέχει τις τιμές που πρέπει να σταλούν και ο *παραλήπτης* περιέχει τις αναγνωριστικές πληροφορίες για το ποιος είναι ο παραλήπτης του μηνύματος. Για να ληφθεί το μήνυμα πρέπει να εκτελεσθεί μια εντολή της μορφής:

receive *λίστα_μεταβλητών* **to** *αποστολέας*;

όπου η *λίστα_μεταβλητών* καθορίζει τις μεταβλητές που θα δεχθούν τις τιμές που έχουν σταλεί από τον αποστολέα και ο *αποστολέας* προσδιορίζει την ταυτότητα της διεργασίας αποστολέα. Με την παραλαβή του μηνύματος οι τιμές που αποστάλθηκαν γίνονται τιμές των μεταβλητών της λίστας μεταβλητών και στη συνέχεια το μήνυμα καταστρέφεται. Τα βασικότερα θέματα στο πέρασμα μηνυμάτων είναι ο *καθορισμός του αποστολέα και του παραλήπτη* των μηνυμάτων, καθώς και ο *συγχρονισμός* των διεργασιών τους.

Ένα απλό σχήμα επικοινωνίας αποτελούν τα κανάλια επικοινωνίας. Ο αποστολέας και ο παραλήπτης καθορίζουν μεταξύ τους ένα *κανάλι επικοινωνίας* (communication channel) με

επακριβώς καθορισμένα τα ονόματα του αποστολέα και του παραλήπτη. Έτσι επιτυγχάνεται η μεταξύ τους επικοινωνία με αποστολή και λήψη μηνυμάτων. Το σχήμα αυτό δεν ταιριάζει πάντα σε όλες τις εφαρμογές. Αν, για παράδειγμα, κάποιος παραλήπτης θέλει να πάρει μήνυμα από ένα σύνολο αποστολέων, τότε θα πρέπει να έχει πολλές εντολές receive για κάθε έναν από τους πιθανούς αποστολείς.

Ένα πιο πολύπλοκο σχήμα επικοινωνίας που λύνει το πρόβλημα αυτό είναι η χρησιμοποίηση *γραμματοκιβωτίου* (mailbox). Οι αποστολείς μπορούν να αποστέλλουν μηνύματα σε συγκεκριμένο γραμματοκιβώτιο και οι παραλήπτες λαμβάνουν τα μηνύματά τους από αυτό. Το σχήμα αυτό είναι αρκετά ικανοποιητικό στην απλή περίπτωση όταν υπάρχει ένας μόνο παραλήπτης και πολλοί αποστολείς. Απεναντίας, η γενική περίπτωση αποστολής μηνυμάτων από πολλούς αποστολείς προς πολλούς παραλήπτες είναι ιδιαίτερα δαπανηρή στην υλοποίησή της. Όταν, για παράδειγμα, κάποιο μήνυμα ληφθεί από έναν παραλήπτη τότε πρέπει να ενημερωθούν όλοι οι παραλήπτες που έχουν αντίστοιχη εντολή receive ότι το μήνυμα έχει ληφθεί και δεν εκκρεμεί πια, γεγονός που αυξάνει αρκετά το κόστος της υλοποίησης.

Ο καθορισμός αποστολέα και παραλήπτη μπορεί να είναι είτε στατικός κατά τη μεταγλώττιση του προγράμματος είτε δυναμικός κατά την εκτέλεση του προγράμματος. Στην περίπτωση δυναμικού καθορισμού χρησιμοποιούνται μεταβλητές που έχουν τιμές τα ονόματα των αποστολέων και παραληπτών. Οι τιμές αυτές μπορούν να αλλάζουν κατά το χρόνο εκτέλεσης του προγράμματος με αποτέλεσμα περισσότερες δυνατότητες επικοινωνίας, αλλά ταυτόχρονα με μεγαλύτερο κόστος υλοποίησης.

Ο συγχρονισμός με πέρασμα μηνυμάτων μπορεί να υλοποιηθεί με χρήση απομονωτή (buffer) για κάθε κανάλι επικοινωνίας ή χωρίς. Ο εκάστοτε απομονωτής μπορεί να είναι πεπερασμένης ή όχι χωρητικότητας. Αν δεν υπάρχει απομονωτής τότε η εκτέλεση της εντολής send καθυστερεί τη διεργασία που την εκτέλεσε έως ότου εκτελεσθεί μία αντίστοιχη εντολή receive από κάποια άλλη διεργασία, οπότε και το μήνυμα μεταβιβάζεται και οι διεργασίες συνεχίζουν την εκτέλεσή τους. Το ίδιο συμβαίνει όταν η δεύτερη διεργασία συναντήσει πρώτη την εντολή receive οπότε και αδρανοποιείται μέχρι να αποσταλεί το μήνυμα από την πρώτη διεργασία. Με τον τρόπο αυτό οι δύο διεργασίες συγχρονίζονται και το μήνυμα που μεταβιβάζεται αναφέρεται στην τρέχουσα κατάσταση του αποστολέα τη στιγμή της μεταφοράς. Η επικοινωνία αυτή ονομάζεται *σύγχρονη*.

Στο άλλο άκρο είναι η περίπτωση να υπάρχει απομονωτής και μάλιστα να έχει άπειρη χωρητικότητα. Τότε αν χρονικά εκτελεσθεί πρώτα η εντολή send από τον αποστολέα το μήνυμα μεταφέρεται στον απομονωτή και η διεργασία του αποστολέα συνεχίζει την εκτελεσή της χωρίς καθυστέρηση. Όταν η διεργασία παραλήτης λάβει το μήνυμα τότε ο αποστολέας έχει προχωρήσει την εκτελεσή του, οπότε στην περίπτωση αυτή τη στιγμή της μεταβίβασης το μήνυμα δεν αντιπροσωπεύει την τρέχουσα κατάσταση του αποστολέα. Αυτό το είδος επικοινωνίας χαρακτηρίζεται ως *ασύγχρονη επικοινωνία*. Ανάμεσα στις δύο αυτές περιπτώσεις υπάρχουν ενδιάμεσες καταστάσεις στις οποίες η χωρητικότητα του απομονωτή είναι πεπερασμένη. Στις περιπτώσεις αυτές ανάλογα το μέγεθός του απομονωτή ο αποστολέας έχει προχωρήσει την εκτέλεσή του σε σχέση με τον παραλήπτη, αλλά όχι πάντα αφού όταν γεμίσει ο απομονωτής ο αποστολέας θα καθυστερεί έως ότου ελευθερωθούν θέσεις για την αποστολή του μηνύματος. Από τη μεριά του παραλήπτη αν η εντολή receive προηγηθεί η διεργασία παραλήπτη θα καθυστερεί μέχρι την αποστολή του μηνύματος.

Τις παραπάνω τεχνικές υλοποιούν ένας μεγάλος αριθμός από γλώσσες υψηλού επιπέδου, οι οποίες χρησιμοποιούν το πέρασμα μηνυμάτων για να αξιοποιήσουν τις αρχές του ταυτοχρονισμού. Πολλές από τις γλώσσες αυτές επηρεάστηκαν από τη σημειολογία που έχει προταθεί από τον Hoare. Ο Hoare πρότεινε τη σημειολογία CSP (communicating sequential processes, επικοινωνούσες σειριακές διεργασίες). Σύμφωνα με αυτή τη μέθοδο οι διεργασίες περιγράφονται μέσω μιας δομής του τύπου cobegin και υποστηρίζουν εντολές εισόδου και εξόδου για το πέρασμα των μηνυμάτων. Μία *εντολή εξόδου* έχει τη μορφή:

προορισμός!*έκφραση*

όπου *προορισμός* είναι ένα όνομα διεργασίας και *έκφραση* είναι η τιμή των δεδομένων που θα μεταβιβασθούν. Αντίστοιχα, μια *εντολή εισόδου* έχει τη μορφή:

πηγή?στόχος

όπου *πηγή* είναι το όνομα της διεργασίας και *στόχος* η τοπική μεταβλητή στην οποία θα αποθηκευθούν τα δεδομένα του μηνύματος. Ένα ζευγάρι τέτοιων εντολών δίνει τη δυνατότητα επικοινωνίας μέσω μηνυμάτων αρκεί ο τύπος της έκφρασης που αποστέλλεται και του στόχου στον οποίο πρόκειται να αποθηκευθεί η έκφραση να είναι ο ίδιος.

Όλα τα παραπάνω χαρακτηριστικά που παρουσιάστηκαν συναντώνται σε διάφορες γλώσσες προγραμματισμού που προορίζονται για προγράμματα με στοιχεία ταυτοχρονισμού. Ο προγραμματιστής καλείται να επιλέξει την κατάλληλη περίπτωση για την εκάστοτε εφαρμογή που σχεδιάζει λαμβάνοντας υπόψη την πλατφόρμα στην οποία προγραμματίζει και ανάλογα το βαθμό στον οποίο επιθυμεί να έρθει κοντά στην υλοποίηση να επιλέξει χαμηλότερου ή υψηλότερου επιπέδου γλώσσα προγραμματισμού.

Κεφάλαιο 3

Η γλώσσα CoPCL

Η γλώσσα CoPCL (**Co**ncurrent **PCL**) είναι μια απλή γλώσσα προστακτικού προγραμματισμού με υποστήριξη στοιχείων ταυτοχρονισμού. Η γραμματική της αποτελεί επέκταση της γραμματικής της γλώσσας PCL με επιπλέον στοιχεία αυτά του ταυτοχρονισμού. Η γλώσσα PCL είναι βασισμένη σε ένα γνήσιο υποσύνολο της ISO Pascal, ενώ είναι εμπλουτισμένη με μερικές μικρές παραλλαγές. Τα κύρια χαρακτηριστικά της CoPCL είναι επιγραμματικά τα εξής:

- Σύνταξη παρόμοια σε γενικές γραμμές με αυτή της Pascal.
- Δομημένες διαδικασίες και συναρτήσεις με τους κανόνες εμβέλειας της Pascal.
- Βασικοί τύποι δεδομένων για ακέραιους αριθμούς, χαρακτήρες, λογικές τιμές και πραγματικούς αριθμούς.
- Πίνακες με γνωστό ή άγνωστο μέγεθος και τύποι δεικτών.
- Δομημένος έλεγχος ταυτόχρονης εκτέλεσης διεργασιών και δυνατότητα συγχρονισμού τους μέσω μεταβλητών συνθήκης.
- Βιβλιοθήκη συναρτήσεων.

Περισσότερες λεπτομέρειες της γλώσσας δίνονται στις παραγράφους που ακολουθούν. Λόγω των πολλών ομοιοτήτων της CoPCL με την Pascal, έμφαση δίνεται στα στοιχεία που της προσδίδουν δυνατότητες ταυτοχρονισμού.

3.1 Λεκτικές μονάδες

Οι λεκτικές μονάδες της γλώσσας CoPCL χωρίζονται στις παρακάτω κατηγορίες:

- Τις *λέξεις κλειδιά*, οι οποίες είναι οι παρακάτω:

`and`	`array`	`begin`	`boolean`	`char`	`cobegin`	`coend`
`condition`	`dispose`	`div`	`do`	`else`	`end`	`false`
`forward`	`function`	`goto`	`if`	`integer`	`label`	`mod`
`new`	`nil`	`not`	`of`	`or`	`procedure`	`program`
`real`	`result`	`return`	`signal`	`synchronized`	`then`	`true`
`var`	`wait`	`when`	`while`			

- Τα *ονόματα*, τα οποία αποτελούνται από ένα πεζό ή κεφαλαίο γράμμα του λατινικού αλφαβήτου, πιθανώς ακολουθούμενο από μια σειρά πεζών ή κεφαλαίων γραμμάτων, δεκαδικών ψηφίων ή χαρακτήρων υπογράμμισης (underscore). Τα πεζά γράμματα θεωρούνται διαφορετικά από τα αντίστοιχα κεφαλαία, επομένως τα ονόματα `foo`, `Foo` και `FOO` είναι όλα διαφορετικά. Επίσης, τα ονόματα δεν πρέπει να συμπίπτουν με τις λέξεις κλειδιά που αναφέρθηκαν παραπάνω.

Πίνακας 3.1: Ακολουθίες διαφυγής (escape sequences).

Ακολουθία διαφυγής	Περιγραφή
\n	ο χαρακτήρας αλλαγής γραμμής (line feed)
\t	ο χαρακτήρας στηλοθέτησης (tab)
\r	ο χαρακτήρας επιστροφής στην αρχή της γραμμής (carriage return)
\0	ο χαρακτήρας με ASCII κωδικό 0
\\	ο χαρακτήρας \ (backslash)
\'	ο χαρακτήρας ' (απλό εισαγωγικό)
\"	ο χαρακτήρας " (διπλό εισαγωγικό)

- Οι *ακέραιες σταθερές* χωρίς πρόσημο, που αποτελούνται από ένα ή περισσότερα δεκαδικά ψηφία. Παραδείγματα ακέραιων σταθερών είναι τα ακόλουθα:

```
0       42       1284       00200
```

- Οι *πραγματικές σταθερές* χωρίς πρόσημο, που αποτελούνται από ένα ακέραιο μέρος, ένα κλασματικό μέρος και ένα προαιρετικό εκθετικό μέρος. Το ακέραιο μέρος αποτελείται από ένα ή περισσότερα δεκαδικά ψηφία. Το κλασματικό μέρος αποτελείται από το χαρακτήρα . της υποδιαστολής, ακολουθούμενο από ένα ή περισσότερα δεκαδικά ψηφία. Τέλος, το εκθετικό μέρος αποτελείται από το πεζό ή κεφαλαίο γράμμα E, ένα προαιρετικό πρόσημο + ή - και ένα ή περισσότερα δεκαδικά ψηφία. Παραδείγματα πραγματικών σταθερών είναι τα ακόλουθα:

```
42.0       4.2e1       0.420e+2       42000.0e-3
```

- Οι *σταθεροί χαρακτήρες*, που αποτελούνται από ένα χαρακτήρα μέσα σε απλά εισαγωγικά. Ο χαρακτήρας αυτός μπορεί να είναι οποιοσδήποτε κοινός χαρακτήρας ή *ακολουθία διαφυγής* (escape sequence). Κοινοί χαρακτήρες είναι όλοι οι εκτυπώσιμοι χαρακτήρες πλην των απλών και διπλών εισαγωγικών και του χαρακτήρα \ (backslash). Οι ακολουθίες διαφυγής ξεκινούν με το χαρακτήρα \ (backslash) και περιγράφονται στον πίνακα 3.1. Παραδείγματα σταθερών χαρακτήρων είναι οι ακόλουθες:

```
'a'       '1'       '\n'       '\''       '\x1d'
```

- Οι *σταθερές συμβολοσειρές*, που αποτελούνται από μια ακολουθία κοινών χαρακτήρων ή ακολουθιών διαφυγής μέσα σε διπλά εισαγωγικά. Οι συμβολοσειρές δεν μπορούν να εκτείνονται σε περισσότερες από μία γραμμές προγράμματος. Παραδείγματα σταθερών συμβολοσειρών είναι οι ακόλουθες:

```
"abc"          "Route 66"          "Hello world!\n"
     "Name:\t\"Douglas Adams\"\nValue:\t42\n"
```

- Τους *συμβολικούς τελεστές*, οι οποίοι είναι οι παρακάτω:

```
=    >    <    <>    >=    <=    +    -    *    /    ^    @
```

- Τους *διαχωριστές*, οι οποίοι είναι οι παρακάτω:

```
:=      ;      ||      .      (      )      :      ,      [      ]
```

Εκτός από τις λεκτικές μονάδες που προαναφέρθηκαν, ένα πρόγραμμα CoPCL μπορεί επίσης να περιέχει τα παρακάτω, τα οποία διαχωρίζουν λεκτικές μονάδες και αγνοούνται:

- *Κενούς χαρακτήρες*, δηλαδή ακολουθίες αποτελούμενες από κενά διαστήματα (space), χαρακτήρες στηλοθέτησης (tab), χαρακτήρες αλλαγής γραμμής (line feed) ή χαρακτήρες επιστροφής στην αρχή της γραμμής (carriage return).

- *Σχόλια*, τα οποία αρχίζουν με την ακολουθία χαρακτήρων (* και τερματίζονται με την πρώτη μετέπειτα εμφάνιση της ακολουθίας χαρακτήρων *). Κατά συνέπεια, τα σχόλια δεν επιτρέπεται να είναι φωλιασμένα. Στο εσωτερικό τους επιτρέπεται η εμφάνιση οποιουδήποτε χαρακτήρα.

3.2 Τύποι δεδομένων

Η CoPCL υποστηρίζει τέσσερις βασικούς τύπους δεδομένων:

- `integer` : ακέραιοι αριθμοί,

- `boolean` : λογικές τιμές,

- `char` : χαρακτήρες,

- `real` : πραγματικοί αριθμοί, και

- `condition` : συνθήκες συγχρονισμού.

Οι τύποι `integer` και `real` ονομάζονται *αριθμητικοί* τύποι. Οι βασικοί τύποι, οι τύποι πινάκων δεδομένου μεγέθους και οι τύποι δεικτών ονομάζονται *πλήρεις* τύποι. Αντίθετα, οι τύποι πινάκων άγνωστου μεγέθους ονομάζονται *ημιτελείς* τύποι. Κατά το σχηματισμό ενός τύπου πίνακα, είτε πλήρους είτε ημιτελούς, ο τύπος του στοιχείου t θα πρέπει να είναι πλήρης. Η αρίθμηση των στοιχείων ενός πίνακα ακολουθεί τη σύμβαση της C: το πρώτο στοιχείο έχει δείκτη 0, το δεύτερο 1, ενώ το τελευταίο έχει δείκτη κατά ένα μικρότερο από την πραγματική διάσταση του πίνακα.

Ο αριθμός των bytes που καταλαμβάνουν τα δεδομένα κάθε τύπου στη μνήμη του υπολογιστή, καθώς και ο ακριβής τρόπος παράστασης αυτών εξαρτώνται από την υλοποίηση της CoPCL. Στο πλαίσιο της εργασίας αυτής, κάναμε τις ακόλουθες παραδοχές για την κατασκευή του μεταγλωττιστή της CoPCL για υπολογιστές βασισμένους στους 32-bit επεξεργαστές της Intel, χρησιμοποιώντας το μοντέλο προγραμμάτων ELF.

- `integer` : μέγεθος 4 bytes, παράσταση συμπληρώματος ως προς 2.

- `boolean` : μέγεθος 1 byte, τιμές `false` $= 0$ και `true` $= 1$.

- `char` : μέγεθος 1 byte, παράσταση σύμφωνα με τον πίνακα ASCII.

- `real` : μέγεθος 10 bytes, παράσταση σύμφωνα με το πρότυπο IEEE 754.

- `condition` : μέγεθος 72 bytes, παράσταση ενός mutex των 24 bytes και μίας μεταβλητής συνθήκης των 48 bytes.

- `array [` n `] of` t : μέγεθος ίσο με n επί το μέγεθος του τύπου t, τοποθέτηση σε αύξουσα σειρά διευθύνσεων.

- `array of` t : μέγεθος 4 bytes, παράσταση με δείκτη στο πρώτο στοιχείο του πίνακα.

- `^`t : μέγεθος 4 bytes.

3.3 Δομή του προγράμματος

Ένα πρόγραμμα CoPCL αποτελείται από την επικεφαλίδα και το σώμα της κύριας δομικής μονάδας. Η επικεφαλίδα αυτής είναι της μορφής:

```
program p;
```

όπου `p` το όνομα του προγράμματος. Το σώμα κάθε δομικής μονάδας μπορεί να περιέχει προαιρετικά:

- Δηλώσεις μεταβλητών,
- Ορισμούς υποπρογραμμάτων.
- Δηλώσεις υποπρογραμμάτων, οι ορισμοί των οποίων θα ακολουθήσουν.

Τελευταίο συστατικό του σώματος μιας δομικής μονάδας είναι μια σύνθετη εντολή, η οποία καθορίζει τη λειτουργία της δομικής μονάδας. Στην περίπτωση της κυρίας δομικής μονάδας, η εκτέλεση του προγράμματος ξεκινά από αυτή τη σύνθετη εντολή.

Η CoPCL ακολουθεί τους κανόνες εμβέλειας της ISO Pascal, όσον αφορά στην ορατότητα των ονομάτων μεταβλητών, υποπρογραμμάτων και παραμέτρων.

3.3.1 Μεταβλητές

Οι δηλώσεις μεταβλητών γίνονται με τη λέξη κλειδί `var`. Ακολουθούν ένα ή περισσότερα ονόματα μεταβλητών και ένας τύπος δεδομένων, ο οποίος πρέπει να είναι πλήρης. Περισσότερες συνεχόμενες δηλώσεις μεταβλητών μπορούν να γίνουν παραλείποντας τη λέξη κλειδί `var`. Παραδείγματα δηλώσεων είναι:

```
var i    : integer;
    x, y : real;
var s    : array [80] of char;
    cnd  : condition;
```

3.3.2 Υποπρογράμματα

Τα υποπρογράμματα διακρίνονται σε *διαδικασίες* (procedures) και *συναρτήσεις* (functions). Κάθε υποπρόγραμμα είναι μια δομική μονάδα και αποτελείται από την επικεφαλίδα του και το σώμα του. Η δομή του σώματος έχει ήδη περιγραφεί. Στην επικεφαλίδα αναφέρεται κατ' αρχήν αν το υποπρόγραμμα είναι διαδικασία ή συνάρτηση, το όνομά του, οι τυπικές του παράμετροι μέσα σε παρενθέσεις και, αν πρόκειται για συνάρτηση, ο τύπος του αποτελέσματος, ο οποίος πρέπει να είναι πλήρης. Οι παρενθέσεις είναι υποχρεωτικές ακόμα και αν ένα υποπρόγραμμα δεν έχει τυπικές παραμέτρους.

Κάθε τυπική παράμετρος χαρακτηρίζεται από το όνομά της, τον τύπο της και τον τρόπο περάσματος. Η CoPCL υποστηρίζει πέρασμα παραμέτρων κατ' αξία (by value) και κατ' αναφορά (by reference). Αν στη δήλωση μιας τυπικής παραμέτρου έχει προηγηθεί η λέξη κλειδί `var`, τότε αυτή περνά κατ' αναφορά, διαφορετικά περνά κατ' αξία. Οι τύποι των τυπικών παραμέτρων που περνούν κατ' αξία πρέπει να είναι πλήρεις.

Ακολουθούν παραδείγματα επικεφαλίδων υποπρογραμμάτων.

```
procedure p1 ();
procedure p2 (n : integer);
procedure p3 (a, b : integer; var b : boolean);
function  f1 (x : real) : real;
function  f2 (var s : array of char) : integer;
function  f3 (x : real) : array [10] of real;
function  f4 (n : integer; x : real) : ^array of real;
```

Στην περίπτωση αμοιβαία αναδρομικών υποπρογραμμάτων, το όνομα ενός υποπρογράμματος χρειάζεται να εμφανιστεί πριν τον ορισμό του. Στην περίπτωση αυτή, για να μην παραβιαστούν οι κανόνες εμβέλειας, πρέπει να έχει προηγηθεί μια δήλωση της επικεφαλίδας αυτού του υποπρογράμματος, χωρίς το σώμα του. Αυτό γίνεται με τη λέξη κλειδί `forward`.

3.4 Εκφράσεις

Κάθε έκφραση της CoPCL διαθέτει ένα μοναδικό τύπο[1] και μπορεί να αποτιμηθεί δίνοντας ως αποτέλεσμα μια τιμή αυτού του τύπου. Οι εκφράσεις διακρίνονται σε δύο κατηγορίες: αυτές που προκύπτουν από l-values, οι οποίες περιγράφονται στην ενότητα 3.4.1, και αυτές που προκύπτουν από r-values, που περιγράφονται στις ενότητες 3.4.2 ως 3.4.4. Τα δυο αυτά είδη τιμών έχουν πάρει το όνομά τους από τη θέση τους σε μια εντολή ανάθεσης: οι l-values εμφανίζονται στο αριστερό μέλος της ανάθεσης ενώ οι r-values στο δεξί.

Τόσο οι l-values όσο και οι r-values μπορούν να εμφανίζονται μέσα σε παρενθέσεις, που χρησιμοποιούνται για λόγους ομαδοποίησης.

3.4.1 L-values

Οι l-values αντιπροσωπεύουν αντικείμενα που καταλαμβάνουν χώρο στη μνήμη του υπολογιστή κατά την εκτέλεση του προγράμματος και τα οποία μπορούν να περιέχουν τιμές. Τέτοια αντικείμενα είναι οι μεταβλητές, οι παράμετροι των υποπρογραμμάτων, οι μεταβλητές που κατασκευάζονται με δυναμική παραχώρηση μνήμης και οι θέσεις όπου φυλλάσσονται τα αποτελέσματα των συναρτήσεων. Συγκεκριμένα:

- Το όνομα μιας μεταβλητής ή μιας παραμέτρου υποπρογράμματος είναι l-value και αντιστοιχεί στο εν λόγω αντικείμενο. Ο τύπος της l-value είναι ο τύπος του αντίστοιχου αντικειμένου.

- Οι σταθερές συμβολοσειρές, όπως περιγράφονται στην ενότητα 3.1 είναι l-values. Έχουν τύπο `array [`n`] of char`, όπου n είναι ο αριθμός χαρακτήρων που περιέχονται στη συμβολοσειρά προσαυξημένος κατά ένα. Κάθε τέτοια l-value αντιστοιχεί σε ένα αντικείμενο τύπου πίνακα, όπου βρίσκονται αποθηκευμένοι με τη σειρά οι χαρακτήρες της συμβολοσειράς. Στο τέλος του πίνακα αποθηκεύεται αυτόματα ο χαρακτήρας `'\0'`, σύμφωνα με τη σύμβαση που ακολουθεί η γλώσσα C για τις συμβολοσειρές. Οι σταθερές συμβολοσειρές είναι το μόνο είδος σταθεράς τύπου πίνακα που επιτρέπεται στην CoPCL.

- Αν l είναι μια l-value τύπου `array [`n`] of` t ή τύπου `array of` t, και e είναι μια έκφραση τύπου `integer`, τότε l`[`e`]` είναι μια l-value με τύπο t. Αν η τιμή της έκφρασης e είναι ο μη αρνητικός ακέραιος n τότε αυτή η l-value αντιστοιχεί στο στοιχείο με δείκτη n του πίνακα που αντιστοιχεί στην l. Η τιμή του n δεν πρέπει να υπερβαίνει τα πραγματικά όρια του πίνακα.

- Αν e είναι μια έκφραση τύπου `^`t, τότε e`^` είναι μια l-value με τύπο t, που αντιστοιχεί στο αντικείμενο όπου δείχνει η τιμή της e.

- Στο σώμα μιας συνάρτησης που επιστρέφει αποτέλεσμα τύπου t, η λέξη κλειδί `result` είναι μια l-value με τύπο t, που αντιστοιχεί στο αντικείμενο όπου φυλλάσσεται το αποτέλεσμα της συνάρτησης. Το αντικείμενο αυτό είναι προσωρινό και δεν πρέπει να χρησιμοποιείται μετά την επιστροφή από τη συνάρτηση.

Αν μια l-value χρησιμοποιηθεί ως έκφραση, η τιμή αυτής της έκφρασης είναι ίση με την τιμή η οποία περιέχεται στο αντικείμενο που αντιστοιχεί στην l-value.

[1] Εξαίρεση αποτελεί η σταθερά μηδενικού δείκτη `nil`, η οποία δεν έχει μοναδικό τύπο. Περιγράφεται στην ενότητα 3.4.2.

3.4.2 Σταθερές

Στις r-values της γλώσσας CoPCL συγκαταλέγονται οι ακόλουθες σταθερές:

- Οι ακέραιες σταθερές χωρίς πρόσημο, όπως περιγράφονται στην ενότητα 3.1. Έχουν τύπο `integer` και η τιμή τους είναι ίση με τον μη αρνητικό ακέραιο αριθμό που παριστάνουν.
- Οι λογικές σταθερές `true` και `false`, με τύπο `boolean` και προφανείς τιμές.
- Οι πραγματικές σταθερές χωρίς πρόσημο, όπως περιγράφονται στην ενότητα 3.1. Έχουν τύπο `real` και η τιμή τους είναι ίση με τον μη αρνητικό πραγματικό αριθμό που παριστάνουν.
- Οι σταθεροί χαρακτήρες, όπως περιγράφονται στην ενότητα 3.1. Έχουν τύπο `char` και η τιμή τους είναι ίση με το χαρακτήρα που παριστάνουν.
- Η λέξη κλειδί `nil` που έχει τύπο $\hat{}t$, για κάθε έγκυρο τύπο t, και η τιμή της οποίας είναι ο μηδενικός δείκτης. Είναι η μόνη έκφραση της CoPCL που δεν έχει μοναδικό τύπο. Ο μηδενικός δείκτης απαγορεύεται να αποδεικτοδοτηθεί με χρήση του τελεστή `^`.

3.4.3 Τελεστές

Οι τελεστές της CoPCL διακρίνονται σε τελεστές με ένα τελούμενο και τελεστές με δύο τελούμενα. Από τους πρώτους, ορισμένοι γράφονται πρίν το τελούμενο (prefix) και ορισμένοι μετά (postfix), ενώ οι δεύτεροι γράφονται πάντα μεταξύ των τελουμένων (infix). Η αποτίμηση των τελουμένων των τελεστών με δυο τελούμενα γίνεται από αριστερά προς τα δεξιά.

Ήδη στην ενότητα 3.4.1 περιγράφηκε ο τελεστής αναφοράς σε στοιχείο πίνακα, η σύνταξη του οποίου αποτελεί εξαίρεση στα παραπάνω, και ο τελεστής αποδεικτοδότησης `^`. Οι δύο αυτοί τελεστές είναι οι μοναδικοί που έχουν ως αποτέλεσμα l-value. Στη συνέχεια περιγράφονται οι υπόλοιποι τελεστές της CoPCL, που έχουν ως αποτέλεσμα r-value.

- Ο τελεστής `@` επιστρέφει τη διεύθυνση ενός αντικειμένου. Αν l είναι μια l-value τύπου t, τότε `@`l είναι μια r-value τύπου $\hat{}t$. Η τιμή της είναι η διεύθυνση του αντικειμένου που αντιστοιχεί στην l και είναι πάντα διαφορετική του μηδενικού δείκτη.
- Οι τελεστές με ένα τελούμενο `+` και `-` υλοποιούν τους τελεστές προσήμου. Το τελούμενο πρέπει να είναι αριθμητικού τύπου και το αποτέλεσμα είναι του ίδιου τύπου με το τελούμενο.
- Ο τελεστής `not` υλοποιεί τη λογική άρνηση. Το τελούμενό του πρέπει να είναι τύπου `boolean`, και τον ίδιο τύπο έχει και το αποτέλεσμα.
- Οι τελεστές με δύο τελούμενα `+`, `-`, `*`, `/`, `div` και `mod` υλοποιούν αντίστοιχα τις αριθμητικές πράξεις της πρόσθεσης, της αφαίρεσης, του πολλαπλασιασμού, της πραγματικής διαίρεσης, του πηλίκου και του υπόλοιπου της ακέραιας διαίρεσης. Τα τελούμενά τους πρέπει να είναι εκφράσεις αριθμητικών τύπων. Στην περίπτωση των τελεστών `+`, `-` και `*`, αν και τα δύο τελούμενα είναι τύπου `integer` τότε και το αποτέλεσμα είναι τύπου `integer`, διαφορετικά το αποτέλεσμα είναι τύπου `real`. Στην περίπτωση του τελεστή `/`, το αποτέλεσμα είναι πάντα τύπου `real`. Τέλος, στην περίπτωση των τελεστών `div` και `mod`, τα τελούμενα πρέπει να είναι τύπου `integer` και το αποτέλεσμα είναι επίσης τύπου `integer`.
- Οι τελεστές `=` και `<>` υλοποιούν αντίστοιχα την ισότητα και την ανισότητα. Το αποτέλεσμα είναι τύπου `boolean`. Τα τελούμενα πρέπει να είναι είτε και τα δύο αριθμητικά, οπότε συγκρίνονται οι αριθμητικές τιμές τους, είτε του ίδιου τύπου, ο οποίος δεν πρέπει να είναι τύπος πίνακα. Στη δεύτερη περίπτωση, συγκρίνονται οι δυαδικές αναπαραστάσεις των τιμών των τελουμένων.

Πίνακας 3.2: Προτεραιότητα και προσεταιριστικότητα των τελεστών της CoPCL.

Τελεστές	Περιγραφή	Αριθμός τελουμένων	Θέση και προσεταιριστικότητα
`[ ]`	Αναφορά σε στοιχείο πίνακα	2	ειδική
`@`	Διεύθυνση αντικειμένου	1	prefix
`^`	Αποδεικτοδότηση	1	postfix
`+ -`	Πρόσημα	1	prefix
`not`	Λογική άρνηση	1	prefix
`* / div mod and`	Πολλαπλασιαστικοί τελεστές	2	infix, αριστερή
`+ - or`	Προσθετικοί τελεστές	2	infix, αριστερή
`= > < <= >= <>`	Σχεσιακοί τελεστές	2	infix, καμία

- Οι τελεστές `<`, `>`, `<=` και `>=` υλοποιούν τις σχέσεις ανισότητας μεταξύ αριθμών. Τα τελούμενα πρέπει να είναι και τα δύο αριθμητικά και το αποτέλεσμα είναι τύπου `boolean`.

- Οι τελεστές `and` και `or` υλοποιούν αντίστοιχα τις πράξεις της λογικής σύζευξης και διάζευξης. Τα τελούμενα πρέπει να είναι τύπου `boolean` και τον ίδιο τύπο έχει και το αποτέλεσμα. Η αποτίμηση εκφράσεων που χρησιμοποιούν αυτούς τους τελεστές γίνεται με *βραχυκύκλωση* (short-circuit). Δηλαδή, αν το αποτέλεσμα της έκφρασης είναι γνωστό από την αποτίμηση και μόνο του πρώτου τελούμενου, το δεύτερο τελούμενο δεν αποτιμάται καθόλου.

Στον πίνακα 3.2 ορίζεται η προτεραιότητα και η προσεταιριστικότητα των τελεστών της CoPCL.

3.4.4 Κλήση συναρτήσεων

Αν f είναι το όνομα μιας συνάρτησης με αποτέλεσμα τύπου t, τότε η έκφραση $f(e_1, \ldots, e_n)$ είναι μια r-value με τύπο t. Ο αριθμός των πραγματικών παραμέτρων n πρέπει να συμπίπτει με τον αριθμό των τυπικών παραμέτρων της f. Επίσης, ο τύπος και το είδος κάθε πραγματικής παραμέτρου πρέπει να συμπίπτει με τον τύπο και τον τρόπο περάσματος της αντίστοιχης τυπικής παραμέτρου, σύμφωνα με τους παρακάτω κανόνες. Η έννοια της *συμβατότητας για ανάθεση* περιγράφεται στην ενότητα 3.5.

- Αν η τυπική παράμετρος είναι τύπου t και περνά κατ' αξία, τότε ο τύπος t' της αντίστοιχης πραγματικής παραμέτρου πρέπει να είναι συμβατός για ανάθεση με τον τύπο t.

- Αν η τυπική παράμετρος είναι τύπου t και περνά κατ' αναφορά, τότε η αντίστοιχη πραγματική παράμετρος πρέπει να είναι l-value τύπου t', όπου ο τύπος $\hat{}t'$ πρέπει να είναι συμβατός για ανάθεση με τον τύπο $\hat{}t$.

Κατά την κλήση μιας συνάρτησης, οι πραγματικές παράμετροι αποτιμώνται από αριστερά προς τα δεξιά.

3.5 Εντολές

Οι εντολές που υποστηρίζει η γλώσσα CoPCL είναι οι ακόλουθες:

- Η κενή εντολή, που δεν κάνει καμία ενέργεια.

- Η εντολή ανάθεσης l := e, όπου l μια l-value τύπου t και e μια έκφραση τύπου t'. Ο τύπος t' πρέπει να είναι *συμβατός για ανάθεση* με τον τύπο t. Αυτή η σχέση συμβατότητας, που πρέπει να τονιστεί ότι δεν είναι συμμετρική, ορίζεται ως εξής:
  - Κάθε πλήρης τύπος είναι συμβατός για ανάθεση με τον εαυτό του.
  - Ο τύπος `integer` είναι συμβατός για ανάθεση με τον τύπο `real`.
  - Ο τύπος `^array [`n`] of` t είναι συμβατός για ανάθεση με τον τύπο `^array of` t.
- Η σύνθετη εντολή, που αποτελείται από μια σειρά έγκυρων εντολών χωρισμένων με το διαχωριστή `;`, ανάμεσα στις λέξεις κλειδιά `begin` και `end`. Οι εντολές αυτές εκτελούνται διαδοχικά, εκτός αν κάποια από αυτές είναι εντολή άλματος.
- Η σύνθετη εντολή ελέγχου ταυτόχρονων διεργασιών, που αποτελείται από μια σειρά έγκυρων εντολών χωρισμένων με το διαχωριστή ταυτοχρονισμού `||`, ανάμεσα στις λέξεις κλειδιά `cobegin` και `coend`. Οι εντολές αυτές δρομολογούνται για ταυτόχρονη εκτέλεση σε ξεχωριστές διεργασίες.
- Η εντολή ελέγχου `if` e `then` s_1 `else` s_2. Η έκφραση e πρέπει να έχει τύπο `boolean` και τα s_1, s_2 να είναι έγκυρες εντολές. Το τμήμα `else` είναι προαιρετικό.
- Η εντολή βρόχου `while` e `do` s. Η έκφραση e πρέπει να έχει τύπο `boolean` και το s να είναι έγκυρη εντολή.
- Η εντολή με ετικέτα I : s, όπου I το όνομα μιας ετικέτας και s μια έγκυρη εντολή. Κάθε ετικέτα πρέπει να δηλώνεται στο τμήμα δηλώσεων της δομικής μονάδας και να ορίζεται το πολύ μια φορά στη σύνθετη εντολή που ορίζει το σώμα αυτής.
- Η εντολή άλματος `goto` I, όπου I το όνομα μιας ετικέτας που πρέπει να εμφανίζεται στην ίδια δομική μονάδα. Πέραν αυτού, δεν υπάρχουν άλλοι περιορισμοί ως προς τη θέση των εντολών αλμάτων ή των ετικετών όπου αυτά οδηγούν.
- Η εντολή `return`, που επιστρέφει τερματίζοντας την εκτέλεση της δομικής μονάδας.
- Η κλήση μιας διαδικασίας. Συντακτικά και σημασιολογικά συμπίπτει με την κλήση μιας συνάρτησης, με τη διαφορά ότι δεν επιστρέφεται αποτέλεσμα.
- Η εντολή `new` με την οποία γίνεται δυναμική παραχώρηση μνήμης, και η οποία εμφανίζεται σε δυο μορφές:
  - `new` l, όπου l πρέπει να είναι μια l-value τύπου `^`t και t πρέπει να είναι πλήρης τύπος. Μετά την εκτέλεση της εντολής, ο δείκτης που αντιστοιχεί στην l τοποθετείται να δείχνει προς ένα νέο δυναμικό αντικείμενο τύπου t.
  - `new [`e`]` l, όπου l πρέπει να είναι μια l-value τύπου `^array of` t και e μια έκφραση τύπου `integer`. Η τιμή της e πρέπει να είναι ένας θετικός αριθμός n. Μετά την εκτέλεση της εντολής, ο δείκτης που αντιστοιχεί στην l τοποθετείται να δείχνει προς ένα νέο δυναμικό αντικείμενο τύπου `array [`n`] of` t.
- Η εντολή `dispose` με την οποία γίνεται η αποδέσμευση της μνήμης που έχει παραχωρηθεί δυναμικά με την εντολή `new`. Εμφανίζεται και αυτή σε δυο μορφές:
  - `dispose` l, όπου l πρέπει να είναι μια l-value τύπου `^`t και t πρέπει να είναι πλήρης τύπος. Η τρέχουσα τιμή της l πρέπει να είναι ένας δείκτης προς ένα δυναμικό αντικείμενο που έχει κατασκευαστεί με χρήση της εντολής `new`. Μετά την εκτέλεση της εντολής, η l περιέχει το μηδενικό δείκτη.

  - `dispose []` l, όπου l πρέπει να είναι μια l-value τύπου `^array of` t. Η τρέχουσα τιμή της l πρέπει να είναι ένας δείκτης προς ένα δυναμικό αντικείμενο που έχει κατασκευαστεί με χρήση της εντολής `new`. Μετά την εκτέλεση της εντολής, η l περιέχει το μηδενικό δείκτη.

- Η εντολή `synchronized` με την οποία γίνεται ο συγχρονισμός δύο διεργασιών με αναφορά σε μία μεταβλητή συνθήκης. Εμφανίζεται σε δύο μορφές:

  - `synchronized` i `do` s, όπου i πρέπει να είναι το όνομα μιας μεταβλητής τύπου `condition` και το s να είναι έγκυρη εντολή. Με την εντολή αυτή εμποδίζονται δύο διεργασίες να εκτελούν κώδικα συγχρονισμένο πάνω στην ίδια μεταβλητή i.
  - `synchronized` i `when` e `do` s, όπου i πρέπει να είναι το όνομα μιας μεταβλητής τύπου `condition`, το e μία έκφραση που έχει τύπο `boolean` και το s έγκυρη εντολή. Όπως και πριν, με την εντολή αυτή εμποδίζονται δύο διεργασίες να εκτελούν κώδικα συγχρονισμένο πάνω στην ίδια μεταβλητή i με τη διαφορά ότι σε αυτήν την περίπτωση η δέσμευση της μεταβλητής `condition` γίνεται μόνο αν ισχύει η λογική συνθήκη e. Σε αντίθετη περίπτωση η διεργασία αδρανοποιείται μέχρι να ειδοποιηθεί (μέσω της εντολής `signal`) από κάποια άλλη διεργασία, ώστε να επαναλάβει τον έλεγχο και αν ισχύει να προχωρήσει στην εκτέλεση της s.

- Η εντολή `wait` i, όπου i πρέπει να είναι το όνομα μιας μεταβλητής τύπου `condition`. Με την εντολή αυτή αδρανοποιείται η διεργασία που την εκτελεί και σε περίπτωση που ο κώδικάς της είναι συγχρονισμένος πάνω στην μεταβλητή i η διεργασία απελευθερώνει τη μεταβλητή συνθήκης, ώστε να επιτραπεί σε κάποια άλλη διεργασία να εκτελέσει συγχρονισμένο κώδικα πάνω σε αυτήν.

- Η εντολή `signal` i, όπου i πρέπει να είναι το όνομα μιας μεταβλητής τύπου `condition`. Με την εντολή αυτή ειδοποιείται κάποια διεργασία που πιθανόν έχει αδρανοποιηθεί είτε εκτελώντας την εντολή `wait` i, είτε αναμένοντας στη δομή `synchronized` i `when` e `do` s να εκτελέσει την εντολή s συγχρονιμένη πάνω στη μεταβλητή i, όταν επαληθευθεί η λογική συνθήκη e. Αν περισσότερες από μία διεργασίες αναμένουν ειδοποίηση σε αναφορά με τη μεταβλητή i, τότε ειδοποιείται εκείνη που εισήλθε πρώτη στην ουρά αναμονής της i (FIFO).

3.6 Προκαθορισμένα υποπρογράμματα

Η CoPCL υποστηρίζει ένα σύνολο προκαθορισμένων υποπρογραμμάτων, τα οποία βρίσκονται στη διάθεση του προγραμματιστή. Τα υποπρογράμματα αυτά είναι ορατά σε κάθε δομική μονάδα, εκτός αν επισκιάζονται από μεταβλητές, παραμέτρους ή υποπρογράμματα με το ίδιο όνομα. Παρακάτω δίνονται οι επικεφαλίδες τους, όπως θα γράφονταν αν τα ορίζαμε σε ένα πρόγραμμα CoPCL. Επίσης, εξηγείται η λειτουργία τους.

3.6.1 Είσοδος και έξοδος

```
procedure writeInteger (n : integer);
procedure writeBoolean (b : boolean);
procedure writeChar    (c : char);
procedure writeReal    (r : real);
procedure writeString  (var s : array of char);
```

Οι διαδικασίες αυτές χρησιμοποιούνται για την εκτύπωση τιμών που ανήκουν στους βασικούς τύπους της CoPCL, καθώς και για την εκτύπωση συμβολοσειρών.

```
function  readInteger () : integer;
function  readBoolean () : boolean;
function  readChar    () : char;
function  readReal    () : real;
procedure readString  (size : integer; var s : array of char);
```

Αντίστοιχα, τα παραπάνω υποπρογράμματα χρησιμοποιούνται για την εισαγωγή τιμών που ανήκουν στους βασικούς τύπους της CoPCL και για την εισαγωγή συμβολοσειρών. Η διαδικασία **readString** χρησιμοποιείται για την ανάγνωση μιας συμβολοσειράς μέχρι τον επόμενο χαρακτήρα αλλαγής γραμμής. Οι παράμετροί της καθορίζουν το μέγιστο αριθμό χαρακτήρων (συμπεριλαμβανομένου του τελικού '\0') που επιτρέπεται να διαβαστούν και τον πίνακα χαρακτήρων στον οποίο αυτοί θα τοποθετηθούν. Ο χαρακτήρας αλλαγής γραμμής δεν αποθηκεύεται. Αν το μέγεθος του πίνακα εξαντληθεί πριν συναντηθεί χαρακτήρας αλλαγής γραμμής, η ανάγνωση θα συνεχιστεί αργότερα από το σημείο όπου διακόπηκε.

3.6.2 Μαθηματικές συναρτήσεις

```
function abs  (n : integer) : integer;
function fabs (r : real)    : real;
```

Η απόλυτη τιμή ενός ακέραιου ή πραγματικού αριθμού.

```
function sqrt   (r : real) : real;
function sin    (r : real) : real;
function cos    (r : real) : real;
function tan    (r : real) : real;
function arctan (r : real) : real;
function exp    (r : real) : real;
function ln     (r : real) : real;
function pi     ()         : real;
```

Βασικές μαθηματικές συναρτήσεις: τετραγωνική ρίζα, τριγωνομετρικές συναρτήσεις, εκθετική συνάρτηση, φυσικός λογάριθμος, ο αριθμός π.

3.6.3 Συναρτήσεις μετατροπής

```
function trunc (r : real) : integer;
function round (r : real) : integer;
```

Η **trunc** επιστρέφει τον πλησιέστερο ακέραιο αριθμό, η απόλυτη τιμή του οποίου είναι μικρότερη από την απόλυτη τιμή του **r**. Η **round** επιστρέφει τον πλησιέστερο ακέραιο αριθμό. Σε περίπτωση αμφιβολίας, προτιμάται ο αριθμός με τη μεγαλύτερη απόλυτη τιμή.

```
function ord (c : char)    : integer;
function chr (n : integer) : char;
```

Μετατρέπουν από ένα χαρακτήρα στον αντίστοιχο κωδικό ASCII και αντίστροφα.

3.6.4 Συναρτήσεις ταυτοχρονισμού

```
procedure sleep (n : integer);
```

Η **sleep** προκαλεί την αδρανοποίηση της διεργασίας που την καλεί για χρονικό διάστημα ίσο με **n** δευτερόλεπτα.

3.7 Πλήρης γραμματική της CoPCL

Η σύνταξη της γλώσσας CoPCL δίνεται παρακάτω σε μορφή EBNF. Η γραμματική που ακολουθεί είναι *διφορούμενη*, οι αμφισημίες όμως μπορούν να ξεπερασθούν αν λάβει κανείς υπόψη τους κανόνες προτεραιότητας και προσεταιριστικότητας των τελεστών, όπως περιγράφονται στην ενότητα 3.4.3. Τα σύμβολα *id*, *integer-const*, *real-const*, *char-const* και *string-literal* είναι τερματικά σύμβολα της γραμματικής.

⟨program⟩ ::= “**program**” ⟨id⟩ “;” ⟨body⟩ “.”

⟨body⟩ ::= (⟨local⟩)* ⟨block⟩

⟨local⟩ ::= “**var**” (⟨id⟩ (“,” ⟨id⟩)* “:” ⟨type⟩ “;”)+
| “**label**” ⟨id⟩ (“,” ⟨id⟩)* “;”
| ⟨header⟩ “;” ⟨body⟩ “;”
| “**forward**” ⟨header⟩ “;”

⟨header⟩ ::= “**procedure**” ⟨id⟩ “(” [⟨formal⟩ (“;” ⟨formal⟩)*] “)”
| “**function**” ⟨id⟩ “(” [⟨formal⟩ (“;” ⟨formal⟩)*] “)” “:” ⟨type⟩

⟨formal⟩ ::= [“**var**”] ⟨id⟩ (“,” ⟨id⟩)* “:” ⟨type⟩

⟨type⟩ ::= “**integer**” | “**real**” | “**boolean**” | “**char**” | “**condition**”
| “**array**” [“[” ⟨integer-const⟩ “]”] “**of**” ⟨type⟩ | “^” ⟨type⟩

⟨block⟩ ::= “**begin**” ⟨stmt⟩ (“;” ⟨stmt⟩)* “**end**”

⟨coblock⟩ ::= “**cobegin**” ⟨stmt⟩ (“||” ⟨stmt⟩)* “**coend**”

⟨stmt⟩ ::= ϵ | ⟨l-value⟩ “:=” ⟨expr⟩ | ⟨block⟩ | ⟨coblock⟩ | ⟨call⟩
| “**if**” ⟨expr⟩ “**then**” ⟨stmt⟩ [“**else**” ⟨stmt⟩]
| “**while**” ⟨expr⟩ “**do**” ⟨stmt⟩
| ⟨id⟩ “:” ⟨stmt⟩ | “**goto**” ⟨id⟩ | “**return**”
| “**new**” [“[” ⟨expr⟩ “]”] ⟨l-value⟩
| “**dispose**” [“[” “]”] ⟨l-value⟩
| “**synchronized**” ⟨id⟩ [“**when**” ⟨expr⟩] “**do**” ⟨stmt⟩
| “**wait**” ⟨id⟩ | “**signal**” ⟨id⟩

⟨expr⟩ ::= ⟨l-value⟩ | ⟨r-value⟩

⟨l-value⟩ ::= ⟨id⟩ | “**result**” | ⟨string-literal⟩ | ⟨l-value⟩ “[” ⟨expr⟩ “]”
| ⟨expr⟩ “^” | “(” ⟨l-value⟩ “)”

⟨r-value⟩ ::= ⟨integer-const⟩ | “**true**” | “**false**” | ⟨real-const⟩ | ⟨char-const⟩
| “(” ⟨r-value⟩ “)” | “**nil**” | ⟨call⟩ | “@” ⟨l-value⟩
| ⟨unop⟩ ⟨expr⟩ | ⟨expr⟩ ⟨binop⟩ ⟨expr⟩

⟨call⟩ ::= ⟨id⟩ “(” [⟨expr⟩ (“,” ⟨expr⟩)*] “)”

⟨unop⟩ ::= “**not**” | “+” | “−”

⟨binop⟩ ::= “+” | “−” | “*” | “/” | “**div**” | “**mod**” | “**or**” | “**and**”
| “=” | “<>” | “<” | “<=” | “>” | “>=”

Κεφάλαιο 4

Μεταγλώττιση της γλώσσας CoPCL

Για τη μεταγλώττιση της CoPCL ακολουθήθηκαν τα εξής στάδια:

- Λεκτική ανάλυση.
- Συντακτική ανάλυση.
- Σημασιολογική ανάλυση με κατάλληλη ενημέρωση του πίνακα συμβόλων.
- Παραγωγή ενδιάμεσου κώδικα.
- Παραγωγή τελικού κώδικα.

Τα στάδια αυτά περιγράφονται στη συνέχεια του κεφαλαίου.

4.1 Λεκτική ανάλυση

Στη φάση της *λεκτικής ανάλυσης* (lexical analysis) ο μεταγλωττιστής δέχεται ως είσοδο ένα αρχικό πρόγραμμα με τη μορφή μιας συμβολοσειράς χαρακτήρων και δίνει ως έξοδο το ίδιο πρόγραμμα με τη μορφή μιας συμβολοσειράς *λεκτικών μονάδων* (tokens). Οι λεκτικές αυτές μονάδες είναι τα τερματικά σύμβολα της γραμματικής που περιγράφει τη σύνταξη της γλώσσας προς υλοποποίηση και η οποία θα χρησιμοποιηθεί στην επόμενη φάση της μεταγλώττισης, δηλαδή στη συντακτική ανάλυση. Έτσι, μπορεί κανείς να θεωρήσει ότι η λεκτική ανάλυση είναι κατ' ουσία μέρος της συντακτικής ανάλυσης. Ο λόγος που στην πράξη υλοποιείται ως ξεχωριστή φάση είναι γιατί αυτό διευκολύνει τη σχεδίαση και την υλοποίηση του μεταγλωττιστή. Το τμήμα του μεταγλωττιστή που υλοποιεί τη φάση της λεκτικής ανάλυσης λέγεται *λεκτικός αναλυτής* (lexical analyser, scanner).

Για την υλοποίηση της λεκτικής ανάλυσης για τη γλώσσα CoPCL χρησιμοποιήθηκε το μεταεργαλείο `flex`, που είναι ένας γεννήτορας λεκτικών αναλυτών. Αποτελεί βελτίωση του μεταεργαλείου `lex`, που κατασκευάστηκε από την AT&T στη δεκαετία του 1970 ως ένα από τα συνοδευτικά προγράμματα του λειτουργικού συστήματος Unix. Τα δυο εργαλεία είναι συμβατά σε αρκετά μεγάλο βαθμό και είναι σήμερα διαθέσιμα σε εκδόσεις και για προσωπικούς υπολογιστές.

Το `flex` δέχεται ως είσοδο ένα μεταπρόγραμμα που περιγράφει τις προς αναγνώριση λεκτικές μονάδες, καθώς και τις ενέργειες που πρέπει να γίνουν όταν αυτές αναγνωρισθούν. Οι λεκτικές μονάδες της γλώσσας CoPCL περιγράφονται αναλυτικά στην ενότητα 3.1. Η έξοδός του `flex` είναι ένα πρόγραμμα γραμμένο σε C, το οποίο περιέχει τη συνάρτηση `yylex`, που υλοποιεί το λεκτικό αναλυτή. Η συνάρτηση αυτή αναγνωρίζει την επόμενη λεκτική μονάδα και επιστρέφει έναν ακέραιο αριθμό, που αντιστοιχεί στον κωδικό της. Σε περίπτωση τέλους της συμβολοσειράς εισόδου, η `yylex` επιστρέφει την τιμή 0. Η ακολουθία των χαρακτήρων που απαρτίζουν την αναγνωρισθείσα λεκτική μονάδα είναι διαθέσιμη στη μεταβλητή `yytext`.

Σε περίπτωση διαπίστωσης λεκτικών σφαλμάτων καλείται μια συνάρτηση σφαλμάτος για την εκτύπωση κατάλληλων διαγνωστικών μηνυμάτων και την ανάνηψη από αυτά. Επίσης, χρησιμοποιείται ένας μετρητής, που αποθηκεύει τον τρέχοντα αριθμό γραμμής, για χρήση στα

μηνύματα αυτά. Ο μετρητής αυτός ενημερώνεται κατάλληλα κατά τη διάρκεια της λεκτικής ανάλυσης. Το μεταεργαλείο `flex` παρέχει επίσης το μηχανισμό των *αρχικών καταστάσεων* (initial states), με τον οποίο ενεργοποιούνται οι κανόνες υπό συνθήκη και απλοποιείται η διαδικασία περιγραφής των λεκτικών μονάδων της γλώσσας. Ο μηχανισμός αυτός χρησιμοποιείται στον λεκτικό αναλυτή της γλώσσας CoPCL για την περιγραφή των σχολίων και μας δίνει τη δυνατότητα να εξασφαλίζουμε την αύξηση του μετρητή των γραμμών του προγράμματος μέσα στα σχόλια. Τέλος, κάθε φορά που ο λεκτικός αναλυτής αναγνωρίζει μια λεκτική μονάδα και την επιστρέφει στο συντακτικό αναλυτή, ενημερώνεται η μεταβλητή `yylval` με τη σημασιολογική τιμή που αντιστοιχεί σε αυτό το τερματικό σύμβολο. Μέσω της μεταβλητής αυτής επιτυγχάνεται η επικοινωνία του λεκτικού με τον συντακτικό αναλυτή.

4.2 Συντακτική ανάλυση

Στη φάση της *συντακτικής ανάλυσης* (syntactic analysis, parsing) ελέγχεται αν το αρχικό πρόγραμμα, που δόθηκε ως είσοδος, ανήκει στη γλώσσα CoPCL, της οποίας η σύνταξη ορίζεται από τη *γραμματική χωρίς συμφραζόμενα*[1] (context-free grammar) της γλώσσας, που περιγράφεται στην ενότητα 3.7. Κατά τη διάρκεια αυτής της φάσης κατασκευάζεται το συντακτικό δέντρο που αντιστοιχεί στο αρχικό πρόγραμμα. Αυτό το δέντρο αποτελεί τη βάση για την παραγωγή του ενδιάμεσου κώδικα (οπότε και αποθηκεύεται για μεταγενέστερη χρήση). Το τμήμα του μεταγλωττιστή που κάνει αυτή τη δουλειά ονομάζεται *συντακτικός αναλυτής* (syntactic analyser, parser).

Οι κόμβοι του συντακτικού δέντρου περιέχουν τα σύμβολα της γραμματικής. Συγκεκριμένα, η ρίζα περιέχει το αρχικό σύμβολο, οι εσωτερικοί κόμβοι περιέχουν μη τερματικά σύμβολα, ενώ τα φύλλα περιέχουν τερματικά σύμβολα. Επιπλέον, οι απόγονοι κάθε εσωτερικού κόμβου ακολουθούν τους κανόνες παραγωγής της γραμματικής. Έστω μια γραμματική με σύνολο τερματικών συμβόλων T, σύνολο μη τερματικών N και σύνολο κανόνων παραγωγής P, όπου:

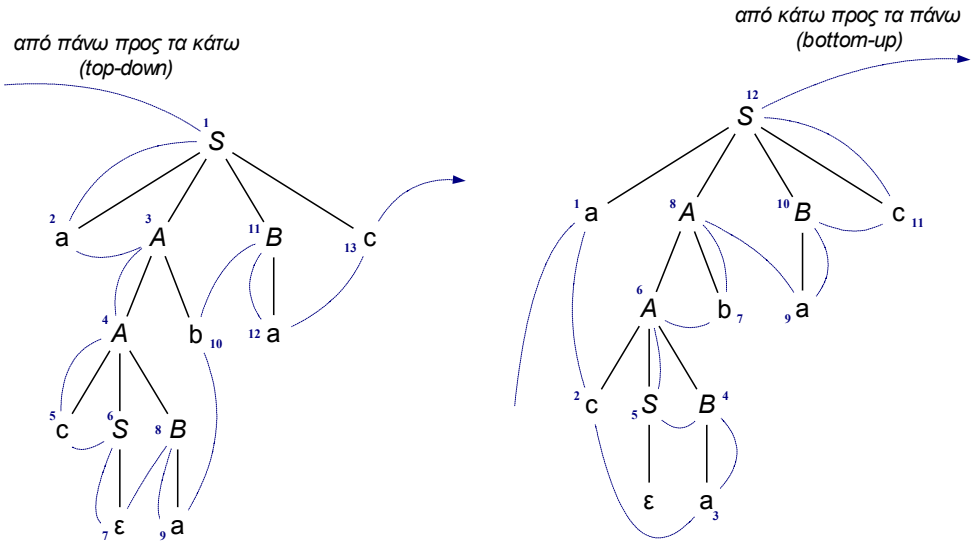
- $T = \{ \mathrm{a}, \mathrm{b}, \mathrm{c} \}$
- $N = \{ S, A, B \}$
- $P = \left\{ \begin{array}{lll} S \to \mathrm{a}AB\mathrm{c} & A \to \mathrm{c}SB & B \to \mathrm{b}B \\ S \to \epsilon & A \to A\mathrm{b} & B \to \mathrm{a} \end{array} \right\}$

Έστω τώρα μια παραγωγή αυτής της γραμματικής:

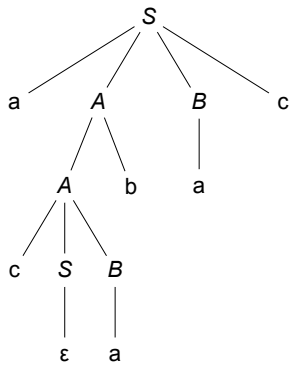
$$S \Rightarrow \mathrm{a}AB\mathrm{c} \Rightarrow \mathrm{a}A\mathrm{b}B\mathrm{c} \Rightarrow \mathrm{ac}SB\mathrm{b}B\mathrm{c} \Rightarrow \mathrm{ac}S\mathrm{ab}B\mathrm{c} \Rightarrow \mathrm{acab}B\mathrm{c} \Rightarrow \mathrm{acabac}$$

η οποία παράγει τη συμβολοσειρά acabac. Τότε το αντίστοιχο συντακτικό δέντρο είναι το ακόλουθο:

[1] Στην κλάση αυτή ανήκουν οι γραμματικές (λέγονται και *τύπου 2 γραμματικές*) με κανόνες της μορφής $A \to \alpha$, όπου A ένα μη τερματικό σύμβολο και α συμβολοσειρά. Στις γραμματικές αυτές επιτρέπεται οι κανόνες να έχουν κενό δεξί μέλος, καθώς αυτό δεν επηρεάζει την εκφραστικότητα του μοντέλου.



Σχήμα 4.1: Παράδειγμα κατασκευής συντακτικού δέντρου.



Υπάρχουν δυο βασικοί τρόποι κατασκευής αυτού του συντακτικού δέντρου δεδομένης της συμβολοσειράς εισόδου acabac. Στο σχήμα 4.1 φαίνεται η πορεία που ακολουθεί η συντακτική ανάλυση με τους δυο αυτούς τρόπους.

- Σύμφωνα με τον πρώτο τρόπο, το δέντρο κατασκευάζεται ξεκινώντας από τη ρίζα και προχωρώντας προς τα φύλλα. Οι ΣΑ που βασίζονται σε αυτό τον τρόπο ονομάζονται συντακτικοί αναλυτές *από πάνω προς τα κάτω* (top-down parsers).

- Σύμφωνα με το δεύτερο τρόπο, το δέντρο κατασκευάζεται ξεκινώντας από τα φύλλα και προχωρώντας προς τη ρίζα. Οι ΣΑ που βασίζονται στο δεύτερο τρόπο ονομάζονται συντακτικοί αναλυτές *από κάτω προς τα πάνω* (bottom-up parsers).

Για την υλοποίηση της συντακτικής ανάλυσης της γλώσσας CoPCL χρησιμοποιήθηκε το μεταεργαλείο `bison`, που είναι ένας γεννήτορας συντακτικών αναλυτών (από κάτω προς τα πάνω) για γλώσσες και γραμματικές τύπου LALR(1). Αποτελεί βελτίωση του μεταεργαλείου `yacc` που, όπως και το `lex`, κατασκευάστηκε από την AT&T στη δεκαετία του 1970 ως ένα από τα συνοδευτικά προγράμματα του λειτουργικού συστήματος Unix. Το `bison` είναι συμβατό σε αρκετά μεγάλο βαθμό με το `yacc`. Και τα δύο αυτά εργαλεία είναι σήμερα διαθέσιμα σε εκδόσεις και για προσωπικούς υπολογιστές.

Το `bison` δέχεται ως είσοδο ένα μεταπρόγραμμα που περιγράφει τη σύνταξη της αρχικής γλώσσας προγραμματισμού, καθώς και τις ενέργειες που πρέπει να γίνονται κατά την αναγνώριση των συμβολοσειρών εισόδου. Η έξοδός του είναι ένα πρόγραμμα γραμμένο σε C,

το οποίο περιέχει τη συνάρτηση `yyparse` που υλοποιεί το συντακτικό αναλυτή. Η συνάρτηση αυτή επιχειρεί να αναγνωρίσει τη συμβολοσειρά εισόδου και, παράλληλα με την κατασκευή του συντακτικού δέντρου, εκτελεί τις ενέργειες που περιγράφονται στο μεταπρόγραμμα. Για την εξάλειψη *αδιεξόδων ή συγκρούσεων* (conflicts) λόγω αμφισημιών στους κανόνες παραγωγής, απαιτήθηκε κατάλληλη μορφοποίηση αυτών, καθώς και των κανόνων προτεραιότητας και προσεταιριστικότητας των τελεστών.

Συντακτικό σφάλμα παράγεται στο `bison`, όταν ο λεκτικός αναλυτής επιστρέψει κάποια λεκτική μονάδα που δε αντιστοιχεί σε μία από τις αναμενόμενες σύμφωνα με την υπάρχουσα γραμματική, όπως αυτή περιγράφεται στους κανόνες παραγωγής του μεταπρογράμματος. Τότε καλείται αυτόματα η συνάρτηση `yyerror` του συντακτικού αναλυτή, για να εκτυπωθεί κατάλληλο διαγνωστικό μήνυμα συντακτικού λάθους. Για την ανάνηψη από σφάλματα χρησιμοποιείται η ιδιότητα που διαθέτει το `bison` να οπισθοδρομεί όταν εντοπίσει συντακτικό σφάλμα, ώσπου να φτάσει σε μια προηγούμενη κατάσταση όπου το σφάλμα να προβλέπεται. Αυτό γίνεται με τη χρήση του ειδικού συμβόλου `error` στους κανόνες παραγωγής, η οποία επιτρέπει την καθοδήγηση του συντακτικού αναλυτή για την αποτελεσματική ανάνηψη.

Το μεταπρόγραμμα που δίνεται ως είσοδος στο `bison` αναλαμβάνει τον κεντρικό έλεγχο του μεταγλωττιστή, που επιτυγχάνεται με τη συνεργασία των παρακάτω:

- του λεκτικού αναλυτή, που κατασκευάστηκε όπως περιγράφηκε στην ενότητα 4.1
- του συντακτικού αναλυτή, που περιγράφεται στους κανόνες παραγωγής του μεταπρογράμματος,
- του σημασιολογικού αναλυτή (θα αναφερθεί στην ενότητα 4.4), που υλοποιήθηκε με κατάλληλες σημασιολογικές ρουτίνες πηγαίου κώδικα C++, και ο οποίος ενεργοποιείται από την κύρια συνάρτηση (`main`) του μεταπρογράμματος του `bison`, εφόσον ο συντακτικός έλεγχος επιτύχει,
- του πίνακα συμβόλων (θα μιλήσουμε στην επόμενη ενότητα), η ενημέρωση του οποίου υλοποιείται με κατάλληλες κλήσεις συναρτήσεων στις ρουτίνες του σημασιολογικού αναλυτή, και
- του γεννήτορα ενδιάμεσου και τελικού κώδικα (για κάποιο τμήμα του αρχικού προγράμματος), που ενεργοποιούνται από το σημασιολογικό αναλυτή μετά την ολοκλήρωση της σημασιολογικής ανάλυσης (για το αντίστοιχο τμήμα του συντακτικού δέντρου).

4.3 Πίνακας συμβόλων

Ο μεταγλωττιστής μας χρειάζεται να συγκεντρώνει πληροφορίες για τα ονόματα, που εμφανίζονται στο αρχικό πρόγραμμα, και στη συνέχεια να τις χρησιμοποιεί κατά τη διάρκεια της μεταγλώττισης. Είδη ονομάτων που μπορούν να εμφανίζονται σε προγράμματα της γλώσσας CoPCL, όπως και των περισσότερων σύγχρονων γλωσσών, είναι τα παρακάτω:

- το ίδιο το πρόγραμμα,
- μεταβλητές,
- υποπρογράμματα, δηλαδή διαδικασίες ή συναρτήσεις,
- παράμετροι υποπρογραμμάτων,
- ετικέτες εντολών, στις οποίες επιτρέπεται η μετάβαση με εντολές όπως η `goto` *id*,
- σταθερές,

- τύποι δεδομένων

Οι πληροφορίες που συλλέγονται για τα ονόματα τοποθετούνται στον *πίνακα συμβόλων* (symbol table), ο οποίος είναι οργανωμένος σε εμβέλειες, ακολουθώντας τις αντίστοιχες εμβέλειες του αρχικού προγράμματος, όπως αυτό αποτυπώνεται στο συντακτικό δέντρο. Η ενημέρωση αυτή πραγματοποιείται σταδιακά πριν και κατά τη διάρκεια της σημασιολογικής ανάλυσης. Συγκεκριμένα, πριν την έναρξη της ανάλυσης αυτής τοποθετούνται στον πίνακα συμβόλων οι πληροφορίες σχετικά με τα υποπρογράμματα και τις μεταβλητές μιας δομικής μονάδας έτσι ώστε στη συνέχεια να είναι διαθέσιμα για τον σημασιολογικό έλεγχο των εντολών του σώματος της δομικής μονάδας. Έπειτα ξεκινά η σημασιολογική ανάλυση, στη διάρκεια της οποίας ελέγχεται η ορθότητα των αποθηκευμένων πληροφοριών με τη βοήθεια του συντακτικού δέντρου, αλλά και η παραπέρα ενημέρωση του πίνακα συμβόλων με την εισαγωγή στοιχείων που αφορούν τις δομές ελέγχου διεργασιών `cobegin`. Καθεμία από τις εντολές των δομών αυτών που εμφανίζονται εισάγονται στον πίνακα συμβόλων με τη μορφή φωλιασμένων διαδικασιών. Με τον τρόπο αυτό επιτυγχάνεται η χρήση των μεταβλητών που βρίσκονται στην εμβέλεια της δομικής μονάδας στην οποία ανήκει η δομή `cobegin`, κάτι που ελέγχεται μέσω του μηχανισμού των εμβελειών του πίνακα συμβόλων.

Οι πληροφορίες που αποθηκεύονται στον πίνακα συμβόλων, όπως άλλωστε ήδη είναι ορατό, χρησιμοποιούνται στα επόμενα στάδια της ανάλυσης. Για παράδειγμα, στη φάση της σημασιολογικής ανάλυσης ελέγχεται αν η χρήση των ονομάτων συμφωνεί με το είδος και τον τύπο τους. Επίσης, κατά την παραγωγή κώδικα πρέπει να είναι γνωστό για κάθε όνομα μεταβλητής ή παραμέτρου πού θα αποθηκευτεί, πόση μνήμη απαιτείται.

Η βασική απαίτηση από τον πίνακα συμβόλων ενός μεταγλωττιστή είναι να υλοποιεί αποδοτικά τις παρακάτω λειτουργίες:

- πρόσθεση ενός νέου ονόματος στην κατάλληλη εμβέλεια,
- αναζήτηση ενός ονόματος

Στις παραπάνω προδιαγραφές, η λέξη «αποδοτικά» αναφέρεται στο κόστος διαχείρισης του πίνακα συμβόλων, τόσο από πλευράς χρόνου όσο και από πλευράς μνήμης που αυτός καταλαμβάνει. Για να ελαχιστοποιηθεί ο χρόνος που απαιτείται για τις παραπάνω λειτουργίες καθώς και η χρησιμοποιούμενη μνήμη, είναι αναγκαίο να υλοποιηθεί ο πίνακας συμβόλων με μια κατάλληλη δομή δεδομένων. Η οργάνωση επομένως του πίνακα συμβόλων επηρεάζει σε αρκετά μεγάλο βαθμό τις επιδόσεις του μεταγλωττιστή σε ταχύτητα και τις απαιτήσεις του σε μνήμη.

Για το σκοπό αυτό χρησιμοποιείται η τεχνική του πίνακα κατακερματισμού με πολλαπλές εμβέλειες, όπως αυτή περιγράφεται στις ενότητες 5.4.3 και 5.4.4 στο βιβλίο αναφοράς [2].

4.4 Σημασιολογική ανάλυση

Στη μελέτη των γλωσσών προγραμματισμού διακρίνουμε συχνά δυο βασικές έννοιες: τη *σύνταξη* (syntax) και τη *σημασιολογία* (semantics), παρότι η διαχωριστική γραμμή ανάμεσα σε αυτές τις δύο δεν είναι πάντα σαφής. Η σύνταξη αναφέρεται στην μορφή και τη δομή των καλώς σχηματισμένων προγραμμάτων και ορίζεται συνήθως με τυπικό τρόπο, όπως είδαμε στις προηγούμενες ενότητες, χρησιμοποιώντας κατάλληλες γραμματικές χωρίς συμφραζόμενα. Από την άλλη πλευρά, η σημασιολογία ασχολείται με θέματα ερμηνείας των καλώς σχηματισμένων προγραμμάτων. Τις περισσότερες φορές η σημασιολογία ορίζεται με άτυπο τρόπο χρησιμοποιώντας περιγραφές σε φυσική γλώσσα.

Ένα συντακτικά σωστό πρόγραμμα, που μπορεί να παραχθεί από τη γραμματική μας χωρίς συμφραζόμενα, είναι πιθανό να περιέχει σφάλματα σημασιολογικής φύσης. Για παράδειγμα, ενώ απαγορεύεται η δήλωση μιας μεταβλητής στην ίδια εμβέλεια περισσότερες από μια φορές, ο περιορισμός αυτός δεν μπορεί να επιβληθεί από μια γραμματική χωρίς συμφραζόμενα. Επίσης, παρά το γεγονός ότι απαγορεύεται η πρόσθεση μιας συμβολοσειράς και ενός πίνακα ακεραίων

αριθμών, η συντακτική ανάλυση ενός προγράμματος αγνοεί τους τύπους των μεταβλητών και κατά συνέπεια είναι αναγκασμένη να επιτρέψει κάθε έκφραση της μορφής «$a + b$», ανεξαρτήτως των τύπων των a και b.

Για να εντοπισθούν σφάλματα όπως τα παραπάνω, θα πρέπει να αποδοθεί κάποιου είδους ερμηνεία στη συντακτική δομή του προγράμματος. Αυτού του είδους οι ερμηνείες θα πρέπει να χρησιμοποιούνται στη συνέχεια, προκειμένου να αποφεύγονται οι πολλαπλές δηλώσεις της ίδιας μεταβλητής μέσα σε μια εμβέλεια και να συσχετίζονται τα ονόματα μεταβλητών με τους τύπους που έχουν δηλωθεί για αυτές. Το τμήμα της σημασιολογικής περιγραφής μιας γλώσσας προγραμματισμού που ασχολείται με τέτοιου είδους θέματα απόδοσης ερμηνείας ονομάζεται *στατική σημασιολογία*. Έτσι, κύριος σκοπός της στατικής σημασιολογίας είναι ο εντοπισμός σημασιολογικών σφαλμάτων κατά τη διάρκεια της μεταγλώττισης.

Στο σημασιολογικό αναλυτή που υλοποιήθηκε, πραγματοποιείται στατικός σημασιολογικός έλεγχος αμέσως μετά τη συντακτική ανάλυση και την αρχική ενημέρωση του πίνακα συμβόλων που αναφέρθηκε στην ενότητα 4.3. Οι έλεγχοι που πραγματοποιούνται από το σημασιολογικό αναλυτή της γλώσσας CoPCL, καθορίζονται με άτυπο τρόπο στην ενότητα 3, χρησιμοποιώντας περιγραφές σε φυσική γλώσσα και παραδείγματα. Ενδεικτικά παραδείγματα τέτοιων σημασιολογικών ελέγχων είναι:

- *Έλεγχοι τύπων*, π.χ. ο μεταγλωττιστής θα πρέπει να διαγνώσει σφάλμα, αν ένας τελεστής εφαρμόζεται σε μη επιτρεπόμενα τελούμενα.

- *Έλεγχοι ύπαρξης ονομάτων*, π.χ. ο μεταγλωττιστής θα πρέπει να διαγνώσει σφάλμα αν ένα όνομα χρησιμοποιείται χωρίς να έχει προηγουμένως δηλωθεί.

- *Έλεγχοι μοναδικότητας*, π.χ. ο μεταγλωττιστής θα πρέπει να διαγνώσει σφάλμα αν μια μεταβλητή δηλώνεται δυο φορές μέσα στην ίδια εμβέλεια.

Οι έλεγχοι μοναδικότητας και ύπαρξης ονομάτων πραγματοποιούνται σε συνεργασία με τον πίνακα συμβόλων. Η σημασιολογική ανάλυση ζητάει από τον πίνακα συμβόλων την πληροφορία αν υπάρχει κάποιο όνομα που συναντάται και ανάλογα το αποτέλεσμα που επιστρέφει ο πίνακας συμβόλων προχωράει στη μεταγλώττιση ή ενημερώνει για το ενδεχόμενο σφάλμα.

Αν ένα συντακτικά σωστό πρόγραμμα περάσει από τη φάση του σημασιολογικού ελέγχου χωρίς να διαγνωσθούν σφάλματα, αυτό γενικά δεν εξασφαλίζει την απουσία σφαλμάτων και κατά την εκτέλεσή του. Τα σφάλματα κατά τη διάρκεια του χρόνου εκτέλεσης των προγραμμάτων εντάσσονται στη δυναμική σημασιολογία της αρχικής γλώσσας και δεν τα αντιμετωπίζουμε κατά τη διάρκεια της μεταγλώττισης λόγω πολυπλοκότητας και μειωμένης απόδοσης.

4.5 Ενδιάμεσος κώδικας

Αντί να μεταφράζουν απευθείας το αρχικό πρόγραμμα στην τελική γλώσσα, οι περισσότεροι μεταγλωττιστές το μεταφράζουν πρώτα σε ένα ισοδύναμο πρόγραμμα γραμμένο σε κάποια *ενδιάμεση γλώσσα* (intermediate language), το οποίο στη συνέχεια μεταφράζουν στην τελική γλώσσα. Η μετάφραση δηλαδή γίνεται σε δύο διαδοχικά βήματα. Η ενδιάμεση γλώσσα είναι χαμηλότερου επιπέδου από την αρχική γλώσσα, αλλά υψηλότερου επιπέδου από την τελική. Το ισοδύναμο πρόγραμμα στην ενδιάμεση γλώσσα ονομάζεται *ενδιάμεσος κώδικας* (intermediate code) και το τμήμα του μεταγλωττιστή που το παράγει ονομάζεται *γεννήτορας ενδιάμεσου κώδικα* (intermediate code generator). Οι λόγοι που συνηγορούν στη μετάφραση πρώτα σε ενδιάμεσο κώδικα είναι οι ακόλουθοι:

- Διευκολύνει το έργο της μετάφρασης, η οποία γίνεται ευκολότερα σε δυο βήματα παρά σε ένα.

- Η βελτιστοποίηση του παραγόμενου κώδικα είναι ευκολότερη και αποτελεσματικότερη όταν γίνεται στον ενδιάμεσο κώδικα. Αυτό φυσικά δεν εμποδίζει την περαιτέρω βελτιστοποίηση του τελικού κώδικα.

- Διευκολύνει την κατάτμηση του μεταγλωττιστή σε εμπρόσθιο και οπίσθιο τμήμα, παρέχοντας ακριβώς την κοινή γλώσσα με την οποία επικοινωνούν αυτά τα δυο τμήματα. Συγκεκριμένα, ορισμένες από τις φάσεις της μεταγλώττισης σχετίζονται μόνο με την αρχική γλώσσα και είναι ανεξάρτητες από την τελική. Οι φάσεις αυτές αποτελούν το *εμπρόσθιο τμήμα* (front-end) του μεταγλωττιστή, το οποίο τυπικά περιλαμβάνει τη λεκτική, τη συντακτική και τη σημασιολογική ανάλυση, την παραγωγή και τη βελτιστοποίηση ενδιάμεσου κώδικα. Αντίθετα, άλλες φάσεις σχετίζονται μόνο με την τελική γλώσσα και τοποθετούνται στο *οπίσθιο τμήμα* (back-end) του μεταγλωττιστή, που περιλαμβάνει την παραγωγή και τη βελτιστοποίηση τελικού κώδικα. Η οργάνωση αυτή, σε δύο τμήματα, διευκολύνει την κατασκευή μεταγλωττιστών για πολλές αρχικές και πολλές τελικές γλώσσες. Έστω ότι απαιτείται να υλοποιηθούν μεταγλωττιστές για n διαφορετικές γλώσσες προγραμματισμού που να κατασκευάζουν εκτελέσιμα προγράμματα για m διαφορετικούς υπολογιστές, χρησιμοποιώντας μια κοινή γλώσσα υλοποίησης. Εν συνόλω απαιτούνται $n \times m$ μεταγλωττιστές. Με το διαχωρισμό όμως καθενός σε εμπρόσθιο και οπίσθιο τμήμα και με την υιοθέτηση της ίδιας ενδιάμεσης γλώσσας για όλους τους μεταγλωττιστές, απαιτείται μόνο η υλοποίηση n εμπρόσθιων και m οπίσθιων τμημάτων, δηλαδή $n + m$ απλούστερων μεταγλωττιστών.

Για την μετάφραση σε ενδιάμεσο κώδικα υπάρχουν δύο κύριες τεχνικές. Η *μετάφραση οδηγούμενη από τη σύνταξη* (syntax-directed translation) και η *μετάφραση οδηγούμενη από τη σημασιολογία* (semantics-directed translation). Στον μεταγλωττιστή της CoPCL χρησιμοποιείται η δεύτερη τεχνική. Στις σημασιολογικές ρουτίνες του που βρίκονται στο συντακτικό αναλυτή, αφότου ολοκληρωθεί ο σημασιολογικός έλεγχος μιας δομικής μονάδας, δίνεται η εντολή για την παραγωγή του ενδιάμεσου κώδικα. Έτσι λοιπόν παράλληλα με τη σημασιολογική ανάλυση έχουμε και το πρώτο στάδιο μετάφρασης, το οποίο ακολουθείται στη συνέχεια από τη μετάφραση του ενδιάμεσου σε τελικό κώδικα.

Ως ενδιάμεση γλώσσα επιλέχθηκε η γλώσσα των *τετράδων* (quadruples). Οι τετράδες είναι απλές εντολές, κάθε μια από τις οποίες έχει τη μορφή:

$$n\text{:}\ \ op, x, y, z$$

όπου n ο αριθμός της τετράδας (μια ετικέτα), op ένας τελεστής και x, y, z τα τελούμενα. Στην περίπτωση που το op είναι αριθμητικός τελεστής με δυο τελούμενα (π.χ. ο τελεστής της πρόσθεσης ακεραίων), τότε τα τελούμενα μπορούν να θεωρηθούν ως μεταβλητές και η παραπάνω τετράδα έχει την έννοια:

$$z \ := \ x \ op \ y$$

Άλλες εντολές έχουν διαφορετική ερμηνεία, που γενικά εξαρτάται από τον τελεστή που χρησιμοποιείται. Όταν ο τελεστής op δε χρειάζεται τρία τελούμενα, τότε στη θέση όσων δε χρειάζονται τοποθετείται το σύμβολο $-$.

Για τη μετάφραση από την αρχική γλώσσα στη γλώσσα τετράδων, απαιτείται ο τεμαχισμός πολύπλοκων υπολογισμών σε πολλούς απλούστερους. Τα ενδιάμεσα αποτελέσματα των απλούστερων υπολογισμών τοποθετούνται σε *προσωρινές μεταβλητές*, τις οποίες στη συνέχεια θα συμβολίζουμε με $\$n$, όπου n φυσικός αριθμός. Για παράδειγμα:

Η έκφραση "`b*b - 4*a*c`" μεταφράζεται στην παρακάτω ακολουθία τετράδων, θεωρώντας ότι η ενδιάμεση γλώσσα χρησιμοποιεί τα ίδια σύμβολα για τους τελεστές και τα τελούμενα όπως η αρχική γλώσσα:

1: $*, \mathrm{b}, \mathrm{b}, \$1$
2: $*, 4, \mathrm{a}, \$2$
3: $*, \$2, \mathrm{c}, \$3$
4: $-, \$1, \$3, \$4$

Το τελικό αποτέλεσμα βρίσκεται στην προσωρινή μεταβλητή $4.

Για τη μετατροπή των γλωσσικών δομών της CoPCL σε ενδιάμεσο κώδικα χρησιμοποιείται ένα σχέδιο, που ονομάζεται *σχέδιο παραγωγής ενδιάμεσου κώδικα*. Η σύνταξη, η σημασιολογία και το σχέδιο παραγωγής της βασικής μορφής της ενδιάμεσης αυτής γλώσσας που χρησιμοποιήθηκε, αναπτύσσονται εκτενώς στην ενότητα 7.2 στο βιβλίο αναφοράς [2]. Ενδεικτικά εδώ θα δοθούν τα τελούμενα και οι τελεστές της γλώσσας των τετράδων για τη γλώσσα CoPCL, η μορφή των τετράδων αυτών και μια μικρή περιγραφή της λειτουργίας κάθε τέτοιας τετράδας, με αναφορές στα νέα στοιχεία και στις διαφοροποιήσεις που εισήχθησαν λόγω της γλώσσας CoPCL.

4.5.1 Τελούμενα

Το πραγματικό πλήθος και το είδος των τελουμένων μιας τετράδας εξαρτάται από τον τελεστή της. Οι μορφές που μπορεί να έχει ένα τελούμενο, παρουσιάζονται στη συνέχεια.

- *Σταθερά.* Στην περίπτωση της CoPCL, οι σταθερές μπορούν να είναι ακέραιες, πραγματικές, λογικές, χαρακτήρες, συμβολοσειρές, ή ο μηδενικός δείκτης `nil`.

- *Όνομα* μεταβλητής, παραμέτρου ή υποπρογράμματος. Στην πραγματικότητα, το τελούμενο σε αυτή την περίπτωση είναι ένας δείκτης στην αντίστοιχη εγγραφή στον πίνακα συμβόλων, μέσω της οποίας μπορούν να αναζητηθούν περισσότερες πληροφορίες όπως το είδος του ονόματος, η εμβέλειά του, ο τύπος του.

- *Προσωρινή μεταβλητή*, η οποία έχει τη μορφή $\$n$, όπου n φυσικός αριθμός. Οι προσωρινές μεταβλητές χρησιμεύουν για την αποθήκευση των ενδιάμεσων αποτελεσμάτων διαφόρων πράξεων και αριθμούνται σε αύξουσα σειρά. Αποθηκεύονται και αυτές στον πίνακα συμβόλων, οπότε και σε αυτή την περίπτωση το τελούμενο είναι στην πραγματικότητα ένας δείκτης στην αντίστοιχη εγγραφή.

Οι παραπάνω μορφές τελουμένων ονομάζονται *απλές*. Σε κάθε ένα απλό τελούμενο μπορεί κανείς να αντιστοιχίσει ένα τύπο δεδομένων της γλώσσας CoPCL, ο οποίος προκύπτει με προφανή τρόπο από την παραπάνω περιγραφή. Εκτός από τις απλές μορφές τελουμένων, υποστηρίζονται όμως και δυο *σύνθετες* μορφές, η ύπαρξη των οποίων σχετίζεται με τους τύπους πίνακα και δείκτη που υποστηρίζει η γλώσσα CoPCL.

- *Αποδεικτοδότηση.* Αν x είναι ένα απλό τελούμενο τύπου $\hat{}t$, τότε το τελούμενο $[x]$ έχει τύπο t και συμβολίζει το αντικείμενο στο οποίο δείχνει το x.

- *Διεύθυνση.* Αν x είναι ένα απλό τελούμενο με τη μορφή μεταβλητής ή παραμέτρου τύπου t, τότε το τελούμενο $\{x\}$ έχει τύπο $\hat{}t$ και συμβολίζει τη διεύθυνση της μεταβλητής ή της παραμέτρου.

Οι μορφές τελουμένων που περιγράφηκαν ως τώρα αφορούν σε δεδομένα, δηλαδή αντικείμενα στη μνήμη του υπολογιστή και τιμές που περιέχονται σε αυτά. Εκτός όμως από αυτά, υπάρχουν και μερικές ακόμα μορφές τελουμένων που συναντώνται σε τετράδες:

- *Ετικέτα τετράδας*, η οποία χρησιμοποιείται για την πραγματοποίηση άλματος σε αυτή την τετράδα, είτε υπό συνθήκη είτε όχι.

⟨program⟩ ::= (⟨quadruple⟩)*
⟨quadruple⟩ ::= ⟨label⟩ ":" ⟨opname⟩ "," ⟨operand⟩ "," ⟨operand⟩ "," ⟨operand⟩
⟨label⟩ ::= ⟨integer-const⟩
⟨opname⟩ ::= "**unit**" | "**endu**" | "+" | "−" | "∗" | "/" | "%" | "**div**"
| ":=" | "**array**" | "=" | "<>" | ">" | "<" | ">=" | "<="
| "**ifb**" | "**label**" | "**jumpl**" | "**call**" | "**par**" | "**ret**"
| "**create**" | "**join**" | "**lock**" | "**unlock**" | "**wait**" | "**signal**"
⟨operand⟩ ::= ⟨simple⟩ | "[" ⟨simple⟩ "]" | "{" ⟨simple⟩ "}"
| ⟨label⟩ | ⟨pass-mode⟩ | "∗" | "−"
⟨simple⟩ ::= ⟨constant⟩ | ⟨object⟩ | ⟨temporary⟩
⟨pass-mode⟩ ::= "**V**" | "**R**" | "**RET**"
⟨constant⟩ ::= ⟨integer-const⟩ | ⟨double-const⟩ | "**true**" | "**false**"
| ⟨char-const⟩ | ⟨string-literal⟩ | "**nil**" | "**NULL**"
⟨object⟩ ::= ⟨id⟩
⟨temporary⟩ ::= "$" ⟨integer-const⟩

Σχήμα 4.2: Μια γραμματική για τη γλώσσα των τετράδων.

- *Ετικέτα εντολής*, η οποία χρησιμοποιείται για άλματα με την εντολή goto *id*. Οι ετικέτες αυτές φυλάσσονται επίσης στον πίνακα συμβόλων, οπότε ως τελούμενα δε διαφοροποιούνται από τα λοιπά ονόματα.

- *Τρόπος περάσματος* παραμέτρου σε μέθοδο. Η CoPCL υποστηρίζει δυο τρόπους περάσματος: κατ' αξία (by value) και κατ' αναφορά (by reference), που ως τελούμενα σε τετράδες παριστάνονται αντίστοιχα με τα γράμματα V και R. Εκτός όμως από αυτούς, στην περίπτωση κλήσης σε συνάρτηση η θέση όπου φυλάσσεται το αποτέλεσμα περνά και αυτή ως παράμετρος, χρησιμοποιώντας τον ειδικό κωδικό RET.

- *Άγνωστη ετικέτα τετράδας*. Σε ορισμένες περιπτώσεις κατά την παραγωγή του ενδιάμεσου κώδικα, στις τετράδες που πραγματοποιούν άλματα δεν είναι γνωστή η ετικέτα της τετράδας στόχου. Αυτό γίνεται όταν η τετράδα στόχου δεν έχει ακόμα παραχθεί και διορθώνεται στη συνέχεια όταν εκείνη παραχθεί. Η κενή θέση που θα συμπληρωθεί αργότερα με μια ετικέτα τετράδας βρίσκεται πάντα στο τελευταίο τελούμενο της τετράδας και συμβολίζεται με ∗. Μετά την ολοκλήρωση της φάσης παραγωγής του ενδιάμεσου κώδικα δεν πρέπει να υπάρχουν τέτοια τελούμενα σε καμία τετράδα.

4.5.2 Τελεστές

Ακολουθούν όλοι οι δυνατοί τελεστές των τετράδων, η μορφή αυτών των τετράδων και μια σύντομη περιγραφή τους. Μια τυπική περιγραφή του ενδιάμεσου κώδικα ενός προγράμματος σε μορφή τετράδων δίνεται στη γραμματική του σχήματος 4.2.

- Τετράδες unit, I, −, −
  και endu, I, −, −

  Ο συνολικός ενδιάμεσος κώδικας που παράγεται για ένα πρόγραμμα είναι χωρισμένος σε δομικές μονάδες. Σε κάθε δομική μονάδα του αρχικού προγράμματος αντιστοιχεί μια σειρά τετράδων. Οι τετράδες unit και endu σηματοδοτούν την έναρξη και το τέλος του ενδιάμεσου κώδικα που αντιστοιχεί σε μια δομική μονάδα του αρχικού προγράμματος. Το τελούμενο I περιέχει το όνομα της δομικής μονάδας. Επίσης, οι τετράδες αυτές χρησιμοποιούνται για την συγκέντρωση των εντολών μιας διεργασίας που δημιουργείται

με χρήση της εντολής `cobegin`. Το όνομα που περιέχουν οι τετράδες αυτές παράγεται αυτόματα και είναι της μορφής $co@i$ όπου i ο αύξων αριθμός της αντίστοιχης προσωρινής μεταβλητής που χρησιμοποιείται για την αποθήκευση της ταυτότητας της διεργασίας. Με το ίδιο όνομα εισάγονται οι μονάδες αυτές στον πίνακα συμβόλων όπως ακριβώς οι διαδικασίες.

- Τετράδα op, x, y, z
  όπου ο τελεστής op είναι ένας από τους τελεστές `+`, `-`, `*`, `/`, `div` και `%`

  Οι τετράδες αυτές αντιστοιχούν στις πράξεις της πρόσθεσης, της αφαίρεσης, του πολλαπλασιασμού, της διαίρεσης, του πηλίκου και του υπόλοιπου της ακέραιας διαίρεσης. Το αποτέλεσμα της πράξης $x\ op\ y$ υπολογίζεται και αποθηκεύεται στο τελούμενο z. Πρέπει να σημειωθεί ότι οι τέσσερις βασικές πράξεις εφαρμόζονται είτε σε ακέραια τελούμενα x και y είτε σε πραγματικά. Το αποτέλεσμα z είναι ακέραιο μόνο όταν και τα δυο τελούμενα x και y είναι ακέραια. Αντίθετα, η πράξη `%` εφαρμόζεται μόνο σε ακέραια τελούμενα.

  Στην περίπτωση του τελεστή `-` είναι δυνατόν να λείπει το τελούμενο y. Σε αυτή την περίπτωση, η τετράδα $-, x, -, z$ υλοποιεί το αρνητικό πρόσημο: το αποτέλεσμα της πράξης $-x$ αποθηκεύεται στο z.

- Τετράδα $:=, x, -, z$

  Η τετράδα αυτή εκτελεί τη λειτουργία της ανάθεσης. Η τιμή του x ανατίθεται στο z. Ο τύπος του x πρέπει να είναι συμβατός για ανάθεση με τον τύπο του z. Αυτή η έννοια της συμβατότητας για ανάθεση περιγράφεται στην ενότητα 3.4.3.

- Τετράδα array, x, y, z

  Η τετράδα αυτή εκτελεί τη λειτουργία της αναφοράς σε στοιχείο πίνακα. Το τελούμενο x πρέπει να είναι τύπου πίνακα από t, ενώ το y πρέπει να είναι τύπου `integer`. Επίσης, το τελούμενο z πρέπει να είναι τύπου `^`t. Η τελική τιμή του z είναι ένας δείκτης στο στοιχείο $x[y]$.

- Τετράδα op, x, y, z
  όπου ο τελεστής op είναι ένας από τους τελεστές `=`, `<>`, `>`, `<`, `>=` και `<=`

  Οι τετράδες αυτές υλοποιούν άλματα υπό συνθήκη: αν η συνθήκη $x\ op\ y$ είναι αληθής, τότε ο έλεγχος μεταφέρεται στην τετράδα με ετικέτα z. Οι τύποι των τελουμένων x και y πρέπει είτε να είναι αριθμητικοί ή, στην περίπτωση των τελεστών `=` και `<>`, να συμπίπτουν και να μην είναι τύπος πίνακα.

- Τετράδα $\text{ifb}, x, -, z$

  Η τετράδα αυτή υπολογίζει την τιμή του τελούμενου x, που πρέπει να έχει τύπο `bool`, και αν αυτή είναι αληθής πραγματοποιεί άλμα στην τετράδα με ετικέτα z.

- Τετράδα $\text{jump}, -, -, z$

  Η τετράδα αυτή πραγματοποιεί άλμα στην τετράδα με ετικέτα z.

- Τετράδες $\text{label}, I, -, -$ και $\text{jumpl}, -, -, I$

  Με την τετράδα label ορίζεται απλώς η θέση που αντιστοιχεί στην ετικέτα με όνομα I, ενώ τα άλματα στην ετικέτα αυτή πραγματοποιούνται με την τετράδα jumpl.

- Τετράδα $\text{call}, -, -, I$

  Με αυτή την τετράδα γίνεται η κλήση του υποπρογράμματος με όνομα I.

- Τετράδα $\text{par}, x, m, -$

  Πριν την κλήση ενός υποπρογράμματος, πρέπει να έχει προηγηθεί το πέρασμα των πραγματικών παραμέτρων. Αυτό γίνεται με την τετράδα par. Το τελούμενο x είναι η πραγματική παράμετρος. Το m συμβολίζει τον τρόπο περάσματος και μπορεί να πάρει τις εξής τιμές:

  - V : πέρασμα κατ' αξία (by value),
  - R : πέρασμα κατ' αναφορά (by reference), ή
  - RET : καθορισμός της θέσης του αποτελέσματος συνάρτησης.

  Ο τύπος και το είδος της πραγματικής παραμέτρου πρέπει να συμφωνούν με τον τύπο και το είδος της αντίστοιχης τυπικής παραμέτρου του υποπρογράμματος που καλείται ή να πληρούνται οι προϋποθέσεις που περιγράφονται στην ενότητα 3.4.4.

- Τετράδα $\text{ret}, -, -, -$

  Με την τετράδα ret τερματίζεται η εκτέλεση της τρέχουσας δομικής μονάδας και ο έλεγχος επιστρέφεται σε αυτήν που την κάλεσε.

  Ας σημειωθεί επίσης ότι, αν ο έλεγχος φτάσει στην τετράδα endu που βρίσκεται στο τέλος μιας δομικής μονάδας, τότε αυτομάτα γίνεται επιστροφή στην καλούσα δομική μονάδα σαν να υπήρχε εκεί τετράδα ret.

- Τετράδα $\text{create}, -, I, x$

  Η τετράδα αυτή χρησιμοποιείται για τη δημιουργία μιας διεργασίας που θα εκτελέσει τη διαδικασία με όνομα το τελούμενο I, ενώ στο τελούμενο x αποθηκεύεται ένας αναγνωριστικός αριθμός της διεργασίας που θα δημιουργηθεί.

- Τετράδα $\text{join}, -, -, x$

  Η τετράδα αυτή χρησιμοποιείται για να αναστείλει την εκτέλεση της καλούσας διεργασίας έως ότου η διεργασία με αναγνωριστικό αριθμό αυτόν που βρίσκεται στο x τερματιστεί.

- Τετράδες $\text{lock}, -, -, I$ και $\text{unlock}, -, -, I$

  Οι τετράδες αυτές χρησιμοποιούνται για το κλείδωμα και ξεκλείδωμα αντίστοιχα της μεταβλητής συνθήκης με όνομα I ώστε να εκτελεσθεί αναμεσά τους κρίσιμο τμήμα.

- Τετράδες $\text{wait}, -, -, I$ και $\text{signal}, -, -, I$

  Οι τετράδες αυτές χρησιμοποιούνται για την αδρανοποίηση μιας διεργασίας και την πρόσθεσή της στην ουρά αναμονής εξυπηρέτησης της μεταβλητής συνθήκης με όνομα I και για την αποστολή ειδοποίησης αφύπνισης στην πρώτη διεργασία που βρίσκεται στην ουρά της μεταβλητής αυτής ώστε να συνεχίσει την εκτέλεσή της.

4.6 Τελικός κώδικας

Η σπουδαιότερη και δυσκολότερη φάση της μετάφρασης ενός προγράμματος είναι η φάση της *παραγωγής του τελικού κώδικα* (code generation). Το κομμάτι του μεταγλωττιστή που υλοποιεί αυτή τη φάση ονομάζεται *γεννήτορας τελικού κώδικα* (code generator). Ο γεννήτορας τελικού κώδικα δέχεται στην είσοδο το αρχικό πρόγραμμα σε μορφή ενδιάμεσου κώδικα, μαζί με πληροφορίες που υπάρχουν στον πίνακα συμβόλων. Οι πληροφορίες αυτές χρησιμοποιούνται για να υπολογιστούν οι διευθύνσεις μνήμης που θα έχουν τα ονόματα κατά την εκτέλεση. Η παραγωγή τελικού κώδικα για τη γλώσσα CoPCL πραγματοποιείται για κάποια δομική μονάδα του αρχικού προγράμματος, αφού πρώτα αυτό έχει μεταφρασθεί σε ενδιάμεσο κώδικα και δεν έχουν βρεθεί σφάλματα. Η έξοδος του γεννήτορα τελικού κώδικα είναι το τελικό πρόγραμμα.

Ο τελικός κώδικας δίνεται σε συμβολική γλώσσα σύμφωνα με το πρότυπο του συμβολομεταφραστή NASM της Netwide. Το τελικό πρόγραμμα αποτελείται από ένα τμήμα μνήμης στο οποίο βρίσκεται ο κώδικας και η στοίβα εκτέλεσης, ενώ ακολουθείται το μοντέλο εκτέλεσης ELF.

Ολόκληρο το πρόγραμμα περιέχεται σε ένα τμήμα μνήμης, το οποίο χρησιμοποιείται για κώδικα, δεδομένα και στοίβα. Το γεγονός αυτό διευκολύνει τις αναφορές στη μνήμη, εφόσον δε χρειάζεται να καθορίζεται το τμήμα, στο οποίο περιέχεται κάθε δεδομένο. Αυτό δεν επιβάλλει κάποιο περιορισμό σε μνήμη εφόσον η διευθυνσιοδότηση που χρησιμοποιούμε γίνεται με διευθύνσεις των 32-bit.

4.6.1 Εγγράφημα δραστηριοποίησης

Κατά την παραγωγή του τελικού κώδικα για την CoPCL, τα τοπικά δεδομένα μιας δομικής μονάδας αποθηκεύονται κατά τη διάρκεια της εκτέλεσης του προγράμματος σε ένα τμήμα της μνήμης που ονομάζεται *εγγράφημα δραστηριοποίησης* (activation record, frame). Το εγγράφημα δραστηριοποίησης (ΕΔ) έχει θέσεις μεταξύ των άλλων για την αποθήκευση παραμέτρων, αποτελεσμάτων, πληροφοριών κατάστασης μηχανής, τοπικών μεταβλητών και προσωρινών μεταβλητών. Το μέγεθος και η μορφή του καθορίζεται από τις πληροφορίες για τα ονόματα που βρίσκονται στον πίνακα συμβόλων.

Η προσπέλαση στις πληροφορίες που περιέχονται σε ένα ΕΔ γίνεται μέσω ενός δείκτη που ονομάζεται *δείκτης πλαισίου* (frame pointer). Στους περισσότερους μεταγλωττιστές που κατασκευάζονται για επεξεργαστές 32-bit της Intel ο δείκτης πλαισίου τοποθετείται στον καταχωρητή ebp. Επομένως, οι πληροφορίες που περιέχονται στο ΕΔ μπορούν να προσπελαστούν στη συμβολική γλώσσα μέσω ορισμάτων της μορφής [ebp + *offset*], όπου *offset* η απόκλιση της πληροφορίας από το σημείο του ΕΔ στο οποίο δείχνει ο δείκτης πλαισίου.

Για την υλοποίηση του τελικού κώδικα του μεταγλωττιστή για την CoPCL, οι πληροφορίες που αποθηκεύονται σε κάθε ΕΔ είναι οι εξής:

- Στις πρώτες θέσεις τοποθετούνται οι πραγματικές παράμετροι της αντίστοιχης κλήσης της μεθόδου.

- Στη συνέχεια σε ένα σταθερό τμήμα, με την έννοια ότι η μορφή του είναι κοινή για όλες τις κλήσεις μεθόδων, αποθηκεύονται η διεύθυνση του αποτελέσματος, ο σύνδεσμος προσπέλασης, η διεύθυνση επιστροφής και η προηγούμενη τιμή του δείκτη πλαισίου, δηλαδή ένας δείκτης στο εγγράφημα δραστηριοποίησης της μεθόδου μέσα στην οποία γίνεται η κλήση.

- Τέλος, αποθηκεύονται οι τοπικές μεταβλητές και οι προσωρινές μεταβλητές.

Στο σχήμα 4.3 δίνεται η γενική μορφή ενός ΕΔ.

Τα ΕΔ αποθηκεύονται σε μια περιοχή της μνήμης που ονομάζεται *στοίβα εκτέλεσης*, στην κορυφή της οποίας δείχνει ο καταχωρητής esp. Τα ΕΔ τοποθετούνται σ' αυτή την περιοχή κατά την εκτέλεση του προγράμματος. Πρώτα τοποθετείται το ΕΔ του κυρίου προγράμματος. Μετά, κάθε φορά που καλείται ένα υποπρόγραμμα, τοποθετείται το αντίστοιχο ΕΔ στον ελεύθερο χώρο που υπάρχει σ' αυτή την περιοχή και παραμένει εκεί έως ότου τελειώσει η εκτέλεση του υποπρογράμματος, οπότε και απομακρύνεται. Το μήκος αυτής της περιοχής αυξομειώνεται κατά την εκτέλεση σύμφωνα με τις δραστηριοποιήσεις των υποπρογραμμάτων.

Όταν καλείται ένα υποπρόγραμμα, το υποπρόγραμμα που το καλεί τοποθετεί στη στοίβα τις πραγματικές παραμέτρους και τα στοιχεία του σταθερού τμήματος του ΕΔ, μέχρι τη διεύθυνση επιστροφής, και μεταφέρει τον έλεγχο στο καλούμενο υποπρόγραμμα. Πολύ σημαντικό είναι να τοποθετηθεί ο σύνδεσμος προσπέλασης, όπως θα δούμε παρακάτω. Όταν ο έλεγχος επιστρέφει, το υποπρόγραμμα που κάλεσε αφαιρεί από τη στοίβα εκτέλεσης το τμήμα του ΕΔ

Παράμετρος 1	
Παράμετρος 2	
...	...
Παράμετρος n	ebp+16
Διεύθυνση αποτελέσματος	ebp+12
Σύνδεσμος προσπέλασης	ebp+8
Διεύθυνση επιστροφής	ebp+4
Προηγούμενο ebp	ebp
Τοπική μεταβλητή 1	ebp-4
Τοπική μεταβλητή 2	ebp-8
...	...
Τοπική μεταβλητή m	
Προσωρινή μεταβλητή 1	
Προσωρινή μεταβλητή 2	
...	
Προσωρινή μεταβλητή k	

Σχήμα 4.3: Πληροφορίες εγγραφήματος δραστηριοποίησης.

του καλούμενου υποπρογράμματος που αυτό δημιούργησε. Σημειώνεται ότι ο δυναμικός τρόπος με τον οποίο γίνεται η καταχώρηση μνήμης στα ΕΔ, επιτρέπει την ταυτόχρονη ύπαρξη περισσότερων του ενός ΕΔ του ίδιου υποπρογράμματος μέσα στη στοίβα εκτέλεσης. Έτσι, μπορεί να υλοποιηθεί αναδρομική κλήση υποπρογραμμάτων.

4.6.2 Μη τοπικά δεδομένα

Κατά την εκτέλεση μιας δομικής μονάδας του προγράμματος, τα τοπικά τελούμενα βρίσκονται στο αντίστοιχο ΕΔ που υπάρχει στην κορυφή της στοίβας εκτέλεσης. Τα μη τοπικά τελούμενα βρίσκονται σε ΕΔ τα οποία είναι τοποθετημένα σε προηγούμενα σημεία μέσα στη στοίβα εκτέλεσης. Έτσι, ο τρόπος που γίνεται η προσπέλαση στα τοπικά τελούμενα είναι διαφορετικός από εκείνον με τον οποίο προσπελαύνονται τα μη τοπικά. Όπως ήδη αναφέρθηκε, τα τοπικά τελούμενα (παράμετροι, τοπικές μεταβλητές, προσωρινές μεταβλητές) μπορούν να προσπελασθούν με το δεικτοδοτημένο τρόπο αναφοράς στη μνήμη, δηλαδή με τελούμενα της μορφής [ebp + *offset*] σε συμβολική γλώσσα, όπου *offset* είναι η σχετική διεύθυνση του τελούμενου στο ΕΔ (θετική ή αρνητική) και ebp ο δείκτης στη βάση του τρέχοντος ΕΔ που βρίσκεται στη στοίβα εκτέλεσης.

Η προσπέλαση μη τοπικών τελούμενων (μη τοπικές παράμετροι, μη τοπικές μεταβλητές) είναι πιο πολύπλοκη. Απαιτείται αρχικά να βρεθεί η διεύθυνση της βάσης του αντίστοιχου ΕΔ, το οποίο σε αυτήν την περίπτωση δεν βρίσκεται στην κορυφή της στοίβας εκτέλεσης, και να τοποθετηθεί αυτή η διεύθυνση σε ένα καταχωρητή (π.χ. στον esi). Έπειτα, μπορεί να χρησιμοποιηθεί ο ίδιος τρόπος προσπέλασης με εκείνον που περιγράφηκε για τα τοπικά τελούμενα, με τη διαφορά ότι θα χρησιμοποιείται αυτός ο καταχωρητής αντί του δείκτη πλαισίου. Δηλαδή, ένα μη τοπικό τελούμενο βρίσκεται στη θέση [esi + *offset*], όπου *offset* είναι η σχετική διεύθυνση του τελούμενου στο ΕΔ του οποίου η διεύθυνση βρίσκεται στον καταχωρητή esi.

Για τη λύση αυτού του προβλήματος χρησιμοποιήθηκαν οι *σύνδεσμοι προσπέλασης*. Η τεχνική στηρίζεται στην έννοια του βάθους φωλιάσματος ενός υποπρογράμματος. Κατά τη συντακτική ανάλυση ενός αρχικού προγράμματος που έχει δομή ενοτήτων, μπορεί να αντιστοιχισθεί ένας φυσικός αριθμός σε κάθε δομική μονάδα, ο οποίος αντιστοιχεί στο βάθος που έχει αυτή η μονάδα μέσα στη δενδρική ιεραρχία των δομικών μονάδων. Ο αριμός αυτός λέγεται *βάθος φωλιάσματος* (nesting depth) και υπολογίζεται ως εξής: το κύριο πρόγραμμα έχει βάθος φωλιάσματος 1, ενώ όλα τα υποπρογράμματα που είναι στο ίδιο επίπεδο της ιεραρχίας έχουν το ίδιο βάθος φωλιάσματος που είναι κατά ένα μεγαλύτερο από το βάθος φωλιάσματος του

υποπρογράμματος μέσα στο οποίο αυτά είναι φωλιασμένα. Το ίδιο συμβαίνει και για τις εντολές μιας δομής `cobegin-coend`. Όπως έχουμε προαναφέρει, καθεμιά από τις εντολές αυτές εισάγεται με ένα αυτόματα παραγόμενο όνομα στον πίνακα συμβόλων ως μία διαδικασία χωρίς παραμέτρους. Αυτό όπως θα δούμε εξυπηρετεί στην επίτευξη της χρήσης κοινής μνήμης.

Σύμφωνα με το βάθος φωλιάσματος του υποπρογράμματος στο οποίο ανήκει ένα μη τοπικό τελούμενο και με το βάθος φωλιάσματος του υποπρογράμματος στο οποίο ζητείται αυτό, χρησιμοποποιούνται οι σύνδεσμοι προσπέλασης για την προσπέλαση του. Η τιμή του συνδέσμου προσπέλασης τοποθετείται στη θέση $\mathsf{ebp} + 4$ του ΕΔ και ενημερώνεται δυναμικά έτσι ώστε αν ένα υποπρόγραμμα p είναι άμεσα φωλιασμένο μέσα στο q, τότε ο σύνδεσμος από ένα ΕΔ του p να δείχνει στη βάση του πιο πρόσφατου ΕΔ του q.

4.6.3 Παραγωγή τελικού κώδικα

Για τη παραγωγή του τελικού κώδικα ακολουθήθηκε ως βάση το πρότυπο του βιβλίου αναφοράς [2] (ενότητες 9.5-9.6). Εδώ θα περιγράψουμε τις προσθήκες που χρειάστηκαν λόγω του χαρακτήρα ταυτοχρονισμού της CoPCL.

Για την υλοποίηση της δημιουργίας και συγχρονισμού διεργασιών χρησιμοποιήθηκε η βιβλιοθήκη συναρτήσεων *POSIX Threads* (ή Pthreads). Τα Pthreads αποτελούν ένα προτυποποιημένο μοντέλο για τη δημιουργία και διαχείριση *νημάτων*, όπως αποκαλούνται οι *«ελαφριές» διεργασίες* (light-weight processes). Το πρότυπο αυτό (POSIX 1003.1c) έχει υλοποιηθεί σε πολλές πλατφόρμες, από τις οποίες χρησιμοποιήσαμε την υλοποίηση στο λειτουργικό σύστημα Linux. Οι συναρτήσεις της βιβλιοθήκης που χρησιμοποιήθηκαν είναι οι εξής:

- pthread_create : Η συνάρτηση αυτή προκαλεί τη δημιουργία ενός νήματος. Οι κύριες παράμετροι που δέχεται είναι:
  - Η διαδικασία που πρόκειται να εκτελεστεί στο νέο νήμα που θα δημιουργηθεί.
  - Ένας δείκτης σε μία μεταβλητή για την αποθήκευση του κωδικού αριθμού του νήματος (thread_id).
  - Ένας δείκτης σε μία περιοχή μνήμης που πρόκειται να περαστεί ως παράμετρος στην διαδικασία που θα εκτελεστεί από το νήμα.

- pthread_join : Η συνάρτηση αυτή προκαλεί την καθυστέρηση του νήματος το οποίο την εκτελεί έως ότου τερματιστεί η εκτέλεση του νήματος με κωδικό αριθμό, αυτόν που της δίνεται ως παράμετρος. Η αποθήκευση του κωδικού αριθμού του νήματος που δημιουργήθηκε μέσω της pthread_create χρησιμεύει για την αναμονή ολοκλήρωσης της εκτέλεσής του.

- pthread_mutex_lock : Η συνάρτηση αυτή λαμβάνει ως παράμετρο μία μεταβλητή συνθήκης και μπλοκάρει πιθανές κλήσεις που έπονται και αναφέρονται στην ίδια μεταβλητή συνθήκης. Με τον τρόπο αυτό εξασφαλίζεται η μοναχική εκτέλεση του νήματος που την κάλεσε, σε αναφορά στην συγκεκριμένη μεταβλητή συνθήκης.

- pthread_mutex_lock : Η συνάρτηση αυτή απελευθερώνει τη μεταβλητή συνθήκης που δέχεται ως παράμετρο, με αποτέλεσμα άλλα νήματα εκτέλεσης να επιτρέπεται να εκτελέσουν την εργασία τους συγχρονισμένα.

- pthread_cond_wait : Η συνάρτηση αυτή δέχεται ως παράμετρο μία μεταβλητή συνθήκης και αδρανοποιεί το νήμα που την κάλεσε, τοποθετώντας το στην ουρά αναμονής που διαθέτει.

- pthread_cond_signal : Η συνάρτηση αυτή δέχεται ως παράμετρο μία μεταβλητή συνθήκης και αφυπνίζει το πρώτο νήμα που βρίσκεται αδρανοποιημένο σε κατάσταση αναμονής στην ουρά της συγκεκριμένης μεταβλητής.

Όπως έχει προαναφερθεί, κάθε εντολή μιας σύνθετης εντολής `cobegin...coend` μοντελοποιείται ως μία φωλιασμένη διαδικασία. Κατά τη δημιουργία του νήματος το οποίο θα την εκτελέσει, η ετικέτα της διαδικασίας αυτής (σε επίπεδο συμβολικής γλώσσας) αποτελεί την 1η από τις παραμέτρους της pthread_create, ενώ για την αποθήκευση του κωδικού αριθμού thread_id χρησιμοποιείται μία προσωρινή μεταβλητή. Επίσης, για την εξασφάλιση της χρήσης κοινών μεταβλητών της εμβέλειας στην οποία είναι φωλιασμένες οι διαδικασίες που εκτελούνται στα νέα νήματα δίνεται ο δείκτης πλαισίου του τρέχοντος ΕΔ ως τελευταία παράμετρος της pthread_create. Ο δείκτης αυτός θα περαστεί στο ΕΔ που θα δημιουργηθεί όταν θα δρομολογηθεί η κλήση της διαδικασίας από τη βιβλιοθήκη των Pthreads και θα παίξει το ρόλο του συνδέσμου προσπέλασης στο ΕΔ του.

Τέλος, πρέπει να σημειωθεί ότι οι μεταβλητές συνθήκης πριν τη χρήση τους πρέπει να είναι αρχικοποιημένες. Στην πλατφόρμα που χρησιμοποιήσαμε η αρχικοποίηση αυτή αντιστοιχεί σε μηδενισμό των περιεχομένων τους. Ο τρόπος που χρησιμοποιήσαμε ήταν να παράγουμε τελικό κώδικα για την αρχικοποίηση των τοπικών οντοτήτων στην αρχή της εκτέλεσης κάθε δομικής μονάδας.

Κεφάλαιο 5

Παραδείγματα

Στο κεφάλαιο αυτό δίνονται τρία παραδείγματα προγραμμάτων στη γλώσσα CoPCL. Για κάθε ένα δίνεται ο αρχικός κώδικας, ο ενδιάμεσος κώδικας, η μορφή των εγγραφημάτων δραστηριοποίησης των δομικών μονάδων, καθώς και ο τελικός κώδικας.

5.1 Hello World!

Το παρακάτω παράδειγμα είναι το απλούστερο πρόγραμμα στη γλώσσα CoPCL που παράγει κάποιο αποτέλεσμα ορατό στο χρήστη. Το πρόγραμμα αυτό τυπώνει απλώς ένα μήνυμα.

```
program hello;
begin
    writeString("Hello World!\n");
end.
```

Ο ενδιάμεσος κώδικας που παράγει ο μεταγλωττιστής της CoPCL είναι ο εξής:

```
1: unit, hello, -, -
2: par, "Hello World!\n", R, -
3: call, -, -, writeString
4: endu, hello, -, -
```

Το εγγράφημα δραστηριοποίησης του προγράμματος είναι εξαιρετικά απλό και δεν παρατίθεται. Ο τελικός κώδικας είναι ο εξής:

```
section    .data
section    .text
global    _start

_start:
    call   _hello_1
    mov    ebx, 0
    mov    eax, 1
    int    80h


@1:
_hello_1
    push   ebp
    mov    ebp, esp

@2:
    mov    eax, @@string_1
    push   eax

@3:
    sub    esp, 4
    push   ebp
    call   _writeString
    add    esp, 12

@4:
@hello_1:
    mov    esp, ebp
    pop    ebp
    ret

    extern    _writeString
    extern    pthread_create
    extern    pthread_join
    extern    pthread_mutex_lock
    extern    pthread_mutex_unlock
    extern    pthread_cond_wait
    extern    pthread_cond_signal

@@string_1    db    'Hello World!'
              db    10
              db    0
```

5.2 Παράδειγμα επίδειξης στοιχείων ταυτοχρονισμού

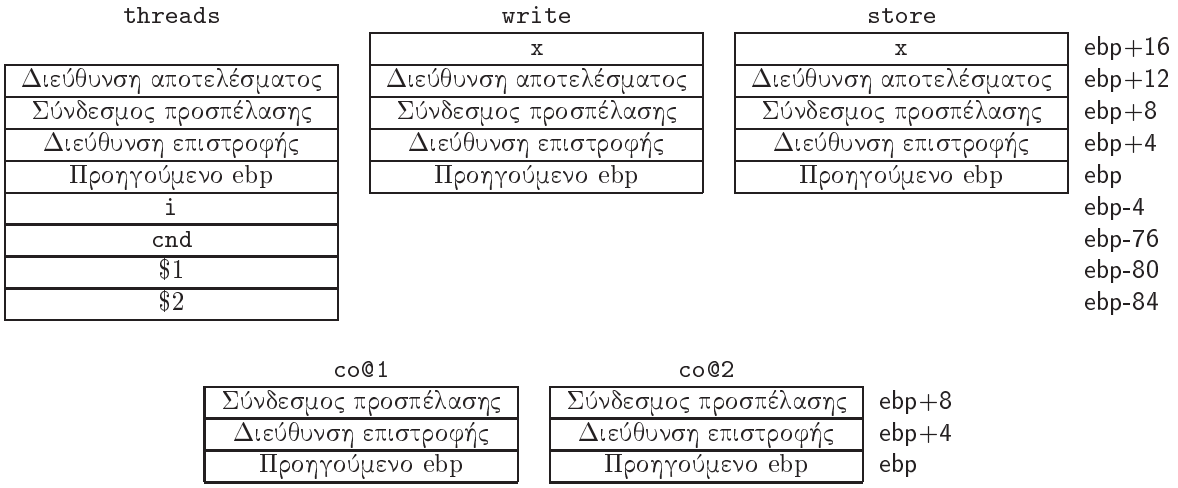
Το επόμενο πρόγραμμα (`threads.pcl`) δείχνει τη χρήση των στοιχείων ταυτοχρονισμού της γλώσσας CoPCL. Παρουσιάζεται η σύνθετη εντολή ταυτοχρονισμού `cobegin...coend`, καθώς και η τεχνική συγχρονισμού μέσω των εντολών `synchronized`, `wait`[1] και `signal`.

```
program threads;
    var i   : integer;
        cnd : condition;

    procedure write(x : integer);
    begin
        writeString("Value : ");
        writeInteger(x);
        writeChar('\n')
    end;

    procedure store(var x : integer);
    begin
        x := 113
    end;

begin
    i := 0;
    cobegin
        synchronized cnd when i>0 do
        begin
            write(i);
        end
        ||
        synchronized cnd do
        begin
            store(i);
            signal cnd
        end
    coend
end.
```

Ο ενδιάμεσος κώδικας που παράγει ο μεταγλωττιστής της CoPCL είναι ο εξής:

```
1: unit, write, -, -
2: par, "Value : ", R, -
3: call, -, -, writeString
4: par, x, V, -
5: call, -, -, writeInteger
6: par, '\n', V, -
7: call, -, -, writeChar
8: endu, write, -, -
9: unit, store, -, -
10: :=, 113, -, x
11: endu, store, -, -
12: unit, threads, -, -
13: :=, 0, -, i
14: create, -, co@1, {$1}
15: unit, co@1, -, -
16: lock, -, -, {cnd}
17: >, i, 0, 21
18: jump, -, -, 19
19: wait, -, -, {cnd}
20: jump, -, -, 17
21: par, i, V, -
22: call, -, -, write
23: unlock, -, -, {cnd}
24: endu, co@1, -, -
25: create, -, co@2, {$2}
26: unit, co@2, -, -
27: lock, -, -, {cnd}
28: par, i, R, -
29: call, -, -, store
30: signal, -, -, {cnd}
```

[1] Εδώ η `wait` εκτελείται μέσω της εντολής `synchronized` i `when` e `do` s, η οποία εμπεριέχει την εντολή βρόχου `while not` e `do wait` i.



Σχήμα 5.1: Εγγραφήματα δραστηριοποίησης για το πρόγραμμα `threads.pcl`.

```
31: unlock, -, -, {cnd}
32: endu, co@2, -, -
33: join, -, -, $1
34: join, -, -, $2
35: endu, threads, -, -
```

Τα εγγραφήματα δραστηριοποίησης για το παραπάνω πρόγραμμα φαίνονται στο σχήμα 5.1. Ο τελικός κώδικας δίνεται παρακάτω:

```
section    .data
section    .text
global    _start

_start:
    call   _threads_1
    mov    ebx, 0
    mov    eax, 1
    int    80h


@1:
_write_2
    push   ebp
    mov    ebp, esp

@2:
    mov    eax, @@string_1
    push   eax

@3:
    sub    esp, 4
    push   ebp
    call   _writeString
    add    esp, 12

@4:
    mov    eax, long [ebp+16]
    push   eax

@5:
    sub    esp, 4
    push   ebp
    call   _writeInteger
    add    esp, 12


@6:
    mov    al, 10
    sub    esp, 1
    mov    esi, esp
    mov    byte [esi], al

@7:
    sub    esp, 4
    push   ebp
    call   _writeChar
    add    esp, 9

@8:
@write_2:
    mov    esp, ebp
    pop    ebp
    ret

@9:
_store_3
    push   ebp
    mov    ebp, esp

@10:
    mov    eax, 113
    mov    esi, long [ebp+16]
    mov    long [esi], eax

@11:
@store_3:
    mov    esp, ebp
    pop    ebp
    ret
```

```
@15:
_co@1_4
    push   ebp
    mov    ebp, esp

@16:
    mov    esi, long [ebp+8]
    lea    eax, [esi-76]
    push   eax
    call   pthread_mutex_lock
    add    esp, 4

@17:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-4]
    mov    edx, 0
    cmp    eax, edx
    jg     @21

@18:
    jmp    @19

@19:
    mov    esi, long [ebp+8]
    lea    eax, [esi-76]
    push   eax
    mov    esi, long [ebp+8]
    lea    eax, [esi-76]
    add    eax, 24
    push   eax
    call   pthread_cond_wait
    add    esp, 8

@20:
    jmp    @17

@21:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-4]
    push   eax

@22:
    sub    esp, 4
    push   long [ebp+8]
    call   _write_2
    add    esp, 12

@23:
    mov    esi, long [ebp+8]
    lea    eax, [esi-76]
    push   eax
    call   pthread_mutex_unlock
    add    esp, 4

@24:
@co@1_4:
    mov    esp, ebp
    pop    ebp
    ret

@26:
_co@2_5
    push   ebp
    mov    ebp, esp

@27:
    mov    esi, long [ebp+8]
    lea    eax, [esi-76]
    push   eax
    call   pthread_mutex_lock
    add    esp, 4

@28:
    mov    esi, long [ebp+8]
    lea    esi, [esi-4]
    push   esi

@29:
    sub    esp, 4
    push   long [ebp+8]
    call   _store_3
    add    esp, 12

@30:
    mov    esi, long [ebp+8]
    lea    eax, [esi-76]
    add    eax, 24
    push   eax
    call   pthread_cond_signal
    add    esp, 4

@31:
    mov    esi, long [ebp+8]
    lea    eax, [esi-76]
    push   eax
    call   pthread_mutex_unlock
    add    esp, 4

@32:
@co@2_5:
    mov    esp, ebp
    pop    ebp
    ret

@12:
_threads_1
    push   ebp
    mov    ebp, esp
    sub    esp, 84
    mov    long [ebp-4], 0
    mov    long [ebp-8], 0
    mov    long [ebp-12], 0
    mov    long [ebp-16], 0
    mov    long [ebp-20], 0
    mov    long [ebp-24], 0
    mov    long [ebp-28], 0
    mov    long [ebp-32], 0
    mov    long [ebp-36], 0
    mov    long [ebp-40], 0
    mov    long [ebp-44], 0
    mov    long [ebp-48], 0
    mov    long [ebp-52], 0
    mov    long [ebp-56], 0
    mov    long [ebp-60], 0
    mov    long [ebp-64], 0
    mov    long [ebp-68], 0
```

```
    mov    long [ebp-72], 0
    mov    long [ebp-76], 0
    mov    long [ebp-80], 0
    mov    long [ebp-84], 0

@13:
    mov    eax, 0
    mov    long [ebp-4], eax

@14:
    mov    eax, ebp
    push   eax
    mov    eax, _co@1_4
    push   eax
    mov    eax, 0
    push   eax
    lea    esi, [ebp-80]
    push   esi
    call   pthread_create
    add    esp, 16

@25:
    mov    eax, ebp
    push   eax
    mov    eax, _co@2_5
    push   eax
    mov    eax, 0
    push   eax
    lea    esi, [ebp-84]
    push   esi
    call   pthread_create
    add    esp, 16
```

```
@33:
    mov    eax, 0
    push   eax
    mov    eax, long [ebp-80]
    push   eax
    call   pthread_join
    add    esp, 8

@34:
    mov    eax, 0
    push   eax
    mov    eax, long [ebp-84]
    push   eax
    call   pthread_join
    add    esp, 8

@35:
@threads_1:
    mov    esp, ebp
    pop    ebp
    ret

    extern    _writeString
    extern    _writeInteger
    extern    pthread_create
    extern    pthread_join
    extern    pthread_mutex_lock
    extern    pthread_mutex_unlock
    extern    pthread_cond_wait
    extern    pthread_cond_signal

@@string_1    db    'Value : '
              db    0
```

5.3 Παράδειγμα παραγωγού-καταναλωτή

Το πρόγραμμα που ακολουθεί προσομοιώνει ένα απλοποιημένο πρόβλημα τύπου «*παραγωγού-καταναλωτή*». Το ζητούμενο είναι δύο διεργασίες που εκτελούνται ταυτόχρονα να συγχρονιστούν κατάλληλα έτσι ώστε η μία να «παράγει» μια τιμή και να την αποθηκεύει σε μια μεταβλητή στην κοινή μνήμη, ενώ η άλλη να διαβάζει την τιμή από την κοινή μεταβλητή και να την «καταναλώνει».

Για τον κατάλληλο συγχρονισμό χρησιμοποιούνται δύο ακέραιες μεταβλητές με ονόματα **nonfull** και **nonempty**. Οι μεταβλητές αυτές παίρνουν τιμές ≥ 0. Όταν η τιμή τους είναι μηδενική, υποδηλώνουν ότι η συνθήκη που εκφράζει το ονομά τους είναι ψευδής, ενώ όσο αυξάνεται η τιμή τους σε τόσο μεγαλύτερο βαθμό υποδηλώνουν τη συνθήκη που εκφράζει το όνομά τους. Στην περίπτωσή μας, έχουμε απομονωτή μίας θέσης, οπότε οι δύο αυτές μεταβλητές έχουν την παρακάτω σημασία:

- Όταν **nonfull** = 0, ο απομονωτής είναι γεμάτος, ενώ όταν **nonfull** = 1, είναι άδειος.
- Όταν **nonempty** = 0, ο απομονωτής είναι άδειος, ενώ όταν **nonfull** = 1, είναι γεμάτος.

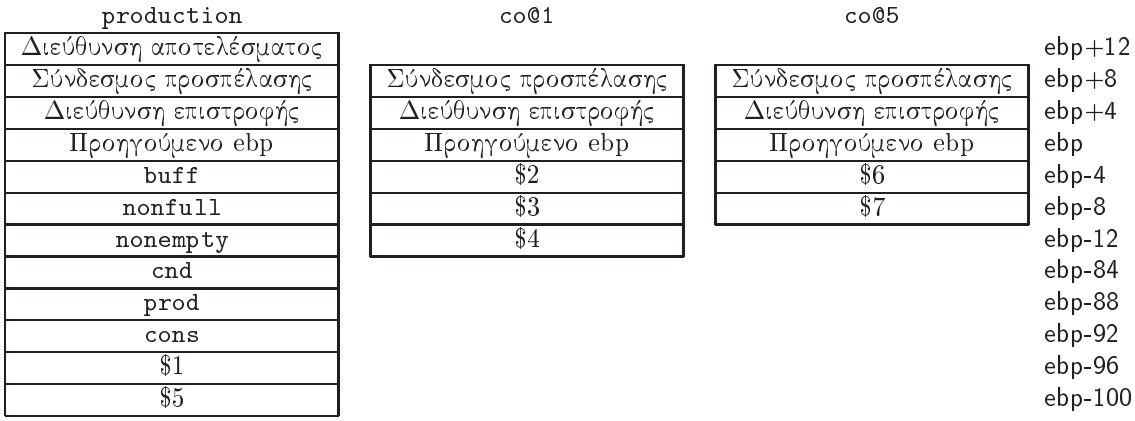
Το πρόγραμμα στη γλώσσα CoPCL που λύνει αυτό το πρόβλημα δίνεται παρακάτω.

```
program production;
    var buff : integer; (* buffer *)
        nonfull, nonempty: integer;
        cnd : condition;
        prod, cons : integer;
```

```
begin
    prod := 0;
    nonfull := 1; nonempty := 0;
    cobegin
        (* producer *)
        while true do
            synchronized cnd when nonfull > 0 do
            begin
                nonfull := nonfull - 1;

                prod := prod + 1; (* produce value *)
                sleep(2);
                writeString("Produced : "); writeInteger(prod); writeChar('\n');
                buff := prod;

                nonempty := nonempty + 1;
                signal cnd
            end
        ||
        (* consumer *)
        while true do
            synchronized cnd when nonempty > 0 do
            begin
                nonempty := nonempty - 1;

                cons := buff; (* consume value *)
                sleep(1);
                writeString("Consumed : "); writeInteger(cons);
                writeString("\n\n");

                nonfull := nonfull + 1;
                signal cnd
            end
    coend
end.
```

Ο ενδιάμεσος κώδικας που παράγει ο μεταγλωττιστής της CoPCL είναι ο εξής:

```
1: unit, production, -, -
2: :=, 0, -, prod
3: :=, 1, -, nonfull
4: :=, 0, -, nonempty
5: create, -, co@1, {$1}
6: unit, co@1, -, -
7: ifb, true, -, 9
8: jump, -, -, 32
9: lock, -, -, {cnd}
10: >, nonfull, 0, 14
11: jump, -, -, 12
12: wait, -, -, {cnd}
13: jump, -, -, 10
14: -, nonfull, 1, $2
15: :=, $2, -, nonfull
16: +, prod, 1, $3
17: :=, $3, -, prod
18: par, 2, V, -
19: call, -, -, sleep
20: par, "Produced : ", R, -
21: call, -, -, writeString
22: par, prod, V, -
23: call, -, -, writeInteger
24: par, '\n', V, -
25: call, -, -, writeChar
26: :=, prod, -, buff
27: +, nonempty, 1, $4
28: :=, $4, -, nonempty
29: signal, -, -, {cnd}
30: unlock, -, -, {cnd}
31: jump, -, -, 7
32: endu, co@1, -, -
33: create, -, co@5, {$5}
34: unit, co@5, -, -
35: ifb, true, -, 37
36: jump, -, -, 58
37: lock, -, -, {cnd}
38: >, nonempty, 0, 42
39: jump, -, -, 40
40: wait, -, -, {cnd}
```



Σχήμα 5.2: Εγγραφήματα δραστηριοποίησης για το πρόγραμμα *παραγωγού-καταναλωτή*.

```
41: jump, -, -, 38
42: -, nonempty, 1, $6
43: :=, $6, -, nonempty
44: :=, buff, -, cons
45: par, 1, V, -
46: call, -, -, sleep
47: par, "Consumed : ", R, -
48: call, -, -, writeString
49: par, cons, V, -
50: call, -, -, writeInteger
51: par, "\n\n", R, -
52: call, -, -, writeString
53: +, nonfull, 1, $7
54: :=, $7, -, nonfull
55: signal, -, -, {cnd}
56: unlock, -, -, {cnd}
57: jump, -, -, 35
58: endu, co@5, -, -
59: join, -, -, $1
60: join, -, -, $5
61: endu, production, -, -
```

Τα εγγραφήματα δραστηριοποίησης φαίνονται στο σχήμα 5.2.
Ο τελικός κώδικας δίνεται παρακάτω:

```
section    .data
section    .text
global    _start

_start:
    call   _production_1
    mov    ebx, 0
    mov    eax, 1
    int    80h


@6:
_co@1_2
    push   ebp
    mov    ebp, esp
    sub    esp, 12
    mov    long [ebp-4], 0
    mov    long [ebp-8], 0
    mov    long [ebp-12], 0

@7:
    mov    al, 1
    or     al, al
    jnz    @9

@8:
    jmp    @32

@9:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    push   eax
    call   pthread_mutex_lock
    add    esp, 4

@10:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-8]
    mov    edx, 0
    cmp    eax, edx
    jg     @14

@11:
    jmp    @12

@12:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    push   eax
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    add    eax, 24
    push   eax
    call   pthread_cond_wait
    add    esp, 8

@13:
    jmp    @10
```

```
@14:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-8]
    mov    edx, 1
    sub    eax, edx
    mov    long [ebp-4], eax

@15:
    mov    eax, long [ebp-4]
    mov    esi, long [ebp+8]
    mov    long [esi-8], eax

@16:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-88]
    mov    edx, 1
    add    eax, edx
    mov    long [ebp-8], eax

@17:
    mov    eax, long [ebp-8]
    mov    esi, long [ebp+8]
    mov    long [esi-88], eax

@18:
    mov    eax, 2
    push   eax

@19:
    sub    esp, 4
    push   ebp
    call   _sleep
    add    esp, 12

@20:
    mov    eax, @@string_1
    push   eax

@21:
    sub    esp, 4
    push   ebp
    call   _writeString
    add    esp, 12

@22:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-88]
    push   eax

@23:
    sub    esp, 4
    push   ebp
    call   _writeInteger
    add    esp, 12

@24:
    mov    al, 10
    sub    esp, 1
    mov    esi, esp
    mov    byte [esi], al

@25:
    sub    esp, 4
    push   ebp
    call   _writeChar
    add    esp, 9

@26:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-88]
    mov    esi, long [ebp+8]
    mov    long [esi-4], eax

@27:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-12]
    mov    edx, 1
    add    eax, edx
    mov    long [ebp-12], eax

@28:
    mov    eax, long [ebp-12]
    mov    esi, long [ebp+8]
    mov    long [esi-12], eax

@29:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    add    eax, 24
    push   eax
    call   pthread_cond_signal
    add    esp, 4

@30:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    push   eax
    call   pthread_mutex_unlock
    add    esp, 4

@31:
    jmp    @7

@32:
@co@1_2:
    mov    esp, ebp
    pop    ebp
    ret

@34:
_co@5_3
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    long [ebp-4], 0
    mov    long [ebp-8], 0

@35:
    mov    al, 1
    or     al, al
    jnz    @37

@36:
    jmp    @58
```

```
@37:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    push   eax
    call   pthread_mutex_lock
    add    esp, 4

@38:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-12]
    mov    edx, 0
    cmp    eax, edx
    jg     @42

@39:
    jmp    @40

@40:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    push   eax
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    add    eax, 24
    push   eax
    call   pthread_cond_wait
    add    esp, 8

@41:
    jmp    @38

@42:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-12]
    mov    edx, 1
    sub    eax, edx
    mov    long [ebp-4], eax

@43:
    mov    eax, long [ebp-4]
    mov    esi, long [ebp+8]
    mov    long [esi-12], eax

@44:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-4]
    mov    esi, long [ebp+8]
    mov    long [esi-92], eax

@45:
    mov    eax, 1
    push   eax

@46:
    sub    esp, 4
    push   ebp
    call   _sleep
    add    esp, 12

@47:
    mov    eax, @@string_2
    push   eax

@48:
    sub    esp, 4
    push   ebp
    call   _writeString
    add    esp, 12

@49:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-92]
    push   eax

@50:
    sub    esp, 4
    push   ebp
    call   _writeInteger
    add    esp, 12

@51:
    mov    eax, @@string_3
    push   eax

@52:
    sub    esp, 4
    push   ebp
    call   _writeString
    add    esp, 12

@53:
    mov    esi, long [ebp+8]
    mov    eax, long [esi-8]
    mov    edx, 1
    add    eax, edx
    mov    long [ebp-8], eax

@54:
    mov    eax, long [ebp-8]
    mov    esi, long [ebp+8]
    mov    long [esi-8], eax

@55:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    add    eax, 24
    push   eax
    call   pthread_cond_signal
    add    esp, 4

@56:
    mov    esi, long [ebp+8]
    lea    eax, [esi-84]
    push   eax
    call   pthread_mutex_unlock
    add    esp, 4

@57:
    jmp    @35

@58:
@co@5_3:
    mov    esp, ebp
    pop    ebp
    ret
```

```
@1:
_production_1
    push   ebp
    mov    ebp, esp
    sub    esp, 100
    mov    long [ebp-4], 0
    mov    long [ebp-8], 0
    mov    long [ebp-12], 0
    mov    long [ebp-16], 0
    mov    long [ebp-20], 0
    mov    long [ebp-24], 0
    mov    long [ebp-28], 0
    mov    long [ebp-32], 0
    mov    long [ebp-36], 0
    mov    long [ebp-40], 0
    mov    long [ebp-44], 0
    mov    long [ebp-48], 0
    mov    long [ebp-52], 0
    mov    long [ebp-56], 0
    mov    long [ebp-60], 0
    mov    long [ebp-64], 0
    mov    long [ebp-68], 0
    mov    long [ebp-72], 0
    mov    long [ebp-76], 0
    mov    long [ebp-80], 0
    mov    long [ebp-84], 0
    mov    long [ebp-88], 0
    mov    long [ebp-92], 0
    mov    long [ebp-96], 0
    mov    long [ebp-100], 0

@2:
    mov    eax, 0
    mov    long [ebp-88], eax

@3:
    mov    eax, 1
    mov    long [ebp-8], eax

@4:
    mov    eax, 0
    mov    long [ebp-12], eax

@5:
    mov    eax, ebp
    push   eax
    mov    eax, _co@1_2
    push   eax
    mov    eax, 0
    push   eax
    lea    esi, [ebp-96]
    push   esi
    call   pthread_create
    add    esp, 16
```

```
@33:
    mov    eax, ebp
    push   eax
    mov    eax, _co@5_3
    push   eax
    mov    eax, 0
    push   eax
    lea    esi, [ebp-100]
    push   esi
    call   pthread_create
    add    esp, 16

@59:
    mov    eax, 0
    push   eax
    mov    eax, long [ebp-96]
    push   eax
    call   pthread_join
    add    esp, 8

@60:
    mov    eax, 0
    push   eax
    mov    eax, long [ebp-100]
    push   eax
    call   pthread_join
    add    esp, 8

@61:
@production_1:
    mov    esp, ebp
    pop    ebp
    ret

    extern    _writeChar
    extern    _writeString
    extern    _writeInteger
    extern    _sleep
    extern    pthread_create
    extern    pthread_join
    extern    pthread_mutex_lock
    extern    pthread_mutex_unlock
    extern    pthread_cond_wait
    extern    pthread_cond_signal

@@string_1    db    'Produced : '
              db    0

@@string_2    db    'Consumed : '
              db    0

@@string_3    db    10
              db    10
              db    0
```

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

Η συνεισφορά της εργασίας αυτής συνοψίζεται στα ακόλουθα:

- Μελετήθηκαν τα στοιχεία ταυτοχρονισμού που συναντώνται σε διάφορες γλώσσες προγραμματισμού. Έμφαση δόθηκε στις δομές ελέγχου διεργασιών καθώς και στις τεχνικές συγχρονισμού τους. Από τα στοιχεία αυτά επιλέχθηκαν τα πλέον κατάλληλα για την επέκταση μιας γλώσσας προστατκτικού χαρακτήρα και προέκυψε η γραμματική της γλώσσας CoPCL που είδαμε στην ενότητα 3.7.

- Υλοποιήθηκε ένας μεταγλωττιστής της γλώσσας CoPCL. Η υλοποίηση του μεταγλωττιστή περιέλαβε τα στάδια του λεκτικού, συντακτικού και σημασιολογικού αναλυτή, καθώς και του γεννήτορα ενδιάμεσου και τελικού κώδικα. Ως τελική γλώσσα του μεταγλωττιστή χρησιμοποιήθηκε συμβολική γλώσσα για 32-bit επεξεργαστές της Intel, ενώ για την υλοποίηση των πρόσθετων στοιχείων ταυτοχρονισμού χρησιμοποιήθηκε η βιβλιοθήκη συναρτήσεων POSIX threads, σε πλατοφόρμα λειτουργικού συτήματος Linux. Ως γλώσσα υλοποίησης του μεταγλωττιστή χρησιμοποιήθηκε η C++.

Ο μεταγλωττιστής που προέκυψε μπορεί να χρησιμοποιηθεί για μεταγλώττιση προγραμμάτων σε CoPCL. Αποτελεί χρήσιμο εκπαιδευτικό εργαλείο για κάποιον που επιθυμεί να γνωρίσει προστακτικά στοιχεία ταυτοχρονισμού, τα οποία αποτελούν και τη βάση όλων των πιο πολύπλοκων δομών που μελετήθηκαν στο Κεφάλαιο 2.

6.2 Μελλοντική έρευνα

Όπως είναι φυσικό, υπάρχουν πολλά περιθώρια επέκτασης της γλώσσας CoPCL και του μεταγλωττιστή με νέα στοιχεία προς την κατεύθυνση της αύξησης της λειτουργικότητας και πρακτικότητάς του. Ένα από τα σημαντικότερα στοχεία είναι η προσθήκη μηχανισμών περάσματος μηνυμάτων. Με τον τρόπο αυτό η γλώσσα CoPCL μπορεί να αξιοποιηθεί και για παράλληλες αρχιτεκτονικές με κατανεμημένη μνήμη.

Τέλος, θα μπορούσαμε να σημειώσουμε πως η κατασκευή και η βελτιστοποίηση τεχνικών κατασκευής μεταγλωττιστών αποτελεί και θα αποτελέσει στο μέλλον ένα τομέα πλούσιας ερευνητικής άνθησης. Οι απαιτήσεις για πολύπλοκες και κατανεμημένες εφαρμογές συνεχώς αυξάνονται, ενώ η κάμψη στην αύξηση της ταχύτητας των μικροεπεξεργαστών που παρατηρείται και η συνεπαγόμενη ανάπτυξη παράλληλων αρχιτεκτονικών, οδηγούν στην εξέλιξη των γλωσσών προγραμματισμού που υποστηρίζουν τεχνικές ταυτοχρονισμού. Η σχεδίαση αυτών των γλωσσών για την όσο το δυνατό αποτελεσματικότερη χρήση τους από τον προγρμματιστή, καθώς και η βελτιστοποίηση της απόδοσης των μεταγλωττιστών τους, αποτελούν θέματα μείζονος σημασίας για την εξέλιξη της πορείας του προγραμματισμού.

Βιβλιογραφία

[1] Γ. Παπακωνσταντίνου, Θ. Θεοχάρης, and Π. Τσανάκας. *Συστήματα Παράλληλης Επεξεργασίας*. Εκδόσεις Συμμετρία, 1994.

[2] Ν. Σ. Παπασπύρου and Ε. Σ. Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002.

[3] Α.-Γ. Ν. Σταφυλοπάτης. *Γλώσσες Προγραμματισμού*. Έκδοση Εθνικού Μετσόβιου Πολυτεχνείου, 2002. Σημειώσεις του αντίστοιχου μαθήματος.

[4] A. V. Aho, R. Sethi, and J. P. Ulman. *Compilers: Principles Techniques and Tools*. Addison-Wesley, 1986.

[5] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, 1983.

[6] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[7] F. L. Bauer and J. Eickel. *Compiler Construction, An Advanced Course*. Springer-Verlag, 1984.

[8] Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1982.

[9] R. Bornat and F. H. Sumner. *Understanding and Writing Compilers: A Do It Yourself Guide*. MacMillan Education Ltd., 3rd edition, 1989.

[10] C. N. Fischer and Jr. R. J. LeBlanc. *Crafting A Compiler with C*. The Benjamin/Cumming Pub. Comp., Inc., 1991.

[11] A. I. Holub. *Compiler Design in C*. Prentice-Hall International, Inc., 1990.

[12] R. Hunter. *The Design and Construction of Compilers*. Prentice-Hall International, Inc., 1981.

[13] T. Pittman and J. Peters. *The Art of Compiler Design, Theory and Practice*. Prentice-Hall International, Inc., 1992.

[14] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[15] Bradford Nichols, Dick Buttlar, and Jacqeline Proulx Farrell. *Pthreads Programming*. O'Reilly, 1996.

[16] R. W. Sebesta. *Concepts of Programming Languages, Fourth Edition*. Addison-Wesley, 1999.

[17] Ravi Sethi. *Programming Languages: Concepts and Constructs, 2nd Edition*. Addison-Wesley, Reading, MA, 1996.

[18] W. M. Waite and G. Goos. *Compiler Construction*. Springer Verlag, 1984.