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

Υλοποίηση Μεταγλωττιστή για μια Γλώσσα Αντικειμενοστρεφούς Προγραμματισμού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΡΙΣΤΟΦΑΝΗΣ Τ. ΤΖΙΑΣΙΟΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2005

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

Υλοποίηση Μεταγλωττιστή για μια Γλώσσα Αντικειμενοστρεφούς Προγραμματισμού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΡΙΣΤΟΦΑΝΗΣ Τ. ΤΖΙΑΣΙΟΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18 Ιουλίου 2005.

....	....	....
Νικόλαος Παπασπύρου Λέκτορας Ε.Μ.Π.	Τιμολέων Σελλής Καθηγητής Ε.Μ.Π.	Ανδρέας-Γεώργιος Σταφυλοπάτης Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2005

....

Αριστοφάνης Τ. Τζιάσιος

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

Copyright © Αριστοφάνης Τ. Τζιάσιος, 2005.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σκοπός της εργασίας είναι η κατασκευή ενός μεταγλωττιστή για μια γλώσσα αντικειμενοστρεφούς προγραμματισμού που ακολουθεί το πρότυπο της Java και είναι βασισμένη στο προστακτικό μοντέλο προγραμματισμού. Η τελική γλώσσα είναι η συμβολική γλώσσα (assembly) των επεξεργαστών x86 της Intel.

Ο αντικειμενοστρεφής προγραμματισμός επιτρέπει την εύκολη αποτύπωση ενός προβλήματος μέσω της περιγραφής των οντοτήτων που συμμετέχουν στο πρόβλημα και των αλληλεπιδράσεών τους. Κάθε πρόγραμμα νοείται ως ένα σύνολο από αλληλεπιδρώντα αντικείμενα, που μπορούν να είναι οργανωμένα σε ιεραρχίες κλάσεων μέσω κληρονομικότητας. Οι κλάσεις είναι πρότυπα που χρησιμοποιούνται για τη δημιουργία ενός αντικειμένου και μπορούν να περιέχουν δύο τύπους πληροφοριών: ιδιότητες και συμπεριφορά. Οι ιδιότητες είναι τα δεδομένα που διαφοροποιούν μια κλάση αντικειμένων από μια άλλη και εκφράζονται με τον ορισμό μεταβλητών. Αντίστοιχα, η συμπεριφορά μιας κλάσης αναφέρεται στις πράξεις που μπορεί να εκτελεστούν από και πάνω στα αντικείμενά της και υλοποιείται με τη χρήση μεθόδων.

Κατά την υλοποίηση του μεταγλωττιστή ακολουθήθηκαν τα στάδια της λεκτικής, συντακτικής, σημασιολογικής ανάλυσης, παραγωγής ενδιάμεσου και τελικού κώδικα. Δόθηκε έμφαση στην ενσωμάτωση των περισσότερων χαρακτηριστικών που υποστηρίζονται από τις σύγχρονες γλώσσες αντικειμενοστρεφούς προγραμματισμού, όπως είναι η απλή κληρονομικότητα, οι κατασκευαστές και καταστροφείς, ο πολυμορφισμός (δυναμικό δέσιμο μεθόδων), ο έλεγχος υποτύπων, οι μετατροπές αντικειμένων και οι διαπροσωπείες. Για την υποστήριξή τους χρησιμοποιήθηκαν οι καθιερωμένες τεχνικές, όπως ο πίνακας ανταπόκρισης μεθόδων και ο περιγραφέας κλάσης.

Λέξεις κλειδιά

Υλοποίηση γλωσσών προγραμματισμού, μεταγλωττιστές, αντικειμενοστρεφής προγραμματισμός.

Abstract

The purpose of this diploma dissertation is the implementation of a compiler for an object-oriented programming language, following the paradigm of Java and based on the imperative programming model. The final language is assembly for Intel's series of x86 processors.

Object-oriented programming allows an easy translation of a problem's solution, by describing the entities that participate in the problem and the way in which they interact with each other. A program is a set of interacting objects, organized in class hierarchies where derived classes inherit from their parent class. Classes are prototypes that are used for the creation of objects. They may contain two types of information: properties and behaviour. Properties are data that differentiate one class from another: they are expressed as variable declarations. On the other hand, the behaviour of a class refers to the actions that can be performed by and on its objects and is implemented by using methods.

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 object-oriented programming languages, such as simple inheritance, constructors and destructors, polymorphism (dynamic binding of methods), run-time type inference, object casts and intefaces. To implement these features, the standard techniques have been used, such as a method dispatch table and a class descriptor.

Key words

Programming language implementation, compilers, object-oriented programming.

Ευχαριστίες

Θέλω να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου, κ. Νίκο Παπασπύρου για το ενδιαφέρον που έδειξε και για τον χρόνο που αφιέρωσε σε αυτή την εργασία. Θέλω επίσης να ευχαριστήσω τους γονείς μου και τα αδέλφια μου για το καλό οικογενειακό κλίμα και τη συμπαράσταση τους κατά τη διάρκεια των σπουδών μου.

Αριστοφάνης Τ. Τζιάσιος,

Αθήνα, 17η Ιουλίου 2005.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-4-05, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2005.

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

Περιεχόμενα

Περίληψη . . . 5

Abstract . . . 7

Ευχαριστίες . . . 9

Περιεχόμενα . . . 11

Πίνακες . . . 13

Σχήματα . . . 15

1. Εισαγωγή . . . 17
1.1 Σκοπός . . . 17
1.2 Σύνοψη της εργασίας . . . 17

2. Αντικειμενοστρεφής προγραμματισμός . . . 19
2.1 Αντικείμενα, κλάσεις, μέθοδοι και μεταβλητές . . . 19
2.2 Απλή κληρονομικότητα . . . 21
2.3 Πολυμορφισμός υποτύπων . . . 24
2.4 Πολλαπλή κληρονομικότητα . . . 24
2.5 Έλεγχος υποτύπων και μετατροπές . . . 25
2.6 Διαπροσωπείες . . . 26

3. Η γλώσσα Ooedsger . . . 27
3.1 Λεκτικές μονάδες . . . 27
3.2 Τύποι δεδομένων . . . 29
3.3 Δομή του προγράμματος . . . 30
3.3.1 Κλάσεις και διαπροσωπείες . . . 30
3.3.2 Μεταβλητές . . . 31
3.3.3 Μέθοδοι . . . 31
3.4 Εκφράσεις . . . 32
3.4.1 L-values . . . 33
3.4.2 Σταθερές . . . 33
3.4.3 Τελεστές . . . 34
3.4.4 Κλήση μεθόδων . . . 36
3.4.5 Κατασκευαστές, καταστροφείς, δυναμική διαχείριση μνήμης . . . 37
3.4.6 Δεσμευμένα αναγνωριστικά `this` και `super` . . . 38
3.4.7 Αλλαγές τύπων . . . 38
3.5 Εντολές . . . 39
3.6 Οδηγίες προς το μεταγλωττιστή . . . 40
3.7 Βιβλιοθήκη έτοιμων συναρτήσεων . . . 40
3.7.1 Είσοδος και έξοδος . . . 40

3.7.2 Μαθηματικές συναρτήσεις 41
3.7.3 Συναρτήσεις μετατροπής 41
3.7.4 Συναρτήσεις διαχείρισης συμβολοσειρών 41
3.8 Πλήρης γραμματική της Ooedsger 42

4. Μεταγλώττιση της γλώσσας Ooedsger 43
4.1 Λεκτική ανάλυση 43
4.2 Συντακτική ανάλυση 44
4.3 Πίνακας συμβόλων 46
4.4 Σημασιολογική ανάλυση 47
4.5 Ενδιάμεσος κώδικας 48
4.5.1 Τελούμενα 50
4.5.2 Τελεστές 51
4.6 Τελικός κώδικας 54
4.6.1 Εγγράφημα δραστηριοποίησης και οργάνωση της μνήμης 54
4.6.2 Αντικείμενα και μέθοδοι 57
4.6.3 Κατασκευαστές και καταστροφείς 58
4.6.4 Απλή κληρονομικότητα 58
4.6.5 Πολυμορφισμός 59
4.6.6 Έλεγχος υποτύπων και μετατροπές 61
4.6.7 Διαπροσωπείες 63

5. Παραδείγματα 67
5.1 Πες γεια! 67
5.2 Παράδειγμα επίδειξης κατασκευαστών και καταστροφέων 68
5.3 Οι πύργοι του Hanoi 74
5.4 Εκτενές παράδειγμα επίδειξης αντικειμενοστρεφών χαρακτηριστικών 84

6. Συμπεράσματα και επεκτάσεις 97

Βιβλιογραφία 99

Πίνακες

3.1 Ακολουθίες διαφυγής (escape sequences). 28
3.2 Προτεραιότητα και προσεταιριστικότητα των τελεστών της Ooedsger. 36

Σχήματα

4.1 Παράδειγμα κατασκευής συντακτικού δέντρου. 45
4.2 Μια γραμματική για τη γλώσσα των τετράδων. 52
4.3 Πληροφορίες που αποθηκεύονται σε ένα ΕΔ. 55
4.4 Μια διαρρύθμιση της μνήμης που είναι διαθέσιμη για την εκτέλεση ενός προγράμματος. 56
4.5 Παράσταση στη μνήμη των αντικειμένων της κλάσης `Shape`. 57
4.6 Παράσταση στη μνήμη των αντικειμένων των κλάσεων `Circle` και `Line` που παράγονται από την κλάση `Shape`. 59
4.7 Παράσταση στη μνήμη των αντικειμένων των κλάσεων `Shape`, `Circle` και `Line` που υποστηρίζει δυναμικό δέσιμο μεθόδων. 60
4.8 Παράσταση στη μνήμη των αντικειμένων των κλάσεων `Circle` και `Line` μετά την προσθήκη της αφηρημένης μεθόδου `length`. 62
4.9 Παράσταση στη μνήμη που επιτρέπει τον έλεγχο υποτύπων. 63
4.10 Παράσταση στη μνήμη κλάσης που υλοποιεί διαπροσωπεία. 64
4.11 Παράσταση στη μνήμη αντικειμένου κλάσης και του αντίστοιχου δείκτη σε διαπροσωπεία. 64

5.1 Εγγραφήματα δραστηριοποίησης για το πρόγραμμα `obj.oed`. 70
5.2 Παράσταση στη μνήμη των κλάσεων για το πρόγραμμα `obj.oed`. 71
5.3 Οι πύργοι του Hanoi. 74
5.4 Εγγραφήματα δραστηριοποίησης για τους Πύργους του Hanoi. 77
5.5 Παράσταση στη μνήμη των κλάσεων για τους Πύργους του Hanoi. 78
5.6 Εγγραφήματα δραστηριοποίησης για το πρόγραμμα `extended.oed`. 87
5.7 Παράσταση στη μνήμη των κλάσεων για το πρόγραμμα `extended.oed`. 88

Κεφάλαιο 1

Εισαγωγή

Ο αντικειμενοστρεφής προγραμματισμός (object-oriented programming) είναι μια μεθοδολογία ανάπτυξης λογισμικού, σύμφωνα με την οποία ένα πρόγραμμα γίνεται αντιληπτό σαν μια ομάδα αντικειμένων που συνεργάζονται μεταξύ τους. Τα αντικείμενα δημιουργούνται χρησιμοποιώντας πρότυπα που καλούνται κλάσεις, και περιέχουν δεδομένα και τον κώδικα που απαιτείται για χρήση αυτών των δεδομένων (συμπεριφορά). Όλα τα αντικείμενα μιας κλάσης μοιράζονται έναν κοινό ορισμό, αλλά διαφέρουν εν γένει στις τιμές που δίνονται στα δεδομένα τους. Βλέπουμε ότι η αντικειμενοστρεφής προσέγγιση μιμείται τον τρόπο με τον οποίο συλλέγονται και συνεργάζονται αντικείμενα στο φυσικό κόσμο. Τα αντικείμενα επίσης ως υποστάσεις μιας κλάσης ευνοούν την επαναχρησιμοποίηση και οικονομία κώδικα, αφού η κάθε κλάση ορίζεται μια φορά και στη συνέχεια μπορούν να δημιουργηθούν πολλά και διαφορετικά αντικείμενα αυτής. Παράλληλα, με τη χρήση του μηχανισμού της κληρονομικότητας οι κλάσεις μπορούν να διαταχθουν σε μια ιεραρχία, μέσα στην οποία κάθε υποκλάση κληρονομεί τα χαρακτηριστικά της (κάθε) υπερκλάσης της. Πρόκειται για ορισμένα μόνο στοιχεία που δείχνουν τη χρησιμότητα και τη λειτουργικότητα του αντικειμενοστρεφούς μοντέλου.

1.1 Σκοπός

Σκοπός της εργασίας είναι η κατασκευή ενός μεταγλωττιστή για μια γλώσσα αντικειμενοστρεφούς προγραμματισμού που βασίζεται στο προστακτικό μοντέλο και στο πρότυπο της Java. Η γλώσσα ονομάστηκε Ooedsger, που σημαίνει Object-oriented Edsger, δηλαδή αντικειμενοστρεφής Edsger. Edsger ήταν το όνομα της γλώσσας που είχε ανατεθεί για κατασκευή μεταγλωττιστή στο αντίστοιχο μάθημα Μεταγλωττιστές του 8ου εξαμήνου, κατά το ακαδημαϊκό έτος 2003-2004 και η περιγραφή της οποίας είναι διαθέσιμη στον ιστοτόπο:

`http://courses.softlab.ntua.gr/compilers/2004a/edsger2004.pdf`

Η γλώσσα εκείνη είχε ονομαστεί έτσι από το όνομα του μεγάλου Δανού επιστήμονα Πληροφορικής, *Edsger Dijkstra* (Rotterdam, 11 Μαΐου, 1930 – Nuenen, 6 Αυγούστου, 2002), γνωστού και από τον *αλγόριθμο του συντομότερου μονοπατιού* (shortest path-algorithm, Dijkstra's algorithm). Η γλώσσα Ooedsger λοιπόν, όπως προκύπτει και από το όνομά της, επεκτείνει τη γλώσσα Edsger με αντικειμενοστρεφή χαρακτηριστικά.

1.2 Σύνοψη της εργασίας

Στο πλαίσιο της υλοποίησης του μεταγλωττιστή για τη γλώσσα Ooedsger 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 αντικειμενοστρεφούς προγραμματισμού.

Κεφάλαιο 3. Ορισμός και περιγραφή της γλώσσας Ooedsger.

Κεφάλαιο 4. Περιγραφή της υλοποίησης και μεταγλώττισης της Ooedsger.

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

Κεφάλαιο 6. Συμπεράσματα και προτάσεις μελλοντικής επέκτασης του μεταγλωττιστή.

Κεφάλαιο 2

Αντικειμενοστρεφής προγραμματισμός

Η σχολή του *αντικειμενοστρεφούς προγραμματισμού* (object-oriented programming) διδάσκει την ανάπτυξη λογισμικού ως ενός συνόλου από αλληλεπιδρώντα *αντικειμένα* (objects). Διαφοροποιείται σε σχέση με το *διαδικαστικό μοντέλο προγραμματισμού*, όπου ένα πρόγραμμα αντιμετωπίζεται σαν μια σειρά εντολών που εκτελούνται σε αλληλουχία. Βασικό χαρακτηριστικό της αντικειμενοστρεφούς προσέγγισης είναι η αφαιρετική από κοινού αντιμετώπιση της κατάστασης και της συμπεριφοράς των αντικειμένων, το οποίο στην ορολογία αυτής της σχολής ονομάζεται συχνά *κελυφοποίηση* ή *ενθυλάκωση* (encapsulation). Έτσι, κάθε αντικείμενο είναι ένα ξεχωριστό τμήμα ενός προγράμματος, που αλληλεπιδρά με τα άλλα τμήματα με συγκεκριμένους, ελεγχόμενους τρόπους.

Εκτός από τα αντικείμενα, κύρια θέση στις περισσότερες γλώσσες αντικειμενοστρεφούς προγραμματισμού κατέχει η έννοια της *κλάσης* (class), που ορίζει μια κατηγορία αντικειμένων με την ίδια συμπεριφορά: αντικείμενα που προέρχονται από την ίδια κλάση είναι δυνατόν να διαφέρουν μόνο ως προς την κατάσταση. Εξέχουσα θέση επίσης έχουν οι έννοιες της *κληρονομικότητας* (inheritance) και του *πολυμορφισμού υποτύπων* (subtype polymorphism). Με τη δημιουργία κλάσεων και ιεραρχιών από κλάσεις, η σχολή του αντικειμενοστρεφούς προγραμματισμού διδάσκει μια νέα φιλοσοφία στην ανάπτυξη λογισμικού που ευνοεί την επαναχρησιμοποίηση (reusability). Οι δημοφιλέστερες σήμερα γλώσσες που ανήκουν σε αυτή τη σχολή είναι η C++ και η Java. Και οι δύο βασίζονται στο προστακτικό μοντέλο προγραμματισμού και έχουν έντονες επιρροές από τη γλώσσα C.

2.1 Αντικείμενα, κλάσεις, μέθοδοι και μεταβλητές

Ο αντικειμενοστρεφής προγραμματισμός μοντελοποιείται με βάση την παρατήρηση ότι στον πραγματικό κόσμο, τα αντικείμενα αποτελούνται από πολλά μικρότερα αντικείμενα. Ένα αντικείμενο είναι μια ενθυλακωμένη συλλογή δεδομένων και πράξεων. Ο τύπος των αντικειμένων διαφέρει από αυτόν των εγγραφών (records, structures) ως προς το ότι ο πρώτος εκτός από πεδία δεδομένων (data fields), που προσδιορίζουν την κατάσταση ενός αντικειμένου, διαθέτει και *μεθόδους* (methods), που υλοποιούν τη συμπεριφορά του. Οι μέθοδοι είναι ουσιαστικά διαδικασίες ή συναρτήσεις που όμως εφαρμόζονται και αφορούν σε μια συγκεκριμένη εμφάνιση ενός αντικειμένου ως απόκριση σε κάποιο *μήνυμα* που λαμβάνει από κάποιο άλλο αντικείμενο για να παραγάγει κάποιο αποτέλεσμα. Οι πράξεις που εκτελούνται για το σκοπό αυτό, και αποτελούν το σώμα της μεθόδου, καθορίζονται μόνο από το αντικείμενο που δέχεται το μήνυμα. Το μήνυμα ισοδυναμεί στην πραγματικότητα με την κλήση ενός υποπρογράμματος. Όπως είναι ορατό, υπάρχει αντιστοιχία ανάμεσα στα μηνύματα και τις μεθόδους ενός αντικειμένου. Το ίδιο μήνυμα μπορεί να στέλνεται σε πολλά αντικείμενα προκαλώντας την εκτέλεση διαφορετικής μεθόδου για διάφορους τύπους αντικειμένων.

Η κλάση είναι ένα πρότυπο που χρησιμοποιείται για τη δημιουργία ενός αντικειμένου. Περιλαμβάνει την περιγραφή των δεδομένων και των μεθόδων που περιέχονται σε κάθε αντικείμενο της κλάσης. Περιέχει, δηλαδή, όλα τα χαρακτηριστικά ενός συγκεκριμένου συνόλου αντικειμένων. Πρόκειται για αφαιρετικό τύπο δεδομένων που αποτελεί επέκταση της συμβατικής

έννοιας του τύπου. Το πιο συνηθισμένο μήνυμα προς μια κλάση αφορά τη δημιουργία ενός νέου αντικειμένου, που αποτελεί *στιγμιότυπο* (instance) της κλάσης.

Θα μπορούσαμε να σημειώσουμε ότι η κάθε κλάση αποτελείται από δύο τύπους πληροφοριών: *ιδιότητες* (attributes) και *συμπεριφορά*. Οι ιδιότητες είναι τα δεδομένα που διαφοροποιούν μια κλάση αντικειμένων από μια άλλη. Μπορούν να χρησιμοποιηθούν για να καθορίσουν την εμφάνιση, την κατάσταση και άλλες ιδιότητες αντικειμένων που ανήκουν στην κλάση. Σε μια κλάση οι ιδιότητες ορίζονται με μεταβλητές όπου αποθηκεύονται πληροφορίες μέσα σε ένα πρόγραμμα.

Οι *μεταβλητές υπόστασης* (instance variables) είναι ιδιότητες που έχουν τιμές, οι οποίες διαφέρουν ανάμεσα σε αντικείμενα, αλλά είναι χαρακτηριστική ενός συγκεκριμένου αντικειμένου. Για παράδειγμα, μια κλάση `Αυτοκίνητο` μπορεί να ορίσει μια μεταβλητή υπόστασης ταχύτητα. Αυτή πρέπει να είναι μια μεταβλητή υπόστασης, αφού κάθε αυτοκίνητο ταξιδεύει εν γένει με διαφορετική ταχύτητα. Οι μεταβλητές υπόστασης μπορούν να αποκτήσουν μια τιμή όταν δημιουργείται ένα αντικείμενο και να παραμείνουν σταθερές για όλη τη διάρκεια ζωής του αντικειμένου ή να αποκτήσουν διαφορετικές τιμές, όταν το αντικείμενο χρησιμοποιείται σε ένα εκτελούμενο πρόγραμμα.

Σε αντιδιαστολή, οι *μεταβλητές κλάσης* (class variables) είναι ιδιότητες μιας ολόκληρης κλάσης. Εφαρμόζονται στην ίδια την κλάση και σε όλες τις υποστάσεις της, οπότε μόνο μια τιμή αποθηκεύεται, ανεξάρτητα από τον αριθμό των υποστασιασθέντων αντικειμένων της κλάσης. Για τη δήλωση των μεταβλητών αυτών χρησιμοποιείται ο τροποποιητής `static`. Σαν αντίστοιχο παράδειγμα με πριν, η κλάση `Αυτοκίνητο` μπορεί να ορίσει μια μεταβλητή κλάσης τρέχουσα_ώρα. Κάθε αντικείμενο της κλάσης θα μοιράζεται την ίδια τιμή για την τρέχουσα ώρα, ενώ δε θα ήταν σωστό αυτή η τιμή να εμανιζόταν διαφορετική μεταξύ των αντικειμένων, κάτι που θα προέκυπτε αν επρόκειτο για μεταβλητή υπόστασης.

Η συμπεριφορά μιας κλάσης αναφέρεται στις πράξεις που μπορεί να εκτελέσει αυτή πάνω στα αντικείμενά της και σε άλλα αντικείμενα που σχετίζονται με αυτήν (κληρονομικότητα). Υλοποιείται με τη χρήση μεθόδων, που είναι μια ομάδα πράξεων σε μια κλάση αντικειμένων, που χειρίζονται μια εργασία και χρησιμοποιούνται για την επικοινωνία μεταξύ αντικειμένων. Όπως υπάρχουν μεταβλητές υπόστασης και κλάσης, υπάρχουν επίσης μέθοδοι υπόστασης και κλάσης. Οι *μέθοδοι υπόστασης* εφαρμόζονται σε ένα αντικείμενο της κλάσης, ενώ οι *μέθοδοι κλάσης* εφαρμόζονται στην ίδια την κλάση.

Ειδική περίπτωση μεθόδων είναι οι *κατασκευαστές* (constructors) και οι *καταστροφείς* (destructors). Οι μέθοδοι αυτές καλούνται έμμεσα κατά την κατασκευή και καταστροφή αντικειμένων, αντίστοιχα. Οι κατασκευαστές εφαρμόζονται για να αρχικοποιήσουν ένα αντικείμενο, αμέσως μετά τη δέσμευση μνήμης για αυτό. Οι καταστροφείς καλούνται όταν ένα αντικείμενο καταστρέφεται, αμέσως πριν αποδεσμευθεί η μνήμη που αυτό καταλαμβάνει.

Σχετικά με τη χρήση μεθόδων, μια σημαντική παράμετρος που πρέπει να συζητηθεί είναι η *υπερφόρτωση* (overloading). Πρόκειται για την περίπτωση της χρήσης μεθόδων με το ίδιο όνομα και επιστρεφόμενο τύπο, αλλά διαφορετική *υπογραφή*. Η υπογραφή σε μια μέθοδο αποτελείται από δύο πράγματα:

- τον αριθμό των ορισμάτων που δέχεται
- τον τύπο δεδομένων κάθε ορίσματος

Για παράδειγμα παρακάτω (σε Java) η μέθοδος `print` είναι υπερφορτωμένη:

```
void print(int i) {
   ...
}

void print(int i, double d) {
   ...
```

```
    }

    void print(String str1) {
       ...
    }
```

Η υπερφόρτωση μεθόδων εξαλείφει την ανάγκη για τελείως διαφορετικές μεθόδους που εκτελούν στην ουσία την ίδια εργασία. Επίσης επιτρέπει σε μεθόδους να συμπεριφέρονται διαφορετικά με βάση τα ορίσματα που δέχονται. Προφανώς η υπερφόρτωση επιτρέπεται και στην περίπτωση μεθόδων δημιουργών, παρέχοντας τη δυνατότητα για τη δημιουργία ενός αντικειμένου με τις επιθυμητές ιδιότητες κάθε φορά. Στην επόμενη ενότητα θα αναφερθούμε και στην περίπτωση χρήσης μεθόδων σε διαφορετικές κλάσεις της ιεραρχίας (όχι μέσα στην ίδια κλάση) με το ίδιο όνομα, επιστρεφόμενο τύπο και υπογραφή, που ονομάζεται υπερσκέλιση. Ας σημειωθεί εδώ ότι δεν επιτρέπεται η χρήση μεθόδων (σε διαφορετικές κλάσεις της ιεραρχίας) με το ίδιο όνομα και υπογραφή, αλλά διαφορετικό επιστρεφόμενο τύπο, κάτι που διαφαίνεται μέσα από την έννοια της υπογραφής (δεν περιλαμβάνει τον επιστρεφόμενο τύπο).

2.2 Απλή κληρονομικότητα

Η *κληρονομικότητα* (inheritance) είναι ο τρόπος με τον οποίο οι περισσότερες γλώσσες αντικειμενοστρεφούς προγραμματισμού επιτρέπουν τη δημιουργία ιεραρχιών κλάσεων και μέσω αυτής την επαναχρησιμοποίηση κώδικα. Στις γλώσσες που υποστηρίζουν *απλή κληρονομικότητα* (single inheritance), κάθε κλάση μπορεί να κληρονομεί τα χαρακτηριστικά (ιδιότητες και συμπεριφορά) μιας άλλης, όχι όμως περισσότερων. Η κλάση που κληρονομεί τα χαρακτηριστικά ονομάζεται *παραγόμενη κλάση* (derived class) ή *υποκλάση* (subclass), ενώ η κλάση που κληροδοτεί τα χαρακτηριστικά ονομάζεται *βασική κλάση* (base class) ή *υπερκλάση* (superclass). Ο μηχανισμός της κληρονομικότητας είναι, τηρουμένων των αναλογιών, παρόμοιος με τον τρόπο που τα παιδιά κληρονομούν χαρακτηριστικά από τους γονείς τους, όπως το ύψος, το χρώμα των μαλλιών, των ματιών και άλλα. Εκτός από τα κληρονομούμενα χαρακτηριστικά, μια παραγόμενη κλάση μπορεί να ορίζει δικά της πεδία ή μεθόδους. Επίσης, μπορεί να επανορίζει μεθόδους (ίδιο όνομα, τύπος επιστροφής και ορίσματα) ή μεταβλητές, οι οποίες *επισκιάζουν* ή *υπερσκελίζουν* (override) τις αντίστοιχες μεθόδους (ή μεταβλητές) της βασικής κλάσης.

Για παράδειγμα, παρακάτω (σε γλώσσα Java) η κλάση `Shape` χρησιμοποιείται ως βάση μιας ιεραρχίας σχημάτων που κληρονομούν τα χαρακτηριστικά της. Ένα παράδειγμα κλάσης αυτής της ιεραρχίας, που κληρονομεί τη `Shape` είναι η κλάση `Circle`. Αυτή, εκτός από τα πεδία `posX`, `posY` και τις μεθόδους της κλάσης `Shape` περιέχει το πεδίο `radius` και τη μέθοδο `scale`. Άλλο παράδειγμα παραγόμενης κλάσης είναι η `Line`. Σε αυτή την κλάση, η μέθοδος `move` επισκιάζει την αντίστοιχη μέθοδο της κλάσης `Shape`.

```
class Shape {
   double posX, posY;

   void move (double dx, double dy) {
      posX = posX + dx;
      posY = posY + dy;
   }
};

class Circle extends Shape {
   double radius;

   void scale (double s) {
      radius = radius * s;
   }
```

```
};

class Line extends Shape {
   double endX, endY;

   void move (double dx, double dy) {
      posX = posX + dx;
      posY = posY + dy;
      endX = endX + dx;
      endY = endY + dy;
   }
};
```

Στις γλώσσες αντικειμενοστρεφούς προγραμματισμού υπάρχουν κάποιες λέξεις κλειδιά, οι λεγόμενοι *τροποποιητές* (modifiers), που μπορούν να προστεθούν στους ορισμούς κλάσεων, μεθόδων ή μεταβλητών για διάφορους σκοπούς. Τέτοιοι τροποποιητές είναι ο `static` για τη δημιουργία μεθόδων και μεταβλητών κλάσεων (που προαναφέραμε), ο `abstract` για τη δημιουργία αφηρημένων κλάσεων και μεθόδων (θα αναφερθούμε παρακάτω), καθώς και οι `public`, `protected` και `private` για τον έλεγχο της προσπέλασης σε μια κλάση, μέθοδο ή μεταβλητή. Οι τελευταίοι αυτοί τροποποιητές καθορίζουν ποιες μεταβλητές και μέθοδοι μιας κλάσης (ή και ποιες κλάσεις, όπως στη Java) είναι ορατές σε άλλες κλάσεις.

Ένα *δημόσιο* (public) μέλος μιας κλάσης μπορεί να χρησιμοποιηθεί από οποιαδήποτε άλλη κλάση. Αν είναι *ιδιωτικό* (private), μπορεί να χρησιμοποιηθεί μόνο μέσα στην ίδια την κλάση όπου ορίζεται. Τέλος, αν είναι *προστατευμένο* (protected), έχει τις ιδιότητες του `private` μέλους με την προσθήκη ότι είναι ορατό και στις δευτερεύουσες κλάσεις της κλάσης όπου ορίζεται. Όπως παρατηρούμε, αυτοί οι τροποποιητές επηρεάζουν καθοριστικά τον τρόπο με τον οποίο κληρονομούνται τα χαρακτηριστικά μιας κλάσης μέσα στην ιεραρχία των κλάσεων. Με τη χρήση τους υλοποιείται η *ενθυλάκωση* (encapsulation), που θα μπορούσε να οριστεί ως η διαδικασία απαγόρευσης σε μεταβλητές μιας κλάσης να διαβάζονται ή να τροποποιούνται από άλλες κλάσεις. Ο μόνος τρόπος για χρήση αυτών των μεταβλητών είναι καλώντας μεθόδους της κλάσης, αν είναι διαθέσιμες.

Μια έννοια που σε πολλές γλώσσες αντικειμενοστρεφούς προγραμματισμού συγχέεται με αυτήν της κληρονομικότητας είναι η έννοια του *υποτύπου* (subtype). Μια παραγόμενη κλάση είναι υποτύπος της αντίστοιχης βασικής κλάσης, με την έννοια ότι τα αντικείμενα της πρώτης μπορούν να θεωρηθούν και ως (εξειδικευμένα) αντικείμενα της δεύτερης. Φυσικά το αντίστροφο δεν μπορεί να συμβεί. Στο παραπάνω παράδειγμα, τα αντικείμενα της κλάσης `Line` μπορούν να χρησιμοποιηθούν και ως αντικείμενα της κλάσης `Shape`, αλλά ένα αντικείμενο της κλάσης `Shape` δεν μπορεί άμεσα να χρησιμοποιηθεί ως αντικείμενο της κλάσης `Line` ή της κλάσης `Circle`, κάτι που είναι και προφανές εννοιολογικά.

Σε κάθε κλήση μεθόδου ενός αντικειμένου που προέρχεται από κάποια κλάση μιας ιεραρχίας, πρέπει να είναι γνωστό σε ποια κλάση έχει οριστεί αυτή η μέθοδος. Αν η αναζήτηση των μεθόδων γίνεται από το μεταγλωττιστή βάσει του τύπου του αντικειμένου στο οποίο εφαρμόζεται η μέθοδος, τότε λέμε ότι η γλώσσα διαθέτει *στατικό δέσιμο μεθόδων* (static binding).[1]

Στο σημείο αυτό σημαντικό στοιχείο αποτελεί η έννοια της *αναφοράς* (reference), που είναι ουσιαστικά ένας τύπος δείκτη που χρησιμοποιείται για να δηλώσει την τιμή ενός αντικειμένου. Έτσι, όταν εκχωρούμε ένα αντικείμενο σε μια μεταβλητή ή δίνουμε ένα αντικείμενο ως όρισμα σε μια μέθοδο, δε χρησιμοποιούμε στην πραγματικότητα αντικείμενα, ούτε καν αντίγραφα του αντικειμένου. Αντί αυτού, χρησιμοποιούμε αναφορές σε αντικείμενα.

Παραδείγματος χάρη, έστω ότι στο προηγούμενο παράδειγμα έχουν προστεθεί οι ορισμοί των παρακάτω μεθόδων δημιουργών, όπου το δεσμευμένο αναγνωριστικό super χρησιμοποιείται σε παραγόμενες κλάσεις για αναφορά στο αντικείμενο της υπερκλάσης και συγκεκριμένα (με

[1] Το δυναμικό δέσιμο μεθόδων περιγράφεται στην ενότητα 2.3.

την παρούσα σύνταξη) στον κατασκευαστή της υπερκλάσης[2]:

```
class Shape {
...
   Shape(double x, double y){
      posX = x;
      posY = y;
   }
}

class Circle extends Shape {
...
   Circle(double x, double y, double r) {
      super(x, y);
      radius = r;
   }
}

class Line extends Shape {
...
   Line(double x, double y, double z, double w) {
      super(x, y);
      endX = z;
      endY = w;
   }
}
```

Ακολούθως, ορίζονται τα παρακάτω αντικείμενα (σε κάποια άλλη κλάση):

```
Shape s = new Shape(10, 20);
Circle c = new Circle(30, 30, 10);
Line l = new Line(10, 20, 30, 30);
```

Στη συνέχεια εφαρμόζονται οι παρακάτω μέθοδοι:

```
s.move(5, 5);
c.move(5, 5);
l.move(5, 5);
c.scale(2)
```

Βάσει του τύπου των αντικειμένων `s`, `c`, `l` και των ορισμών των κλάσεων, ο μεταγλωττιστής θα καλέσει διαδοχικά τις παρακάτω μεθόδους:

```
move(5, 5) της κλάσης Shape για το s
move(5, 5) της κλάσης Shape για το c
move(5, 5) της κλάσης Line για το l
scale(2) της κλάσης Shape για το c
```

Η κλάση `Circle` δεν ορίζει τη μέθοδο `move`, κατά συνέπεια η κλήση `c.move` οδηγεί σε κλήση της αντίστοιχης μεθόδου της `Shape`. Αντίθετα, η κλάση `Line` επανορίζει αυτή τη μέθοδο και άρα η κλήση `l.move` οδηγεί σε κλήση της νέας μεθόδου.

Ας υποθέσουμε ότι στο ίδιο πρόγραμμα Java περιέχονται και τα ακόλουθα:

[2] Στην περίπτωση της γλώσσας Ooedsger δεν χρησιμοποιείται η σύνταξη αυτή για το `super` (ή για το `this`, που είναι αναφορά στο αντικείμενο της τρέχουσας κλάσης, και με την αντίστοιχη σύνταξη χρησιμοποιείται για προσπέλαση σε κατασκευαστή της τρέχουσας κλάσης), αφού επιτρέπεται ο ορισμός πολλών κατασκευαστών με διαφορετικό όνομα μέσα στην ίδια την κλάση, οπότε ο κάθε κατασκευαστής μπορεί να κληθεί με το όνομά του, όπως θα περιγραφεί στην ενότητα 3.3.3.

```
Shape p;
...
p = l;
p.move(5, 5);
```

Η αναφορά `p` τοποθετείται να δείχνει στο αντικείμενο `l`. Η ανάθεση αυτή επιτρέπεται, γιατί η κλάση `Line` στην οποία ανήκει το αντικείμενο `l` είναι υποτύπος της κλάσης `Shape`. Επειδή όμως η κλήση `p.move` επιλύεται στατικά και η έκφραση `p` έχει τύπο `Shape`, όπως προκύπτει από τον έλεγχο τύπων κατά τη διάρκεια της μεταγλώττισης, η μέθοδος που καλείται είναι η `move` της κλάσης `Shape`.

2.3 Πολυμορφισμός υποτύπων

Οι πρώτες γλώσσες αντικειμενοστρεφούς προγραμματισμού δε χρησιμοποιούσαν για την επίλυση της κλήσης μεθόδων τον αλγόριθμο του στατικού δεσίματος, αλλά αναζητούσαν δυναμικά τη μέθοδο που έπρεπε να κληθεί. Ο τρόπος αυτός του δεσίματος των μεθόδων ονομάζεται *δυναμικό δέσιμο* (dynamic binding) και υποστηρίζεται από όλες τις σύγχρονες γλώσσες αντικειμενοστρεφούς προγραμματισμού (Java, C++). Η έννοια του *πολυμορφισμού υποτύπων* (subtype polymorphism), που συχνά ονομάζεται με τον καταχρηστικό όρο "πολυμορφισμός" στην ορολογία των αντικειμενοστρεφών γλωσσών,[3] ταυτίζεται ουσιαστικά με το δυναμικό δέσιμο των μεθόδων.

Ο μεταγλωττιστής μιας γλώσσας που υποστηρίζει το δυναμικό δέσιμο μεθόδων δεν είναι πάντοτε σε θέση να αποφασίσει ποια μέθοδος πρέπει να κληθεί, κατά το χρόνο της μεταγλώττισης. Αν ο δείκτης (reference) στο οποίο εφαρμόζεται η μέθοδος προέρχεται από την εκχώρηση σε αυτό ενός αντικειμένου, όπως στην περίπτωση του τελευταίου παραδείγματος της προηγούμενης ενότητας, τότε ο μεταγλωττιστής συνήθως δεν είναι σε θέση να γνωρίζει την πραγματική κλάση στην οποία ανήκει αυτό το αντικείμενο. Η επίλυση της μεθόδου άρα μπορεί να γίνει μόνο δυναμικά, κατά το χρόνο εκτέλεσης του προγράμματος. Στο εν λόγω παράδειγμα, με δυναμικό δέσιμο μεθόδων θα κληθεί για το δείκτη `p` η μέθοδος `move` της Line.

Πολλές γλώσσες αντικειμενοστρεφούς προγραμματισμού υποστηρίζουν δυναμικές[4] μεθόδους που δηλώνονται μεν σε κάποια κλάση, ο κώδικάς τους όμως ορίζεται μόνο σε υποκλάσεις αυτής. Οι μέθοδοι αυτές ονομάζονται συνήθως *αφηρημένες* (abstract) και οι κλάσεις που περιέχουν τέτοιες μεθόδους ονομάζονται αφηρημένες κλάσεις. Για το σκοπό αυτό συχνά χρησιμοποιείται ο τροποποιητής `abstract`. Οι κλάσεις αυτές συνήθως δεν διατίθενται για την κατασκευή αντικειμένων (όχι αναφορών) των κλάσεων αυτών ή δεν έχει νόημα να δημιουργηθούν τέτοια αντικείμενα.

2.4 Πολλαπλή κληρονομικότητα

Στην περίπτωση της απλής κληρονομικότητας, που εξετάσαμε μέχρι τώρα, η προσπέλαση των πεδίων και των μεθόδων ενός αντικειμένου ήταν αρκετά απλή. Τα πράγματα γίνονται πιο πολύπλοκα σε γλώσσες που επιτρέπουν *πολλαπλή κληρονομικότητα* (multiple inheritance), όπως η C++. Σε αυτές, μια κλάση μπορεί να κληρονομεί χαρακτηριστικά περισσότερων κλάσεων. Αυτό το χαρακτηριστικό δεν υποστηρίζεται από την προς υλοποίηση γλώσσα Ooedsger και για το λόγο αυτό δε θα γίνει εκτενής αναφορά στην παρούσα ενότητα.

[3] Αυτός ο τύπος πολυμορφισμού δε σχετίζεται με τον *παραμετρικό πολυμορφισμό* (parametric polymorphism) που συναντάται συνήθως σε γλώσσες της σχολής του συναρτησιακού προγραμματισμού και που συχνά ονομάζεται επίσης "πολυμορφισμός". Για την αποφυγή παρερμηνειών είναι προτιμότερο να χρησιμοποιούνται τα περιγραφικά ονόματα, ιδιαίτερα σε γλώσσες, όπως η C++, που υποστηρίζουν και τους δύο τύπους (πολυμορφισμό υποτύπων στις ιεραρχίες κλάσεων και παραμετρικό πολυμορφισμό με το μηχανισμό των templates).

[4] Στην ορολογία των αντικειμενοστρεφών γλωσσών προγραμματισμού, οι δυναμικές μέθοδοι συχνά ονομάζονται και *εικονικές* (virtual) μέθοδοι.

Ως παράδειγμα πολλαπλής κληρονομικότητας, παρακάτω (σε C++) η κλάση `employee`, καθώς και η κλάση `temporary` χρησιμοποιούνται ως βάσικές κλάσεις μιας ιεραρχίας εργαζομένων που κληρονομούν τα χαρακτηριστικά τους. Ένα παράδειγμα κλάσης αυτής της ιεραρχίας, που κληρονομεί (με απλή κληρονομικότητα) την `employee` είναι η κλάση `secretary`. Παραγόμενη κλάση με χρήση πολλαπλή κληρονικότητας είναι η `temp_secr`, που κληρονομεί τόσο από την `temporary` όσο και από την `secretary`:

```
class employee { // υπάλληλος
...
};

class temporary { // προσωρινός
...
};

class secretary : public employee { // γραμματέας
...
};

class temp_secr : public temporary, public secretary { // προσωρινός γραμματέας
...
};
```

Αν για την παράσταση στη μνήμη των αντικειμένων, τα τμήματα που αντιστοιχούν στις κληρονομούμενες κλάσεις είναι ανεξάρτητα μεταξύ τους, η πολλαπλή κληρονομικότητα ονομάζεται *ανεξάρτητη* (independent). Ένα άλλο είδος πολλαπλής κληρονομικότητας είναι η *εξαρτημένη* (dependent). Τα δύο είδη διακρίνονται μόνο στην περίπτωση που δύο ή περισσότερες κληρονομούμενες κλάσεις είναι απόγονοι της ίδιας κλάσης. Τότε, στην περίπτωση της ανεξάρτητης πολλαπλής κλήρονομικότητας τα πεδία της κοινής κλάσης εμφανίζονται περισσότερες φορές στην παράσταση του αντικειμένου της υποκλάσης, ενώ στην περίπτωση της εξαρτημένης πολλαπλής κληρονομικότητας εμφανίζονται μόνο μία φορά. Η πολλαπλή κληρονομικότητα, αν και παρέχει τον τρόπο δημιουργίας κλάσεων και ενσωμάτωσης σχεδόν κάθε δυνατής συμπεριφοράς, ωστόσο ιδιαίτερα η εξαρτημένη περιπλέκει τον ορισμό των κλάσεων και την παράσταση των αντικειμένων στη μνήμη.

2.5 Έλεγχος υποτύπων και μετατροπές

Συχνά είναι χρήσιμο να μπορεί ο προγραμματιστής να βρει κατά το χρόνο εκτέλεσης του προγράμματος σε ποια κλάση ανήκει πραγματικά κάποιο αντικείμενο. Λίγο πιο δύσκολη είναι η περίπτωση που πρέπει να ελεγχθεί αν ένα αντικείμενο ανήκει ή όχι σε κάποια δεδομένη κλάση ή σε κάποια υποκλάση αυτής. Ο έλεγχος αυτός λέγεται *έλεγχος υποτύπων* (subtype testing). Για το σκοπό αυτό στη Java διατίθεται ο τελεστής `instanceof`, που δέχεται δύο τελεστέους: μια αναφορά προς ένα αντικείμενο ως πρώτο και ένα όνομα κλάσης ως δεύτερο. Η έκφραση επιστρέφει `true` ή `false` ανάλογα με το αν το αντικείμενο είναι μια υπόσταση της ονομασμένης κλάσης ή μιας από τις δευτερεύουσες κλάσεις της κλάσης:

```
boolean b1 = "Texas" instanceof String;   // true
Object obj = new Shape(5, 10);
boolean b2 = obj instanceof String;       // false
```

Στις αντικειμενοστρεφείς γλώσσες σημαντική θέση κατέχουν και οι *μετατροπές ή αλλαγές τύπων* (casting) μεταξύ αντικειμένων από μια κλάση προέλευσης σε μια κλάση προορισμού. Αυτό είναι επιτρεπτό, με τον εξής καταρχάς περιορισμό: Οι κλάσεις προέλευσης και προορισμού πρέπει να σχετίζονται με κληρονομικότητα. Η μια κλάση πρέπει να είναι δευτερεύουσα κλάση της άλλης.

Αν η κλάση προέλευσης είναι υποκλάση αυτής του προορισμού (*upcasting*), δε χρειάζεται να γίνει σαφής αλλαγή τύπου κι έτσι μια υπόσταση μιας δευτερεύουσας κλάσης μπορεί να χρησιμοποιηθεί οπουδήποτε αναμένεται μια υπερκλάση. Αντίθετα, αν θέλουμε να χρησιμοποιήσουμε αντικείμενα υπερκλάσης όταν αναμένονται αντικείμενα δευτερεύουσας κλάσης (*downcasting*), γενικά θα πρέπει το αντικείμενο της κλάσης προέλευσης να είναι στην πραγματικότητα ένα αντικείμενο της κλάσης προορισμού ή μιας υποκλάσης αυτής. Στην τελευταία περίπτωση πρέπει να γίνει σαφώς αλλαγή τύπου. Για παράδειγμα στη Java:

```
Shape s = new Shape(3, 5);
Circle c = new Circle(5, 8, 10);
s = c;                    // δε χρειάζεται ρητή αλλαγή τύπου (upcasting)
c = (Circle)s;            // πρέπει να γίνει σαφής αλλαγή τύπου (downcasting)
```

2.6 Διαπροσωπείες

Οι *διαπροσωπείες ή διασυνδέσεις* (interfaces) είναι η εναλλακτική λύση που υιοθετήθηκε στη γλώσσα Java αντί της πολλαπλής κληρονομικότητας. Παρέχουν μερικά από τα πλεονεκτήματα της πολλαπλής κληρονομικότητας, χωρίς να εισάγουν την πολυπλοκότητα που αναφέρθηκε στην ενότητα του πολυμορφισμού. Μια διαπροσωπεία είναι ουσιαστικά μια κλάση χωρίς πεδία δεδομένων, της οποίας όλες οι μέθοδοι είναι αφηρημένες. Οι διαπροσωπείες μπορούν να είναι οργανωμένες σε ιεραρχίες, όπως ακριβώς και οι κλάσεις, με απλή ή πολλαπλή κληρονομικότητα. Κάθε κλάση μπορεί να υλοποιεί μια ή περισσότερες διαπροσωπείες. Έτσι, οι διαπροσωπείες, που υλοποιούνται από κάποια κλάση, είναι συλλογές μεθόδων που δηλώνουν ότι η κλάση αυτή έχει κάποια συμπεριφορά επιπρόσθετα σε αυτή που κληρονομεί από τις υπερκλάσεις της. Οι μέθοδοι που περιλαμβάνονται σε μια διαπροσωπεία δεν καθορίζουν αυτήν τη συμπεριφορά, αλλά αυτή η εργασία αφήνεται στις κλάσεις που υλοποιούν τη διαπροσωπεία. Για παράδειγμα, έστω παρακάτω η διαπροσωπεία `Printable` που υλοποιείται από τις κλάσεις `Circle` και `Line`, οι οποίες καλούνται να υλοποιήσουν τη μέθοδο `print()` ανάλογα με τη συμπεριφορά που χρειάζονται:

```
interface Printable {
   void print();
}

class Circle extends Shape implements Printable {
   ...

   void print() {
      ...
   }
}

class Line extends Shape implements Printable {
   ...

   void print() {
      ...
   }
}
```

Κεφάλαιο 3

Η γλώσσα Ooedsger

Η γλώσσα Ooedsger είναι μια απλή γλώσσα αντικειμενοστρεφούς προγραμματισμού βασισμένη στο προστακτικό μοντέλο προγραμματισμού. Επεκτείνει τη γραμματική της γλώσσας Edsger, χωρίς να περιλαμβάνει φωλιασμένες συναρτήσεις, εμπλουτίζοντάς την με αντικειμενοστρεφή χαρακτηριστικά, όντας ένα γνήσιο υποσύνολο της Java. Τα κύρια χαρακτηριστικά της εν συντομία είναι τα εξής:

- Σύνταξη παρόμοια σε γενικές γραμμές με αυτή της Java.
- Δυναμικό δέσιμο μεθόδων και υποστήριξη κλήσης κατ' αναφορά.
- Υποστήριξη απλής κληρονομικότητας μεταξύ κλάσεων και διασυνδέσεων.
- Βασικοί τύποι δεδομένων για ακέραιους αριθμούς, χαρακτήρες, λογικές τιμές και πραγματικούς αριθμούς.
- Τύποι δεικτών και (έμμεσα) πίνακες, συνδεδεμένοι σημασιολογικά όπως στη C, καθώς επίσης και τύποι αναφορών σε κλάσεις ή διασυνδέσεις, όπως στη Java.
- Βιβλιοθήκη συναρτήσεων.

Περισσότερες λεπτομέρειες της γλώσσας δίνονται στις παραγράφους που ακολουθούν.

3.1 Λεκτικές μονάδες

Οι λεκτικές μονάδες της γλώσσας Ooedsger χωρίζονται στις παρακάτω κατηγορίες:

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

```
abstract     bool       break         byref        char
class        continue   constructor   delete       destroy
destructor   double     else          extends      for
false        if         implements    instanceof   int
interface    new        nil           NULL         return
static       super      this          true         void
```

- 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 ονόματα δεν πρέπει να συμπίπτουν με τις λέξεις κλειδιά που αναφέρθηκαν παραπάνω.
- Οι *ακέραιες σταθερές* χωρίς πρόσημο, που αποτελούνται από ένα ή περισσότερα δεκαδικά ψηφία. Παραδείγματα ακέραιων σταθερών είναι τα ακόλουθα:

```
0    42    1284    00200
```

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)
\x*nn*	ο χαρακτήρας με ASCII κωδικό *nn* στο δεκαεξαδικό σύστημα

- 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 μια 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 Ooedsger 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 μιας γραμμής*, τα οποία αρχίζουν με την ακολουθία χαρακτήρων `//` και τερματίζονται με το τέλος της τρέχουσας γραμμής.
- *Σχόλια πολλών γραμμών*, τα οποία αρχίζουν με την ακολουθία χαρακτήρων `/*` και τερματίζονται με την ακολουθία χαρακτήρων `*/`. Τα σχόλια αυτής της μορφής επιτρέπεται να είναι φωλιασμένα. Στο εσωτερικό τους επιτρέπεται η εμφάνιση οποιουδήποτε χαρακτήρα.

3.2 Τύποι δεδομένων

Η Ooedsger υποστηρίζει τέσσερις βασικούς τύπους δεδομένων:

- `int` : ακέραιοι αριθμοί,
- `char` : χαρακτήρες,
- `bool` : λογικές τιμές, και
- `double` : πραγματικοί αριθμοί.

Οι τύποι `int` και `double` ονομάζονται *αριθμητικοί* τύποι.

Εκτός από τους βασικούς τύπους, η Ooedsger υποστηρίζει επίσης τύπους δεικτών, που συμβολίζονται με t `*`, όπου ο t θεωρούμε ότι είναι ένας έγκυρος τύπος, αλλά όχι τύπος αναφοράς (reference). Κάθε δείκτης τύπου t `*` μπορεί να χρησιμοποιηθεί ως ένας μονοδιάστατος πίνακας, αποτελούμενος από στοιχεία τύπου t. Στην περίπτωση αυτή, η αρίθμηση των στοιχείων του πίνακα ακολουθεί τη σύμβαση της C: το πρώτο στοιχείο βρίσκεται στη θέση 0, το δεύτερο στη θέση 1, ενώ το τελευταίο βρίσκεται σε θέση κατά ένα μικρότερη από το πλήθος των στοιχείων του πίνακα. Παράλληλα, υποστηρίζονται τύποι αναφορών σε κλάσεις ή διασυνδέσεις που συμβολίζονται με `obj` ή `obji` αντίστοιχα, αρκεί προφανώς να έχουν ορισθεί οι αντίστοιχες κλάσεις ή διασυνδέσεις κάπου μέσα στο πρόγραμμα. Εξυπονοείται εδώ ότι η εμβέλεια του ορισμού μιας κλάσης εκτείνεται από την αρχή του προγράμματος (και όχι από το σημείο ορισμού) μέχρι το τέλος του, όπως συμβαίνει στη Java.

Ο αριθμός των bytes που καταλαμβάνουν τα δεδομένα κάθε τύπου στη μνήμη του υπολογιστή, καθώς και ο τρόπος παράστασης των δεδομένων περιγράφεται παρακάτω. Η περιγραφή διευκολύνει την υλοποίηση μεταγλωττιστών για υπολογιστές βασισμένους σε επεξεργαστές της σειράς x86 της Intel, με το μοντέλο προγραμμάτων COM.

- `int` : μέγεθος 2 bytes, παράσταση συμπληρώματος ως προς 2.
- `char` : μέγεθος 1 byte, παράσταση σύμφωνα με τον πίνακα ASCII.
- `bool` : μέγεθος 1 byte, τιμή 0 για `false` και $\neq 0$ για `true`.
- `double` : μέγεθος 10 bytes, παράσταση σύμφωνα με το πρότυπο IEEE 754.
- t `*` : μέγεθος 2 bytes.
- `obj` : μέγεθος 2 bytes.
- `obji` : μέγεθος 4 bytes.

3.3 Δομή του προγράμματος

3.3.1 Κλάσεις και διαπροσωπείες

Ένα πρόγραμμα Ooedsger αποτελείται από μια ή περισσότερες δηλώσεις κλάσεων ή και διαπροσωπειών με την αντίστοιχη σύνταξη που ακολουθεί η Java. Η δήλωση κλάσης γίνεται με αναγραφή του ονόματος της κλάσης, ακολουθούν προαιρετικά τα ονόματα της υπερκλάσης και των διαπροσωπειών από τις οποίες τυχόν κληρονομεί η δηλούμενη κλάση, ενώ στο τέλος δίνονται καμία, μία ή περισσότερες δηλώσεις στοιχείων της κλάσης. Η κληρονομικότητα στην ιεραρχία των κλάσεων είναι απλή, ενώ κάθε κλάση μπορεί να κληρονομεί από περισσότερες από μία διαπροσωπείες. Στην περίπτωση των διαπροσωπειών η δήλωσή τους επιτρέπει απλή κληρονομικότητα στην ιεραρχία των διαπροσωπειών, ενώ στις δηλώσεις στοιχείων της διαπροσωπείας επιτρέπονται μόνο δηλώσεις μεθόδων (θα αναφερθούμε αργότερα). Όπως είναι φυσικό απαγορεύεται η δημιουργία κύκλου είτε στην ιεραρχία των κλάσεων είτε στην ιεραρχία των διαπροσωπειών. Παραδείγματα των δηλώσεων αυτών είναι:

```
class Circle extends Shape implements Printable, Whateverable {
... // δηλώσεις πεδίων της κλάσης Circle
};

class Shape {
... // δηλώσεις πεδίων της κλάσης Shape
};

interface Printable extends YAInterf1, YAInterf2 {
... // δηλώσεις μεθόδων της διαπροσωπείας Printable
};

interface YAInterf1 {
... // δηλώσεις μεθόδων της διαπροσωπείας YAInterf1
};

interface YAInterf2 {
... // δηλώσεις μεθόδων της διαπροσωπείας YAInterf2
};

interface Whateverable {
... // δηλώσεις μεθόδων της διαπροσωπείας Whateverable
};
```

Οι δηλώσεις των στοιχείων μιας κλάσης ανήκουν σε τρεις κατηγορίες:

- Δηλώσεις μεταβλητών (variable declarations),
- Ορισμοί μεθόδων (method definitions),
- Δηλώσεις μεθόδων (method declarations).

Σε κάθε πρόγραμμα πρέπει να υπάρχει μέσα σε μια κλάση ο ορισμός μιας μεθόδου με επικεφαλίδα:

```
static void main()
```

από την οποία ξεκινά η εκτέλεση του προγράμματος.

3.3.2 Μεταβλητές

Οι δηλώσεις μεταβλητών μέσα σε κλάσεις (εκτός μεθόδων) γίνονται με προαιρετική αναγραφή του τροποποιητή static, και κατόπιν του τύπου, ακολουθούμενου από έναν ή περισσότερους *δηλωτές* (declarators) μεταβλητών. Η απλούστερη μορφή δηλωτή είναι ένα αναγνωριστικό (identifier), που αντιστοιχεί στο όνομα της μεταβλητής. Προαιρετικά ένας δηλωτής μπορεί να περιέχει στο τέλος του μια έκφραση αρχικοποίησης, η οποία προφανώς θα πρέπει να συμφωνεί με τον τύπο της αντίστοιχης μεταβλητής που δηλώνεται. Παραδείγματα τέτοιων δηλώσεων είναι:

```
int i = 7;
static double x = 0.1, y = 6.023e23;
Shape s = makeShape(5.6, 6.2);
Printable p1, p2;
int * p, q;
```

Στην τρίτη γραμμή του προηγούμενου παραδείγματος έχουμε τη δήλωση μιας αναφοράς στην κλάση **Shape** και την εκχώρηση σε αυτήν (την αναφορά) ενός αντικειμένου με κλήση της μεθόδου κατασκευαστή (θα αναφερθούμε στην ενότητα 3.3.3) **makeShape** για την κλάση. Ας σημειωθεί ακόμη ότι, αντίθετα με τη γλώσσα C, στην τελευταία δήλωση και οι δύο μεταβλητές p και q έχουν τύπο `int *`. Επίσης, η χρήση του τροποποιητή static καθιστά μια μεταβλητή ως μεταβλητή κλάσης (οι `x`, `y` στη δεύτερη γραμμή), ενώ η απουσία του ως μεταβλητή υπόστασης. Οι δηλώσεις μεταβλητών μέσα σε μια μέθοδο δε διαφέρουν, παρά μόνο στην απουσία αυτού του (προαιρετικού για την περίπτωση των κλάσεων) τροποποιητή.

Για τη διευκόλυνση της δήλωσης μεταβλητών τύπου δείκτη που χρησιμοποιούνται ως μονοδιάστατοι πίνακες, η γλώσσα Ooedsger υποστηρίζει ένα δεύτερο τύπο δηλωτή που αποτελείται από ένα αναγνωριστικό και την επιθυμητή διάσταση του πίνακα μέσα σε αγκύλες. Παραδείγματα τέτοιων δηλώσεων είναι:

```
int a[100];
double x[5], y[7];
int * p[12];
```

Στις παραπάνω δηλώσεις, η μεταβλητή a έχει τύπο `int *` (δείκτης σε ακέραιο, δηλαδή ισοδύναμα μονοδιάστατος πίνακας ακεραίων), οι μεταβλητές `x` και `y` έχουν τύπο `double *`, ενώ η μεταβλητή p έχει τύπο `int **` (δείκτης σε δείκτη σε ακέραιο, δηλαδή ισοδύναμα μονοδιάστατος πίνακας δεικτών σε ακέραιο).

Οι μεταβλητές τύπου δείκτη που δηλώνονται με δηλωτές της δεύτερης μορφής ονομάζονται *μόνιμες* (immutable) και η τιμή τους δεν μπορεί να μεταβληθεί κατά την εκτέλεση του προγράμματος. Για παράδειγμα, το παρακάτω είναι σφάλμα:

```
int a[100], b[200];

a = b;  // wrong!
```

3.3.3 Μέθοδοι

Κάθε μέθοδος είναι μια δομική μονάδα και αποτελείται από την επικεφαλίδα της και το σώμα της. Στην επικεφαλίδα αναφέρεται τυχόν τροποποιητής `static` ή `abstract`, ο τύπος επιστροφής της μεθόδου, το όνομά της και οι τυπικές της παράμετροι μέσα σε παρενθέσεις. Οι παρενθέσεις είναι υποχρεωτικές ακόμα και αν μια μέθοδος δεν έχει τυπικές παραμέτρους. Ο τροποποιητής `static` χρησιμοποιείται, αντίστοιχα με την περίπτωση των μεταβλητών, για να καθορίσει ότι πρόκειται για μέθοδο κλάσης. Η απουσία αυτού συνεπάγεται ότι πρόκειται για μέθοδο υπόστασης, άρα δυναμική μέθοδο, λόγω του δυναμικού δεσίματος μεθόδων στην Ooedsger.

Ο τροποποιητής **abstract** δηλώνει ότι η μέθοδος είναι αφηρημένη, δηλαδή δεν ορίζεται μέσα στην κλάση, αλλά μόνο δηλώνεται. Αν βέβαια μέσα σε μια κλάση ή διαπροσωπεία εμφανισθεί η δήλωση μιας μεθόδου (χωρίς υλοποίηση), τότε η μέθοδος θεωρείται αφηρημένη, χωρίς να χρειάζεται να δηλωθεί ως τέτοια. Σε αυτήν την περίπτωση η κλάση που την περιλαμβάνει θεωρείται αφηρημένη, που σημαίνει ότι δεν επιτρέπεται να κατασκευασθούν αντικείμενα αυτής, άρα δεν επιτρέπεται να υπάρχουν μέσα στην κλάση μέθοδοι δημιουργοί (κατασκευαστές). Οι μέθοδοι αυτοί μπορούν να υλοποιηθούν από δευτερεύουσες κλάσεις, αλλιώς καθίστανται και αυτές αφηρημένες. Στην περίπτωση των διαπροσωπειών όμως, οι οποίες, όπως προαναφέρθηκε, περιλαμβάνουν μόνο δηλώσεις μεθόδων, κάθε κλάση που υλοποιεί μία από αυτές θα πρέπει είτε η ίδια ή κάποια υπερκλάση της να παρέχει τον ορισμό όλων των μεθόδων που περιλαμβάνονται στη διαπροσωπεία.

Ο τύπος αποτελέσματος **void** υποδηλώνει ότι μια μέθοδος δεν επιστρέφει αποτέλεσμα, ο **constructor** ότι πρόκειται για μέθοδο κατασκευαστή της κλάσης, ενώ ο **destructor** ότι η μέθοδος είναι καταστροφέας (!) της κλάσης. Δεν επιτρέπεται μια κλάση να περιλαμβάνει περισσότερους από έναν καταστροφείς. Μάλιστα αν δεν δοθεί ο ορισμός μιας μεθόδου καταστροφέα, παρέχεται αυτόματα ένας καταστροφέας για την κλάση με κενό σώμα και όνομα **@class_destruc**. Επίσης, υποστηρίζεται η υπερφόρτωση και η υπέρβαση για τη δημιουργία μεθόδων, έννοιες που αναπτύχθηκαν στις ενότητες 2.1 και 2.2. Ακόμη, επεκτείνοντας το πρότυπο της Java, επιτρέπεται η δημιουργία κατασκευαστών με οποιοδήποτε όνομα (όχι υποχρεωτικά το όνομα της κλάσης), επιτρέποντας έτσι τη χρήση κατασκευαστών με την ίδια υπογραφή (αλλά διαφορετικό όνομα) και διαφορετική υλοποίηση ανάλογα με τις εργασίες που θέλουμε να επιτελούν.

Κάθε τυπική παράμετρος χαρακτηρίζεται από το όνομά της, τον τύπο της και τον τρόπο περάσματος. Σε αντίθεση με τη Java, η γλώσσα Ooedsger υποστηρίζει πέρασμα παραμέτρων *κατ' αξία* (by value) και *κατ' αναφορά* (by reference). Αν στη δήλωση μιας τυπικής παραμέτρου έχει προηγηθεί η λέξη κλειδί **byref**, τότε αυτή περνά κατ' αναφορά, διαφορετικά περνά κατ' αξία.

Ακολουθούν παραδείγματα επικεφαλίδων μεθόδων:

```
static void p1 (Shape s, Circle c)
abstract void p2 (int n)
constructor makeShape (double posX, double posY)
void p3 (int a, int b, byref bool b)
double f1 (double x, Whateverable w)
int  f2 (byref char * s)
double * f3 (double x)
```

Το σώμα μιας μεθόδου περικλείεται μέσα σε άγκιστρα { και }. Μπορεί να περιέχει δηλώσεις ή και εντολές με οποιαδήποτε σειρά. Η Ooedsger ακολουθεί τους κανόνες εμβέλειας της Java, όσον αφορά στην ορατότητα των ονομάτων μεταβλητών, μεθόδων και παραμέτρων.

3.4 Εκφράσεις

Κάθε έκφραση της Ooedsger 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 δίνουν l-values, οι οποίες περιγράφονται στην ενότητα 3.4.1, και αυτές που δίνουν 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 δεξιό.

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

[1] Εξαίρεση αποτελούν η σταθερά μηδενικού δείκτη NULL, καθώς και η σταθερά μηδενικής αναφοράς (για τις κλάσεις ή διαπροσωπείες) nil, οι οποίες δεν έχουν μοναδικό τύπο. Περιγράφονται στην ενότητα 3.4.2.

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, είτε τοπικές μέσα σε μια μέθοδο είτε μεταβλητές υπόστασης ή κλάσης, επίσης οι παράμετροι των μεθόδων και οι μεταβλητές που κατασκευάζονται με δυναμική παραχώρηση μνήμης, είτε με χρήση του `new` είτε με την κλήση κάποιου κατασκευαστή για μια μεταβλητή αναφοράς σε κλάση. Συγκεκριμένα:

- Το όνομα μιας μη μόνιμης μεταβλητής ή μιας παραμέτρου μεθόδου είναι l- value και αντιστοιχεί στο εν λόγω αντικείμενο. Ο τύπος της l-value είναι ο τύπος του αντίστοιχου αντικειμένου. Εδώ εισάγουμε τη σημασία του τελεστή της τελείας (`.`), ο οποίος χρησιμοποιείται για την αναφορά σε μεταβλητές και μεθόδους μιας κλάσης. Πρόκειται για ένα τελεστή με 2 τελούμενα, εκ των οποίων το πρώτο πρέπει να είναι ένα έγκυρο αντικείμενο ή το όνομα μιας κλάσης, ενώ το δεύτερο πρέπει να είναι το όνομα της μεταβλητής ή μεθόδου για προσπέλαση. Αν το πρώτο τελούμενο είναι το όνομα μιας κλάσης, τότε η μεταβλητή ή μέθοδος θα πρέπει να προσδιορίζεται ως στατική (static) μέσα στην κλάση αυτή. Αν το πρώτο τελούμενο είναι ένα αντικείμενο, τότε η μεταβλητή ή μέθοδος μπορεί να είναι είτε στατική μέσα στην κλαση αναφοράς του αντικειμένου είτε θα πρόκειται για μεταβλητή ή μέθοδο υπόστασης του αντικειμένου, όπως συμβαίνει στην Java. Για παράδειγμα, αν έχουμε ένα αντικείμενο `myShape` και αυτό έχει μια μεταβλητή με όνομα `pos`, μπορούμε να αναφερθούμε στην τιμή αυτής της μεταβλητής ως εξής:

  ```
  myShape.pos = ... ;
  ```

  Αυτή η μορφή προσπέλασης μεταβλητών είναι μια έκφραση (επιστρέφει μια τιμή) και επίσης το πρώτο τελούμενο της τελείας είναι μια έκφραση. Αυτό σημαίνει ότι μπορούμε να ενθέσουμε προσπέλαση μεταβλητής υπόστασης. Σημειώνουμε εδώ ότι οι εκφράσεις τελείας αποτιμώνται από αριστερά προς τα δεξιά. Έτσι, αν στο προηγούμενο παράδειγμα η μεταβλητή υπόστασης `pos` περιέχει ένα αντικείμενο (άρα θα έχει τύπο αναφοράς) και αυτό έχει τη δική του μεταβλητή υπόστασης με όνομα `yapos`, μπορούμε να αναφερθούμε σε αυτήν ως εξής:

  ```
  myShape.pos.yapos = ... ;
  ```

  Έτσι, μια έγκυρη έκφραση με τον τελεστή της τελείας, όπου ως δεύτερο τελούμενο της τελευταίας εμφάνισης της τελείας είναι μια μεταβλητή, αποτελεί μια έγκυρη l- value.

- Αν p είναι μια έκφραση τύπου $t*$, τότε $*p$ είναι μια l-value με τύπο t, που αντιστοιχεί στο αντικείμενο όπου δείχνει ο δείκτης που παριστάνει η τιμή της p.

- Αν p είναι μια έκφραση τύπου $t*$ και e είναι μια έκφραση τύπου `int`, τότε $p[e]$ είναι μια l-value με τύπο t. Αυτή η l-value είναι ισοδύναμη με την l-value $*(p+e)$, όπως αυτή περιγράφεται στην ενότητα 3.4.3.

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

3.4.2 Σταθερές

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

- Οι ακέραιες σταθερές χωρίς πρόσημο, όπως περιγράφονται στην ενότητα 3.1. Έχουν τύπο `int` και η τιμή τους είναι ίση με τον μη αρνητικό ακέραιο αριθμό που παριστάνουν.

- Οι λέξεις κλειδιά `true` και `false`, με τύπο `bool` και προφανείς τιμές.

- Οι πραγματικές σταθερές χωρίς πρόσημο, όπως περιγράφονται στην ενότητα 3.1. Έχουν τύπο `double` 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 `NULL` που έχει τύπο t `*`, για κάθε έγκυρο τύπο t, και η τιμή της οποίας είναι ο μηδενικός δείκτης. Ο μηδενικός δείκτης απαγορεύεται να αποδεικτοδοτηθεί με χρήση του τελεστή `*`.

- Η λέξη κλειδί `nil` που έχει τύπο `obj`, για κάθε έγκυρη κλάση ή διαπροσωπεία, και η τιμή της οποίας είναι η μηδενική αναφορά σε κλάση ή διαπροσωπεία. Μαζί με την προηγούμενη σταθερά είναι οι μόνες εκφράσεις της Ooedsger που δεν έχουν μοναδικό τύπο.

- Οι σταθερές συμβολοσειρές, όπως περιγράφονται στην ενότητα 3.1. Έχουν τύπο `char *`. Κάθε τέτοια r-value είναι ένας δείκτης σε έναν πίνακα αντικείμενων τύπου `char`, 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.

3.4.3 Τελεστές

Οι τελεστές της Ooedsger διακρίνονται σε τελεστές με ένα, δύο και τρία τελούμενα. Από τους τελεστές με ένα τελούμενο, ορισμένοι γράφονται πρίν το τελούμενο (prefix) και ορισμένοι μετά (postfix). Οι τελεστές με δύο τελούμενα γράφονται πάντα μεταξύ των τελουμένων (infix) και η η αποτίμηση των τελουμένων γίνεται από αριστερά προς τα δεξιά. Ο μοναδικός τελεστής με τρία τελούμενα είναι ο `?:` και έχει ειδική σύνταξη και σημασιολογία, όπως θα εξηγηθεί παρακάτω.

Ήδη στην ενότητα 3.4.1 περιγράφηκε ο τελεστής αναφοράς σε στοιχείο πίνακα, η σύνταξη του οποίου αποτελεί εξαίρεση στα παραπάνω, ο τελεστής αποδεικτοδότησης `*`, καθώς και ο τελεστής της τελείας. Οι τρεις αυτοί τελεστές είναι οι μοναδικοί που έχουν ως αποτέλεσμα l-value. Στη συνέχεια περιγράφονται οι υπόλοιποι τελεστές της Ooedsger, που έχουν ως αποτέλεσμα r-value. Οι τελεστές αυτοί χωρίζονται σε δύο κατηγορίες: τους *τελεστές υπολογισμού* και τους *τελεστές ανάθεσης*. Οι πρώτοι χρησιμοποιούνται για τον υπολογισμό τιμών, ενώ οι δεύτεροι για την ανάθεση τιμών σε μεταβλητές. Η αποτίμηση μιας έκφρασης που περιέχει έναν ή περισσότερους τελεστές ανάθεσης, όπως στη C, εκτός από το να υπολογίζει μια τιμή αποτελέσματος έχει ως *παρενέργεια* τη μεταβολή των τιμών κάποιων μεταβλητών.

Οι τελεστές υπολογισμού είναι οι ακόλουθοι:

- Ο τελεστής `&` επιστρέφει τη διεύθυνση ενός αντικειμένου. Αν l είναι μια l-value τύπου t, τότε `&`l είναι μια r-value τύπου 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 `!` υλοποιεί τη λογική άρνηση. Το τελούμενό του πρέπει να είναι τύπου `bool`, και τον ίδιο τύπο έχει και το αποτέλεσμα.

- Οι τελεστές με δύο τελούμενα `+`, `-`, `*`, `/` και `%` υλοποιούν τις αριθμητικές πράξεις. Τα επιτρεπτά τελούμενα αυτών των τελεστών περιγράφονται παρακάτω:

  - Αν τα τελούμενα των +, -, *, / και % είναι τύπου int τότε και το αποτέλεσμα είναι τύπου int. Στην περίπτωση αυτή, οι τελεστές / και % υλοποιούν την ακέραια διαίρεση.
  - Αν τα τελούμενα των +, -, * και / είναι τύπου double τότε και το αποτέλεσμα είναι τύπου double. Στην περίπτωση αυτή, ο τελεστής / υλοποιεί την πραγματική διαίρεση.
  - Αν το πρώτο τελούμενο των + και - είναι τύπου t* και το δεύτερο τελούμενο είναι τύπου int, τότε το αποτέλεσμα είναι τύπου t*. Η τιμή του αποτελέσματος είναι ένας δείκτης μετατοπισμένος ως προς την τιμή του πρώτου τελούμενου κατά τόσες θέσεις αντικειμένων τύπου t όσες ορίζει η τιμή του δεύτερου τελουμένου (pointer arithmetic).

- Οι τελεστές == και != υλοποιούν αντίστοιχα την ισότητα και την ανισότητα. Το αποτέλεσμα είναι τύπου bool. Τα τελούμενα πρέπει να είναι του ίδιου τύπου.

- Οι τελεστές <, >, <= και >= υλοποιούν τις σχέσεις ανισότητας μεταξύ αριθμών ή δεικτών. Το αποτέλεσμα είναι τύπου bool. Τα τελούμενα πρέπει να είναι του ίδιου τύπου. Αν είναι αριθμητικά, η σύγκριση γίνεται βάσει των αριθμητικών τους τιμών. Αν είναι τύπου bool, η σύγκριση γίνεται θεωρώντας ότι false<true. Τέλος, αν είναι δείκτες, η σύγκριση έχει νόημα μόνο όταν τα τελούμενα περιέχουν διευθύνσεις αντικείμενων που βρίσκονται στον ίδιο μονοδιάστατο πίνακα και γίνεται βάσει της σχετικής θέσης αυτών των αντικειμένων μέσα στον πίνακα.

- Οι τελεστές && και || υλοποιούν αντίστοιχα τις πράξεις της λογικής σύζευξης και διάζευξης. Τα τελούμενα πρέπει να είναι τύπου bool 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.

- Ο τελεστής , (κόμμα) υπολογίζει τις τιμές των δύο τελουμένων και επιστρέφει την τιμή του δεύτερου, αγνοώντας αυτήν του πρώτου. Τα δύο τελούμενα μπορούν να είναι οποιουδήποτε έγκυρου τύπου. Το αποτέλεσμα είναι του ίδιου τύπου με αυτόν του δεύτερου τελούμενου.

- Ο τελεστής instanceof περιγράφηκε στην ενότητα 2.5 και διατίθεται μόνο για κλάσεις (όχι για διασυνδέσεις).

- Ο τελεστής με τρία τελούμενα ?: έχει την ειδική σύνταξη e ? e_1 : e_2. Το πρώτο τελούμενο πρέπει να είναι τύπου bool, ενώ τα άλλα δύο τελούμενα πρέπει να είναι του ίδιου τύπου. Το αποτέλεσμα έχει τον ίδιο τύπο με τα δύο τελευταία τελούμενα. Η αποτίμηση της έκφρασης e ? e_1 : e_2 γίνεται ξεκινώντας από την αποτίμηση της e. Αν η τιμή αυτής είναι true, τότε αποτιμάται η e_1 και η τιμή αυτής είναι το αποτέλεσμα. Διαφορετικά, αποτιμάται η e_2 και η τιμή αυτής είναι το αποτέλεσμα. Σε κάθε περίπτωση, αποτιμάται μόνο ένα από τα τελούμενα e_1 και e_2.

Οι τελεστές ανάθεσης είναι οι ακόλουθοι:

- Ο τελεστής = αναθέτει την τιμή του δεύτερου τελουμένου στη μεταβλητή που αντιστοιχεί στο πρώτο τελούμενο. Το πρώτο τελούμενο πρέπει να είναι μια l- value οποιουδήποτε έγκυρου τύπου t, ενώ το δεύτερο τελούμενο πρέπει να είναι του ίδιου τύπου t (εδώ ο t περιλαμβάνει και τους τύπους αναφορών σε κλάσεις ή διαπροσωπείες. Το αποτέλεσμα είναι και αυτό τύπου t και η τιμή του είναι η τιμή που ανατίθεται στη μεταβλητή. Αν τα δύο τελούμενα δεν είναι ίδιου τύπου, η ανάθεση επιτρέπεται στην περίπτωση που είναι

Πίνακας 3.2: Προτεραιότητα και προσεταιριστικότητα των τελεστών της Ooedsger.

Τελεστές	Περιγραφή	Αριθμός τελουμένων	Θέση και προσεταιριστικότητα
`.`	Τελεστής τελείας	2	infix, αριστερή
`[ ]` `( )`	Στοιχείο πίνακα, κλήση μεθόδου	2 ή >	ειδική
`++` `--`	Αύξηση και μείωση	1	postfix
`&` `*` `+` `-` `!`	Διεύθυνση αντικειμένου, αποδεικτοδότηση, πρόσημα, λογική άρνηση	1	prefix
`instanceof`	Τελεστής ελέγχου κλάσης	2	infix, αριστερή
`new` `delete` `destroy`	Δυναμική παραχώρηση μνήμης και αποδέσμευση	1	prefix
`++` `--`	Αύξηση και μείωση	1	prefix
`( )`	Μετατροπή τύπου	2	ειδική
`*` `/` `%`	Πολλαπλασιαστικοί τελεστές	2	infix, αριστερή
`+` `-`	Προσθετικοί τελεστές	2	infix, αριστερή
`==` `!=` `>` `<` `<=` `>=`	Σχεσιακοί τελεστές	2	infix, καμία
`&&`	Λογική σύζευξη	2	infix, αριστερή
`||`	Λογική διάζευξη	2	infix, αριστερή
`?:`	Τελεστής συνθήκης	3	ειδική
`=` `+=` `-=` `*=` `/=` `%=`	Τελεστές ανάθεσης	2	infix, δεξιά
`,`	Τελεστής παράθεσης	2	infix, αριστερή

εφικτή η αλλαγή τύπου από τον τύπο του δεύτερου τελούμενου στον τύπο του πρώτου, που περιγράφεται στην ενότητα 3.4.7. Αν ένα διατεταγμένο ζεύγος τύπων (t_1, t_2) πληροί τις παραπάνω προϋποθέσεις (όπου t_1, t_2 οι τύποι του πρώτου και δεύτερου τελούμενου αντίστοιχα), ορίζουμε ότι ο τύπος t_2 είναι *συμβατός για ανάθεση* στον τύπο t_1.

- Ο τελεστής *op*`=`, όπου $op \in \{$`+`, `-`, `*`, `/`, `%`$\}$, συνδυάζει υπολογισμό και ανάθεση. Η έκφραση *l op*`=` *e* είναι σημασιολογικά ισοδύναμη με *l* `=` *l op e*, με τη διαφορά ότι το τελούμενο *l* θα υπολογιστεί μόνο μια φορά.

- Οι τελεστές `++` και `--` δέχονται ένα τελούμενο και υπάρχουν σε δύο μορφές, μια prefix και μια postfix. Το τελούμενό τους πρέπει να είναι μια l-value αριθμητικού τύπου ή τύπου δείκτη. Το αποτέλεσμα είναι του ίδιου τύπου με το τελούμενο. Οι τελεστές προκαλούν την αύξηση ή τη μείωση της τιμής της l-value. Στην περίπτωση της μορφής prefix το αποτέλεσμα που υπολογίζεται είναι η τιμή της l-value μετά την αύξηση ή τη μείωση, ενώ στην περίπτωση της μορφής postfix το αποτέλεσμα είναι η τιμή της l-value πριν την αύξηση ή τη μείωση.

Στον πίνακα 3.2 ορίζεται η προτεραιότητα και η προσεταιριστικότητα των τελεστών της Ooedsger. Οι γραμμές που βρίσκονται υψηλότερα στον πίνακα περιέχουν τελεστές μεγαλύτερης προτεραιότητας. Τελεστές που βρίσκονται στην ίδια γραμμή έχουν την ίδια προτεραιότητα. Στον πίνακα συμπεριλαμβάνονται η κλήση μεθόδων, η μετατροπή τύπου και οι τελεστές δυναμικής παραχώρησης μνήμης, που περιγράφονται σε επόμενες ενότητες.

3.4.4 Κλήση μεθόδων

Αν f είναι το όνομα μιας μεθόδου με αποτέλεσμα τύπου t, τότε η έκφραση $f(e_1, \ldots, e_n)$ είναι μια r-value με τύπο t. Η αναζήτηση των μεθόδων ακολουθεί σε γενικές γραμμές το πρότυπο της

Java. Αρχικά η μέθοδος αναζητείται στην τρέχουσα κλάση, στη συνέχεια στις δευτερεύουσες κλάσεις και τέλος ελέγχεται αν πρόκειται για κλήση μεθόδου κατασκευαστή. Η αναζήτηση αυτή λαμβάνει υπόψη τις έννοιες της υπερφόρτωσης και της υπέρβασης μεθόδων. Τα κριτήρια με τα οποία γίνεται η αναζήτηση έχουν ως εξής:

Ο αριθμός των πραγματικών παραμέτρων n πρέπει να συμπίπτει με τον αριθμό των τυπικών παραμέτρων της f. Επίσης, αν οι τυπικές παράμετροι της μεθόδου προς κλήση δεν περιλαμβάνουν τύπους αναφορών σε κλάσεις ή διαπροσωπείες, 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.

- Αν η τυπική παράμετρος είναι τύπου t και περνά κατ' αξία, τότε η αντίστοιχη πραγματική παράμετρος πρέπει να είναι τύπου t.

- Αν η τυπική παράμετρος είναι τύπου t και περνά κατ' αναφορά, τότε η αντίστοιχη πραγματική παράμετρος πρέπει να είναι l-value τύπου t.

Αν οι τυπικές παράμετροι της μεθόδου προς κλήση περιλαμβάνουν και τύπους αναφορών σε κλάσεις ή διαπροσωπείες, τότε επιλέγεται ως μέθοδος προς κλήση εκείνη η $f(e_1, \ldots, e_n)$, της οποίας οι τύποι των τυπικών παραμέτρων βρίσκονται πιο "κοντά" στην ιεραρχία των κλάσεων και των διαπροσωπειών με τους τύπους των πραγματικών παραμέτρων. Αυτή η επιλογή προφανώς γίνεται με με βάση τα ορίσματα που έχουν ως τύπο έναν τύπο αναφοράς.

Καθώς μια μέθοδος μπορεί να επιστρέφει μια αναφορά σε ένα αντικείμενο, μπορούμε να καλέσουμε μεθόδους αυτού του αντικειμένου μέσα στην ίδια έκφραση. Επιτρέπεται δηλαδή η ένθεση μεθόδων με τη βοήθεια του τελεστή της τελείας, κατά τον ίδιο τρόπο που επιτρέπεται η ένθεση μεταβλητών, όπως παρουσιάστηκε στην ενότητα 3.4.1. Όπως είναι λογικό, υπάρχει η δυνατότητα να συνδυαστούν κλήσεις ένθετων μεθόδων και αναφορών μεταβλητών υπόστασης. Για παράδειγμα, παρακάτω ένα αντικείμενο `myShape` έχει μια μέθοδο με όνομα `procShape`. Αυτή επιστρέφει ένα αντικείμενο που έχει με τη σειρά του μια μεταβλητή υπόστασης `someRef` με τύπο μια αναφορά (σε αντικείμενο). Στη μεταβλητή αυτή έχει εκχωρηθεί ένα αντικείμενο που έχει μια μέθοδο με όνομα `myShape`, την οποία και καλούμε ως εξής:

```
myShape.procShape().someRef.myShape();
```

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

3.4.5 Κατασκευαστές, καταστροφείς, δυναμική διαχείριση μνήμης

Οι κλήσεις κατασκευαστών και καταστροφέων, καθώς και οι τελεστές `new` `delete` χρησιμοποιούνται για τη δυναμική διαχείριση μνήμης.

- Οι κατασκευαστές καλούνται για την κατασκευή νέων αντικειμένων της κλάσης στην οποία ορίζονται. Αρχικά δεσμεύουν μνήμη για το αντικείμενο της κλάσης και στη συνέχεια επιτελούν εργασίες πάνω σε αυτό, όπως είναι η αρχικοποίηση των μεταβλητών υπόστασης του αντικειμένου. Οι εργασίες αυτές πάντως καθορίζονται πλήρως από το σώμα των μεθόδων αυτών, όπως αυτό δίνεται από το χρήστη- προγραμματιστή. Το αποτέλεσμα της κλήσης είναι τύπου αναφοράς στο αντικείμενο που δημιουργήθηκε.

- Η κλήση του καταστροφέα για ένα αντικείμενο γίνεται αποκλειστικά με μια έκφραση `destroy` e, όπου e έκφραση τύπου αναφοράς σε έγκυρο αντικείμενο. Οι καταστροφείς καλούνται για να αποδεσμεύσουν τη μνήμη που κατείχε το αντικείμενο και να επιτελέσουν τις εργασίες που καθορίζονται από το σώμα των μεθόδων αυτών, όπως ένα κατάλληλο μήνυμα εξόδου. Η μνήμη που αποδεσμεύεται θα πρέπει προφανώς να έχει προηγουμένως

παραχωρηθεί με την κλήση μιας μεθόδου κατασκευαστή. Όπως προαναφέρθηκε στην ενότητα 3.3.3, αν δεν έχει δοθεί ο ορισμός μιας μεθόδου καταστροφέα για το αντικείμενο, χρησιμοποιείται ένας προκαθορισμένος (default) καταστροφέας για το αντικείμενο, με κενό σώμα. Το αποτέλεσμα της έκφρασης είναι τύπου `obj` και η τιμή του είναι πάντοτε `nil`.

- Η έκφραση `new` t `[`e`]`, όπου t οποιοσδήποτε έγκυρος τύπος t και e έκφραση τύπου `int`, προκαλεί τη δυναμική παραχώρηση μνήμης για τους υπόλοιπους τύπους εκτός των αναφορών σε αντικείμενα. Η έκφραση e καθώς και οι αγκύλες είναι προαιρετικές. Αν δίνονται, τότε η τιμή της e πρέπει να είναι ένας θετικός ακέραιος αριθμός n. Διαφορετικά, θεωρείται ότι $n = 1$. Το αποτέλεσμα είναι τύπου t`*`. Ο δείκτης αυτός τοποθετείται να δείχνει προς το πρώτο από n νέα δυναμικά αντικείμενα τύπου t.

- Η έκφραση `delete` e, όπου e έκφραση τύπου t`*`, προκαλεί την αποδέσμευση της μνήμης στην οποία δείχνει η τιμή της e. Η μνήμη αυτή πρέπει να έχει προηγουμένως παραχωρηθεί δυναμικά με χρήση του τελεστή `new`. Το αποτέλεσμα είναι τύπου t`*` και η τιμή του είναι πάντοτε `NULL`.

3.4.6 Δεσμευμένα αναγνωριστικά this και super

Η έκφραση `this` χρησιμοποιείται μέσα στο σώμα του ορισμού μιας μεθόδου για αναφορά στο τρέχον αντικείμενο (όντας το ίδιο μια τέτοια αναφορά), δηλαδή στο αντικείμενο με το οποίο κλήθηκε η μέθοδος. Αυτό μπορεί να χρησιμοποιηθεί σε περιπτώσεις, όπως η χρήση μεταβλητών υπόστασης αυτού του αντικειμένου ή για το πέρασμα του τρέχοντος αντικειμένου ως ορίσματος σε μια άλλη μέθοδο. Έτσι το `this` χρησιμοποιείται εκεί όπου κανονικά θα αναφερόμασταν στο όνομα ενός αντικειμένου. Παρακάτω δίνονται μερικά παραδείγματα χρήσης του `this`:

```
t = this.x;       // παίρνουμε τη μεταβλητή υπόστασης x για αυτό το αντικείμενο
this.m1(this);    // καλείται η μέθοδος \nc{m1} που ορίζεται στην τρέχουσα κλάση,
                  // και της δίνεται ως όρισμα το τρέχον αντικείμενο
return this;      // επιστρέφει το τρέχον αντικείμενο (σε συμφωνία με τον
                  // επιστρεφόμενο τύπο της τρέχουσας μεθόδου)
```

Στην πρώτη εντολή του παραδείγματος η χρησιμότητα του `this` είναι εμφανής, στην περίπτωση που μέσα στο σώμα της τρέχουσας μεθόδου (και πριν την εντολή αυτή) βρίσκεται μια δήλωση της μεταβλητής `x`, που επισκιάζει τη μεταβλητή υπόστασης `x` της κλάσης. Ο μόνος τρόπος τότε να αναφερθούμε στην τελευταία είναι με τη χρήση του `this`. Στη δεύτερη εντολή, όπως γίνεται εύκολα αντιληπτό, η χρήση του `this` ως πρώτο όρισμα του τελεστή της τελείας είναι περιττή, αφού ούτως ή άλλως θα κληθεί η η μέθοδος `m1` που ορίζεται στην τρέχουσα κλάση. Τέλος, επειδή το `this` είναι μια αναφορά στην τρέχουσα υπόσταση της κλάσης, μπορεί να χρησιμοποιείται μόνο μέσα στο σώμα του ορισμού μιας μεθόδου υπόστασης και όχι μέσα σε μια μέθοδο κλάσης (static).

Η έκφραση `super` αντίστοιχα χρησιμοποιείται μέσα στο σώμα του ορισμού μιας μεθόδου υπόστασης ως αναφορά στο αντικείμενο της υπερκλάσης του τρέχοντος αντικειμένου. Μπορεί να χρησιμοποιηθεί οπουδήποτε μπορεί και το `this`, αλλά το `super` αναφέρεται στην υπερκλάση αντί στο τρέχον αντικείμενο.

3.4.7 Αλλαγές τύπων

Οι αλλαγές τύπων υποστηρίζονται από τη γλώσσα Ooedsger στις ακόλουθες περιπτώσεις:

- Αλλαγή τύπου βασικών τύπων για όλα τα δυνατά ζεύγη αυτών εκτός από εκείνα που περιλαμβάνουν τον βασικό τύπο `bool`. Ο τύπος αυτός δεν επιτρέπεται να χρησιμοποιηθεί σε αλλαγή τύπου και οι τιμές τύπου `bool` πρέπει να είναι `true` ή `false`. Στις αλλαγές

τύπων ο προορισμός μπορεί να περιέχει μεγαλύτερες τιμές από την προέλευση, οπότε η αλλαγή τύπου μπορεί να γίνει χωρίς ρητή δήλωση της αλλαγής (χρήση παρενθέσεων), καθώς δεν υπάρχει απώλεια ακρίβειας. Για παράδειγμα:

```
int i = 8;
double d;
d = i;
```

Αντίθετα, αν ο προορισμός περιέχει μικρότερες τιμές από την προέλευση, πρέπει να χρησιμοποιηθεί σαφής αλλαγή τύπου με τη χρήση παρενθέσεων, αφού η μετατροπή θα έχει σαν αποτέλεσμα απώλεια ακρίβειας. Για παράδειγμα:

```
double x, y;
int i, j;
j = (int)x;
i = (int)(x/y);
```

Στην τελευταία εντολή του παραδείγματος χρησιμοποιήθηκαν παρενθέσεις και στην έκφραση `x/y`, επειδή η προτεραιότητα της αλλαγής τύπου είναι υψηλότερη από αυτή των προσθετικών και πολλαπλασιαστικών τελεστών. Αν δε χρησιμοποιηθούν παρενθέσεις εκεί, θα προκύψει λάθος, καθώς θα έχουμε πρώτα την αλλαγή τύπου του `x` σε ακέραιο και μετά τη διαίρεση αυτού με το `y`, οπότε θα προκύψει πραγματικός, που δεν μπορεί άμεσα να αποθηκευθεί σε έναν ακέραιο.

- Αλλαγή τύπου αναφορών αντικειμένων μεταξύ κλάσεων είτε προς τα πάνω είτε προς τα κάτω στην ιεραρχία των κλάσεων (upcasting, downcasting), όπως περιγράφεται στην ενότητα 2.5, καθώς επίσης αλλαγή τύπου από αναφορά αντικειμένου κλάσης σε αναφορά διαπροσωπείας ή από αναφορά διαπροσωπείας σε αναφορά άλλης διαπροσωπείας μόνο στην περίπτωση της προς τα πάνω αλλαγής (upcasting).

3.5 Εντολές

Οι εντολές που υποστηρίζει η γλώσσα Ooedsger είναι οι ακόλουθες:

- Η κενή εντολή `;` που δεν κάνει καμία ενέργεια.
- Η εντολή έκφρασης e `;` που αποτιμά την έκφραση e και αγνοεί την τιμή του αποτελέσματος.
- Η σύνθετη εντολή, που αποτελείται από μια σειρά έγκυρων εντολών ανάμεσα σε άγκιστρα { και }. Οι εντολές αυτές εκτελούνται διαδοχικά, εκτός αν κάποια είναι εντολή άλματος.
- Η εντολή ελέγχου `if (`e`)` s_1 `else` s_2. Η έκφραση e πρέπει να έχει τύπο `bool` και τα s_1, s_2 να είναι έγκυρες εντολές. Το τμήμα `else` είναι προαιρετικό.
- Η εντολή βρόχου `for (`e_1 `;` e_2 `;` e_3`)` s. Οι εκφράσεις e_1, e_2 και e_3 είναι προαιρετικές. Αν δίνεται, η έκφραση e_2 πρέπει να έχει τύπο `bool`, διαφορετικά θεωρείται ίση με `true`. Το s πρέπει να είναι έγκυρη εντολή. Η σημασιολογία της εντολής βρόχου είναι η ίδια με αυτή της C. Μπροστά από την εντολή βρόχου μπορεί να υπάρχει μια ετικέτα της μορφής I: το όνομα της οποίας πρέπει να είναι μοναδικό μέσα στην τρέχουσα μέθοδο.
- Οι εντολές άλματος `break` I`;` και `continue` I`;`, όπου το I είναι προαιρετικό. Αν δίνεται, το I πρέπει να είναι το όνομα μιας ετικέτας εντολής βρόχου μέσα στο σώμα της οποίας βρίσκεται η εντολή άλματος. Αν το I παραλείπεται, οι εντολές αυτές έχουν την ίδια

σημασιολογία με τη γλώσσα C, δηλαδή προκαλούν τον τερματισμό ή τη συνέχιση του βρόχου μέσα στον οποίο βρίσκονται. Αν το I δίνεται, προκαλείται ο τερματισμός ή η συνέχιση του βρόχου που φέρει την ετικέτα I.

- Η εντολή άλματος `return` e `;` που τερματίζει την εκτέλεση της τρέχουσας μεθόδου και επιστρέφει την τιμή της e ως αποτέλεσμα της μεθόδου. Αν η τρέχουσα μέθοδος έχει τύπο αποτελέσματος `void` ή `constructor` ή `destructor` τότε η έκφραση e πρέπει να παραλείπεται. Διαφορετικά, η έκφραση e πρέπει να έχει τύπο ίδιο με τον τύπο αποτελέσματος της τρέχουσας μεθόδου.

3.6 Οδηγίες προς το μεταγλωττιστή

Η μοναδική οδηγία, που επιτρέπει ο μεταγλωττιστής της γλώσσας Ooedsger είναι η οδηγία `#include`. Η οδηγία αυτή έχει την ίδια σημασιολογία όπως και στη γλώσσα C, δηλαδή επιτρέπει την ανάγνωση ενός εξωτερικού αρχείου σαν αυτό να ήταν τμήμα του προγράμματος. Πρέπει να βρίσκεται υποχρεωτικά στην αρχή της γραμμής (να μην προηγούνται κενά διαστήματα). Μετά την οδηγία `#include` ακολουθεί μια σταθερή συμβολοσειρά που περιέχει το όνομα του αρχείου το οποίο θα συβμπεριληφθεί σε αυτό το σημείο.

Η οδηγία αυτή απευθύνεται ουσιαστικά στο λεκτικό αναλυτή. Σε περίπτωση που αυτός συναντήσει αυτή την οδηγία, θα πρέπει να σταματήσει την ανάγνωση του αρχείου προγράμματος και να συνεχίσει με την επεξεργασία του αρχείου που ζητείται να συμπεριληφθεί. Μετά το τέλος αυτού του αρχείου, ο λεκτικός αναλυτής πρέπει να συνεχίσει από το σημείο του αρχείου προγράμματος, στο οποίο είχε σταματήσει. Τα αρχεία που διαβάζονται με την οδηγία `#include` μπορούν να περιέχουν και αυτά τέτοιες οδηγίες. Φυσικά, μεμονωμένες λεκτικές μονάδες καθώς και σχόλια πρέπει να περιέχονται πλήρως σε ένα αρχείο προγράμματος (δεν επιτρέπεται να αρχίζουν σε ένα αρχείο προγράμματος και να τελειώνουν σε κάποιο άλλο).

3.7 Βιβλιοθήκη έτοιμων συναρτήσεων

Η Ooedsger υποστηρίζει ένα σύνολο προκαθορισμένων μεθόδων, οι οποίες έχουν υλοποιηθεί σε assembly του 80x86 ως μια *βιβλιοθήκη έτοιμων συναρτήσεων* (run-time library). Οι επικεφαλίδες αυτών των μεθόδων βρίσκονται σε μια προκαθορισμένη κλάση `System`, η οποία δημιουργείται αυτόματα για κάθε πρόγραμμα εισόδου. Έτσι, η βιβλιοθήκη έτοιμων συναρτήσεων χρησιμοποιείται κατά τη συνένωση (linking) του τελικού κώδικα και οι μέθοδοι που περιέχει βρίσκονται στη διάθεση του προγραμματιστή, μέσω της κλάσης `System` ως στατικές μέθοδοι. Για την κλήση τους χρησιμοποιείται ο τελεστής της τελείας με πρώτο τελούμενο το όνομα της κλάσης `System` και δεύτερο το όνομα της μεθόδου.

Παρακάτω δίνονται οι δηλώσεις τους, όπως βρίσκονται στην κλάση `System`, και εξηγείται η λειτουργία τους.

3.7.1 Είσοδος και έξοδος

```
void writeInteger (int n);
void writeBoolean (bool b);
void writeChar    (char c);
void writeReal    (double d);
void writeString  (char * s);
```

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

```
int    readInteger ();
bool   readBoolean ();
```

```
char   readChar    ();
double readReal    ();
void   readString  (int size, char * s);
```

Αντίστοιχα, οι παραπάνω μέθοδοι χρησιμοποιούνται για την εισαγωγή τιμών που ανήκουν στους βασικούς τύπους της Ooedsger και για την εισαγωγή συμβολοσειρών. Η συνάρτηση **readString** χρησιμοποιείται για την ανάγνωση μιας συμβολοσειράς μέχρι τον επόμενο χαρακτήρα αλλαγής γραμμής. Οι παράμετροι 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.

3.7.2 Μαθηματικές συναρτήσεις

```
int    abs  (int n);
double fabs (double d);
```

Η απόλυτη τιμή ενός ακέραιου ή πραγματικού αριθμού.

```
double sqrt (double d);
double sin  (double d);
double cos  (double d);
double tan  (double d);
double atan (double d);
double exp  (double d);
double ln   (double d);
double pi   ();
```

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

3.7.3 Συναρτήσεις μετατροπής

```
int trunc (double d);
int round (double d);
```

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

```
int  ord (char c);
char chr (int n);
```

Μετατρέπουν από ένα χαρακτήρα στον αντίστοιχο κωδικό ASCII και αντίστροφα.

3.7.4 Συναρτήσεις διαχείρισης συμβολοσειρών

```
int  strlen (char * s);
int  strcmp (char * s1, char * s2);
void strcpy (char * trg, char * src);
void strcat (char * trg, char * src);
```

Οι συναρτήσεις αυτές έχουν ακριβώς την ίδια λειτουργία με τις συνώνυμές τους στη βιβλιοθήκη συναρτήσεων της γλώσσας C.

3.8 Πλήρης γραμματική της Ooedsger

Η σύνταξη της γλώσσας Ooedsger 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 ⟨I⟩, ⟨int-const⟩, ⟨double-const⟩, ⟨char-const⟩ και ⟨string-literal⟩ είναι τερματικά σύμβολα της γραμματικής.

```
⟨program⟩              ::= ( ⟨declaration⟩ )+
⟨declaration⟩          ::= (“class” | “interface”) ⟨I⟩ [ “extends” ⟨I⟩ ]
                             [ “implements” ⟨I⟩ ( “,” ⟨I⟩ )* ]
                             “{” ( ⟨mf-declaration⟩ )* “}”
⟨mf-declaration⟩       ::= ⟨field-declaration⟩ | ⟨method-declaration⟩ | ⟨method-definition⟩
⟨field-declaration⟩    ::= [ “modifier” ] ⟨variable-declaration⟩
⟨method-declaration⟩   ::= [ “modifier” ] ⟨result-type⟩ ⟨I⟩ “(” [ ⟨parameter-list⟩ ] “)” “;”
⟨method-definition⟩    ::= [ “modifier” ] ⟨result-type⟩ ⟨I⟩ “(” [ ⟨parameter-list⟩ ] “)”
                             “{” ( ⟨variable-declaration⟩ | ⟨statement⟩ )* “}”
⟨modifier⟩             ::= “static” | “abstract”
⟨variable-declaration⟩ ::= ⟨type⟩ ⟨declarator⟩ ( “,” ⟨declarator⟩ )* “;”
⟨type⟩                 ::= ⟨basic-type⟩ ( “*” )* | ⟨I⟩
⟨basic-type⟩           ::= “int” | “char” | “bool” | “double”
⟨result-type⟩          ::= ⟨type⟩ | “void” | “constructor” | “destructor”
⟨declarator⟩           ::= ⟨I⟩ [ “[” ⟨constant-expression⟩ “]” ] [ “=” ⟨expression⟩ ]
⟨parameter-list⟩       ::= ⟨parameter⟩ ( “,” ⟨parameter⟩ )*
⟨parameter⟩            ::= [ “byref” ] ⟨type⟩ ⟨I⟩
⟨statement⟩            ::= “;” | ⟨expression⟩ “;” | “{” ( ⟨variable-declaration⟩ | ⟨statement⟩ )* “}”
                        |  “if” “(” ⟨expression⟩ “)” ⟨statement⟩ [ “else” ⟨statement⟩ ]
                        |  [ ⟨I⟩ “:” ] “for” “(” [ ⟨expression⟩ ] “;” [ ⟨expression⟩ ] “;”
                             [ ⟨expression⟩ ] “)” ⟨statement⟩
                        |  “continue” [ ⟨I⟩ ] “;” | “break” [ ⟨I⟩ ] “;” | “return” [ ⟨expr⟩ ] “;”
⟨expression⟩           ::= ⟨I⟩ | “(” ⟨expression⟩ “)” | “true” | “false” | “NULL” | “nil”
                        |  ⟨int-const⟩ | ⟨char-const⟩ | ⟨double-const⟩ | ⟨string-literal⟩
                        |  ⟨I⟩ “(” [ ⟨expression-list⟩ ] “)” | ⟨expression⟩ “[” ⟨expression⟩ “]”
                        |  ⟨unary-operator⟩ ⟨expression⟩ | ⟨expression⟩ ⟨binary-operator⟩ ⟨expression⟩
                        |  ⟨unary-assignment⟩ ⟨expression⟩ | ⟨expression⟩ ⟨unary-assignment⟩
                        |  ⟨expression⟩ ⟨binary-assignment⟩ ⟨expression⟩
                        |  “(” ⟨basic-type⟩ ( “*” )* “)” ⟨expression⟩
                        |  ⟨expression⟩ “?” ⟨expression⟩ “:” ⟨expression⟩
                        |  “new” ⟨new-specifier⟩ | “delete” ⟨expression⟩ | “destroy” ⟨expression⟩
                        |  “this” | “super” | ⟨expression⟩ “.” ⟨I⟩
                        |  ⟨expression⟩ “.” ⟨I⟩ “(” [ ⟨expression-list⟩ ] “)”
                        |  ⟨expression⟩ “instanceof” ⟨I⟩
⟨new-specifier⟩        ::= ⟨basic-type⟩ [ “[” ⟨expression⟩ “]” ]
                        |  “(” ⟨basic-type⟩ ( “*” )* “)” [ “[” ⟨expression⟩ “]” ]
⟨expression-list⟩      ::= ⟨expression⟩ ( “,” ⟨expression⟩ )*
⟨constant-expression⟩  ::= ⟨expression⟩
⟨unary-operator⟩       ::= “&” | “*” | “+” | “-” | “!”
⟨binary-operator⟩      ::= “*” | “/” | “%” | “+” | “-” | “<” | “>” | “<=” | “>=” | “==” | “!=”
                        |  “&&” | “||” | “,”
⟨unary-assignment⟩     ::= “++” | “--”
⟨binary-assignment⟩    ::= “=” | “*=” | “/=” | “%=” | “+=” | “-=”
```

Κεφάλαιο 4

Μεταγλώττιση της γλώσσας Ooedsger

Για τη μεταγλώττιση της Ooedsger 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

Κατά τη 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 Ooedsger 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 Ooedsger 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. Για το σκοπό αυτό χρησιμοποιείται ο μηχανισμός των *αρχικών καταστάσεων* (initial states) του `flex` στην περίπτωση των σχολίων πολλών γραμμών, που επιτρέπεται να είναι και φωλιασμένα, όπως ελέχθη στην ενότητα 3.1. Χρήση αυτού του μηχανισμού γίνεται και για την υποστήριξη της οδηγίας `#include`, που περιγράφεται στην ενότητα 3.6. Τέλος, κάθε φορά που ο λεκτικός αναλυτής αναγνωρίζει μια λεκτική μονάδα και πριν την επιστρέψει στο συντακτικό αναλυτή, ενημερώνει τη μεταβλητή `yylval` με τη σημασιολογική τιμή που αντιστοιχεί σε αυτό το τερματικό σύμβολο. Μέσω της μεταβλητής αυτής επιτυγχάνεται η επικοινωνία του λεκτικού αναλυτή 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 Ooedsger, της οποίας η σύνταξη ορίζεται από τη *γραμματική χωρίς συμφραζόμενα*[1] (context-free grammar) της γλώσσας, που περιγράφεται στην ενότητα 3.8. 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).

Όπως είναι λογικό, 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:

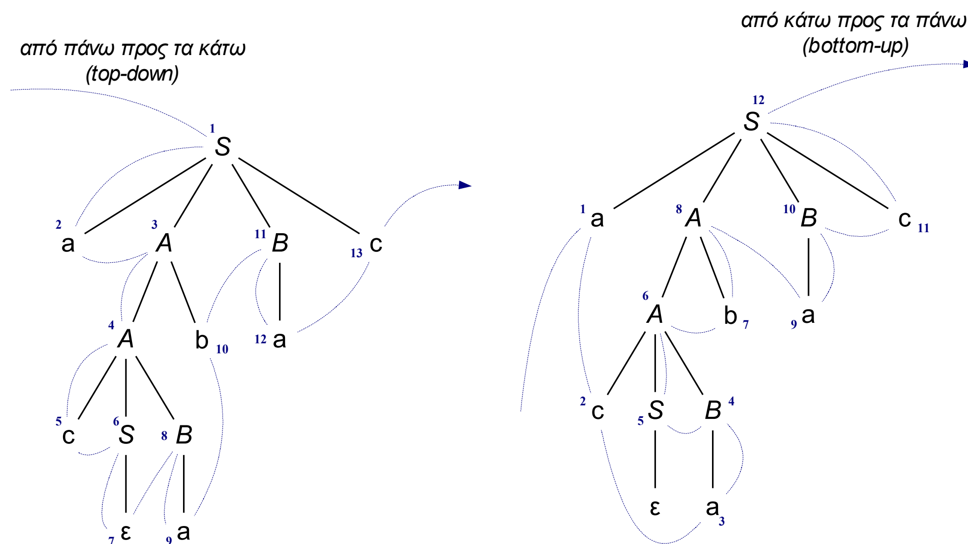
- $T = \{\,\mathrm{a}, \mathrm{b}, \mathrm{c}\,\}$
- $N = \{\,S, A, B\,\}$
- $P = \left\{ \begin{array}{lll} S \rightarrow \mathrm{a}AB\mathrm{c} & A \rightarrow \mathrm{c}SB & B \rightarrow \mathrm{b}B \\ S \rightarrow \epsilon & A \rightarrow A\mathrm{b} & B \rightarrow \mathrm{a} \end{array} \right\}$

Έστω τώρα μια παραγωγή αυτής της γραμματικής:

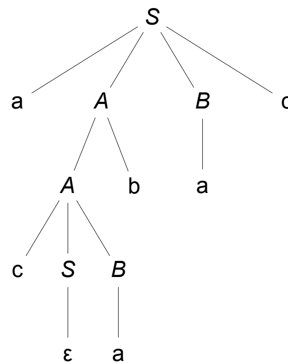
$$S \Rightarrow \mathrm{a}AB\mathrm{c} \Rightarrow \mathrm{a}A\mathrm{b}B\mathrm{c} \Rightarrow \mathrm{ac}SB\mathrm{b}B\mathrm{c} \Rightarrow \mathrm{ac}S\mathrm{ab}B\mathrm{c} \Rightarrow \mathrm{acab}B\mathrm{c} \Rightarrow \mathrm{acabac}$$

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 $A \rightarrow \alpha$, 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 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 συμβαίνει, όταν αναγνωσθεί κάποια λεκτική μονάδα που δε συμφωνεί με την υπάρχουσα γραμματική, όπως αυτή περιγράφεται στους κανόνες παραγωγής του μεταπρογράμματος. Τότε καλείται αυτόματα η συνάρτηση yyerror του συντακτικού αναλυτή, για να εκτυπώσει 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 υλοποιείται 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. Ενδεικτικά κάποια είδη ονομάτων που μπορούν να εμφανίζονται σε προγράμματα της γλώσσας Ooedsger είναι τα παρακάτω:

- κλάσεις ή διαπροσωπείες,
- μεταβλητές,
- μέθοδοι,
- παράμετροι μεθόδων,
- ετικέτες για την εντολή for, στις οποίες επιτρέπεται η μετάβαση με εντολές όπως η continue$I$,
- 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 λίγο πριν και κατά τη διάρκεια της σημασιολογικής ανάλυσης. Συγκεκριμένα, πριν την έναρξη της ανάλυσης αυτής τοποθετούνται στον πίνακα συμβόλων οι πληροφορίες σχετικά με τις μεθόδους κατασκευαστές των κλάσεων, ώστε αυτές οι μέθοδοι να είναι άμεσα προσεγγίσιμες μέσα από οποιαδήποτε κλάση και όχι μόνο μέσα στην κλάση όπου ορίζονται. Στη συνέχεια, συλλέγονται οι πληροφορίες σχετικά με τις κλάσεις και τις διαπροσωπείες, τις μεταβλητές υπόστασης και κλάσης και τις μεθόδους υπόστασης και κλάσης, ώστε να είναι ορατές στη συνέχεια, στη σημασιολογική ανάλυση, σε οποιοδήποτε σημείο του προγράμματος, όπως φαίνεται και στο πρώτο παράδειγμα της ενότητας 3.3.1. Έπειτα ξεκινά η σημασιολογική ανάλυση, στη διάρκεια της οποίας ελέγχεται η ορθότητα των αποθηκευμένων πληροφοριών με τη βοήθεια του συντακτικού δέντρου (με νέα διάσχισή του), αλλά και η παραπέρα ενημέρωση του πίνακα συμβόλων με τις παραμέτρους, τις τοπικές μεταβλητές των μεθόδων, καθώς και με τις τυχόν ετικέτες της εντολής `for`.

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 [1].

Στην περίπτωση που η αναζήτηση ενός ονόματος γίνει με επιτυχία, γίνονται διαθέσιμες οι πληροφορίες που αντιστοιχούν σε αυτό το όνομα.

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 δυο 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.

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 Ooedsger, 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 ροής*, π.χ. ο μεταγλωττιστής θα πρέπει να διαγνώσει σφάλμα αν χρησιμοποιείται μια εντολή εξόδου από κάποια δομή ελέγχου (όπως η `break`) έξω από την αντίστοιχη δομή.
- *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 ή αν σε μια δομή `for` υπάρχουν δυο ετικέτες με την ίδια τιμή.

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.

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.

Η τεχνική της *μετάφρασης οδηγούμενης από τη σύνταξη* (syntax-directed translation) θεωρείται σήμερα η πλέον κατάλληλη για την παραγωγή ενδιάμεσου κώδικα.[2] Η τεχνική αυτή βασίζεται στη σύνταξη της αρχικής γλώσσας προγραμματισμού και παρέχει ένα πλαίσιο για την παραγωγή ενδιάμεσου κώδικα. Στην γλώσσα Ooedsger χρησιμοποιούμε αυτήν την τεχνική για την παραγωγή ενδιάμεσου κώδικα παράλληλα με τη σημασιολογική ανάλυση (χωρίς την εμπλοκή του συντακτικού αναλυτή). Έτσι, όταν ολοκληρώνεται ο σημασιολογικός έλεγχος για κάποιο τμήμα του συντακτικού δέντρου, παράγεται ο αντίστοιχος κώδικας για το τμήμα αυτό μέσα στις κατάλληλες σημασιολογικές ρουτίνες. Στην περίπτωσή μας δηλαδή δεν τροποποιούμε το συντακτικό αναλυτή για την παραγωγή ενδιάμεσου κώδικα, κάτι που εμφανίζεται συχνά στην περίπτωση της μετάφρασης οδηγούμενης από τη σύνταξη.

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

$n\!: \ op, x, y, z$

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

[2] Εναλλακτική είναι η τεχνική της *μετάφρασης οδηγούμενης από τη σημασιολογία* (semantics-directed translation), που βασίζεται σε μια κατάλληλη τυπική περιγραφή της σημασιολογίας της αρχικής γλώσσας προγραμματισμού.

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.

Στο σημείο αυτό είναι απαραίτητη η μετατροπή των γλωσσικών δομών της Ooedsger σε ενδιάμεσο κώδικα με τη βοήθεια ενός σχεδίου, 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 [1]. Ενδεικτικά εδώ θα δοθούν τα τελούμενα και οι τελεστές της γλώσσας των τετράδων για τη γλώσσα Ooedsger, η μορφή των τετράδων αυτών και μια μικρή περιγραφή της λειτουργίας κάθε τέτοιας τετράδας, με αναφορές στις διαφοροποιήσεις που εισήχθησαν λόγω της γλώσσας Ooedsger. Χρειάστηκαν επίσης κάποιες προσθήκες στο σχέδιο αναφοράς [1] για την παραγωγή του ενδιάμεσου κώδικα, οι οποίες όμως δεν οδήγησαν στη δημιουργία νέων τετράδων, αλλά υλοποιήθηκαν με χρήση των ήδη υπαρχουσών τετράδων (π.χ. δομή `for` της Ooedsger σε σχέση με τη δομή `while` του σχεδίου αναφοράς). Χάριν απλότητας, αυτές δεν αναφέρονται παρακάτω.

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 Ooedsger, οι σταθερές μπορούν να είναι ακέραιες, πραγματικές, λογικές, χαρακτήρες, συμβολοσειρές, `NULL` ή `nil`.

- *Όνομα* κλάσης, μεταβλητής, παραμέτρου ή μεθόδου. 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.

- *Αποτέλεσμα μεθόδου*, που συμβολίζεται με $$. Αυτό το τελούμενο χρησιμεύει για την αποθήκευση του αποτελέσματος της τρέχουσας δομικής μονάδας, και φυσικά μπορεί να χρησιμοποιηθεί μόνο όταν αυτή είναι μέθοδος που επιστρέφει αποτέλεσμα (όχι `void`).

Οι παραπάνω μορφές τελουμένων ονομάζονται *απλές*. Σε κάθε ένα απλό τελούμενο μπορεί κανείς να αντιστοιχίσει ένα τύπο δεδομένων της γλώσσας Ooedsger, 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 Ooedsger.

- *Αποδεικτοδότηση.* Αν x είναι ένα απλό τελούμενο τύπου 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 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.

- *Ετικέτα εντολής*, η οποία χρησιμοποιείται για άλματα με τις εντολές `break` I και `continue` I σε ετικέτες της εντολής `for`. Οι ετικέτες αυτές φυλάσσονται επίσης στον πίνακα συμβόλων, οπότε ως τελούμενα δε διαφοροποιούνται από τα λοιπά ονόματα.

- *Τρόπος περάσματος* παραμέτρου σε μέθοδο. Η Ooedsger 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 μεθόδου μη `void` 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 $\text{unit}, I, -, -$
  και $\text{endu}, I, -, -$

  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. Οι τετράδες αυτές, για λόγους απλοποίησης, δεν περιέχουν το όνομα της κλάσης στην οποία ανήκει η δομική μονάδα (μέθοδος). Η πληροφορία όμως αυτή υπάρχει στο τελούμενο I (δείκτης στην αντίστοιχη εγγραφή στον πίνακα συμβόλων) και χρησιμοποιείται αργότερα κατά τον τελικό κώδικα.

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

  Οι τετράδες αυτές αντιστοιχούν στις πράξεις της πρόσθεσης, της αφαίρεσης, του πολλαπλασιασμού, της διαίρεσης και του υπόλοιπου της ακέραιας διαίρεσης. Το αποτέλεσμα της πράξης $x\ op\ y$ υπολογίζεται και αποθηκεύεται στο τελούμενο z. Πρέπει να σημειωθεί

⟨program⟩	::=	(⟨quadruple⟩)*
⟨quadruple⟩	::=	⟨label⟩ “:” ⟨opname⟩ “,” ⟨operand⟩ “,” ⟨operand⟩ “,” ⟨operand⟩
⟨label⟩	::=	⟨integer-const⟩
⟨opname⟩	::=	“**unit**” | “**endu**” | “+” | “−” | “∗” | “/” | “%”
	|	“:=” | “**array**” | “=” | “<>” | “>” | “<”
	|	“>=” | “<=” | “**ifb**” | “**label**” | “**jumpl**”
	|	“**call**” | “**par**” | “**ret**” | “**instanceof**” | “**dot**”
⟨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.

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 sec:ooedsger:exprs:operators).

- 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 `int`. 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 $\mathrm{ifb}, x, -, z$

  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 $\mathrm{jump}, -, -, z$

  Η τετράδα αυτή πραγματοποιεί άλμα στην τετράδα με ετικέτα z.

- Τετράδα $\text{label}, I, -, -$

  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 jump, αφού βρεθεί ο αριθμός της τετράδας label.

- Τετράδα $\text{call}, -, I_1, I_2$

  Με αυτή την τετράδα γίνεται η κλήση της μεθόδου με όνομα I_2 για το αντικείμενο (για την ακρίβεια, την αναφορά στο αντικείμενο) με όνομα I_1 ή για την κλάση με το όνομα αυτό (αν η μέθοδος είναι στατική). Αν δε δίνεται το I_1 (στη θέση του υπάρχει -), τότε έχουμε την περίπτωση κλήσης των υποπρογραμμάτων βιβλιοθήκης `_new` και `_dispose` λόγω χρήσης των εντολών `new` και `delete` αντίστοιχα. Αυτά τα υποπρογράμματα (`_new` και `_dispose`) δεν μπορούν να κληθούν άμεσα από το αρχικό πρόγραμμα, καθώς τα ονόματά τους δεν είναι έγκυρα ονόματα μεθόδων της Ooedsger. Οι επικεφαλίδες τους σε μορφή που μοιάζει με Ooedsger είναι οι ακόλουθες:

  ```
  T  _new     (int size);
  T  _dispose (byref T* pointer);
  ```

  Η κλήση στη `_new` έχει ως αποτέλεσμα τη δυναμική παραχώρηση μνήμης, το μέγεθος της οποίας σε bytes καθορίζεται από την τιμή της παραμέτρου. Το αποτέλεσμα της συνάρτησης είναι ένας δείκτης προς την παραχωρηθείσα μνήμη. Αντίθετα, η κλήση στην `_dispose` έχει ως αποτέλεσμα την αποδέσμευση της μνήμης όπου δείχνει η παράμετρος. Ο τύπος `T` και στις δυο επικεφαλίδες είναι αυθαίρετος.

- 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.[3]

  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 $\text{instof}, I_1, I_2, -$

  Η τετράδα αυτή χρησιμοποιείται για τη λειτουργία του αντίστοιχου τελεστή `instanceof` της Ooedsger, όπου το τελούμενο I_1 είναι το όνομα μιας αναφοράς προς ένα αντικείμενο και το I_2 είναι ένα όνομα κλάσης. Εδώ ακολουθήθηκε ανάλογο σχέδιο παραγωγής ενδιάμεσου κώδικα με τους υπόλοιπους (π.χ. `>=`) σχεσιακούς τελεστές [1].

[3] Στην περίπτωση που η τρέχουσα δομική μονάδα είναι μέθοδος μη `void`, η τιμή του αποτελέσματος θα πρέπει να έχει αποθηκευτεί στη θέση του αποτελέσματος με μια ανάθεση στο ειδικό τελούμενο $$.

- Τετράδα $\mathrm{dot}, I_1, I_2, I_3$

  Η τετράδα αυτή χρησιμοποιείται για τη λειτουργία του τελεστή της τελείας της Ooedsger στην περίπτωση της χρήσης του για αναφορά σε μεταβλητές. Η αναφορά σε μεθόδους (με τον τελεστή αυτό) καλύπτεται στην τετράδα call. Εδώ το τελούμενο I_1 είναι το όνομα μιας αναφοράς προς ένα αντικείμενο ή το όνομα μιας κλάσης (μεταβλητή `static`), το I_2 είναι το όνομα της μεταβλητής στην οποία θέλουμε να αναφερθούμε μέσω του I_1 και I_3 είναι μια προσωρινή μεταβλητή, στην οποία ανατίθεται η τιμή της μεταβλητής I_2, ώστε να επιτρέπεται η ένθετη προσπέλαση μεταβλητών μέσω πολλαπλών χρήσεων του τελεστή της τελείας στην ίδια έκφραση, όπως εξηγείται στην ενότητα 3.4.1.

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 Ooedsger 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 MASM. Για λόγους απλότητας, το τελικό πρόγραμμα θα ακολουθεί το μοντέλο εκτέλεσης `.COM`, που προσφέρεται για την υλοποίηση μικρών προγραμμάτων. Τα κυριότερα χαρακτηριστικά αυτού του μοντέλου, που είναι επίσης γνωστό ως *μικροσκοπικό μοντέλο* (tiny model), είναι τα εξής:

- Ολόκληρο το πρόγραμμα περιέχεται σε ένα τμήμα μνήμης, το οποίο χρησιμοποιείται για κώδικα, δεδομένα και στοίβα. Αυτό σημαίνει ότι οι καταχωρητές cs, ds και ss περιέχουν την ίδια τιμή. Το γεγονός αυτό διευκολύνει τις αναφορές στη μνήμη, εφόσον δε χρειάζεται να καθορίζεται το τμήμα, στο οποίο περιέχεται κάθε δεδομένο. Επιβάλλει όμως τον περιορισμό ότι ο κώδικας του προγράμματος, τα δεδομένα και η στοίβα περιέχονται σε 64 KB.

- Η πρώτη εκτελέσιμη εντολή του προγράμματος βρίσκεται στη διεύθυνση `100h` (μέσα στο μοναδικό τμήμα). Οι διευθύνσεις που προηγούνται αυτής χρησιμοποιούνται από το λειτουργικό σύστημα. Η αρχική τιμή του καταχωρητή sp είναι `FFFEh` και τα περιεχόμενα της βάσης της στοίβας είναι μηδενικά.

4.6.1 Εγγράφημα δραστηριοποίησης και οργάνωση της μνήμης

Κατά την παραγωγή του τελικού κώδικα για την Ooedsger, τα τοπικά δεδομένα μιας μεθόδου 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 των παραμέτρων, αποτελεσμάτων, πληροφοριών κατάστασης μηχανής, του τρέχοντος αντικειμένου, των τοπικών μεταβλητών, των 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 τον 8086 ο δείκτης πλαισίου τοποθετείται στον καταχωρητή bp. Επομένως, οι πληροφορίες που περιέχονται στο ΕΔ μπορούν να προσπελαστούν στη συμβολική γλώσσα

	Παράμετρος 1
...	...
bp+10	Παράμετρος n
bp+8	Διεύθυνση αποτελέσματος
bp+6	Διεύθυνση “this”
bp+4	Σύνδεσμος προσπέλασης
bp+2	Διεύθυνση επιστροφής
bp	Προηγούμενο bp
bp–...	Τοπική μεταβλητή 1
...	...
	Τοπική μεταβλητή m
	Προσωρινή μεταβλητή 1
	...
	Προσωρινή μεταβλητή k

Σχήμα 4.3: Πληροφορίες που αποθηκεύονται σε ένα ΕΔ.

του 8086 μέσω ορισμάτων της μορφής [bp + *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 Ooedsger, οι πληροφορίες που αποθηκεύονται σε κάθε ΕΔ είναι οι εξής:

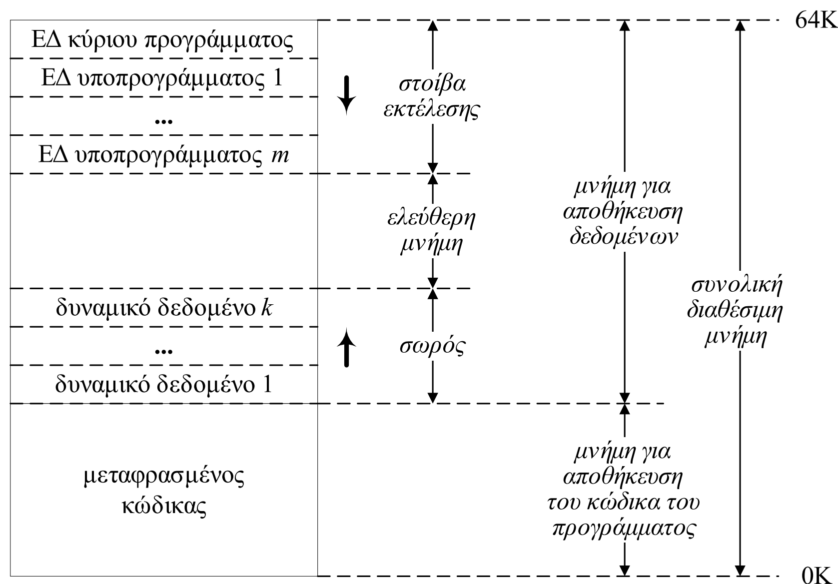
- Στις πρώτες θέσεις τοποθετούνται οι πραγματικές παράμετροι της αντίστοιχης κλήσης της μεθόδου.
- Στη συνέχεια σε ένα σταθερό τμήμα, με την έννοια ότι η μορφή του είναι κοινή για όλες τις κλήσεις μεθόδων, αποθηκεύονται η διεύθυνση του αποτελέσματος, η διεύθυνση του τρέχοντος αντικειμένου για το οποίο κλήθηκε η μέθοδος (εκτός αν πρόκειται για στατική μέθοδο ή για μέθοδο από τη βιβλιοθήκη χρόνου εκτέλεσης ή για κάποιο από τα ειδικά υποπρογράμματα βιβλιοθήκης `_new` και `_dispose`, οπότε απουσιάζει αυτή η διεύθυνση[4]), ο σύνδεσμος προσπέλασης (περιγράφεται στην ενότητα 9.3.4 του βιβλίου αναφοράς, δε χρησιμοποιείται στην παρούσα υλοποίηση, αλλά αφήνεται για λόγους απλότητας και επεκτασιμότητας), 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.

Η μνήμη του υπολογιστή κατά το χρόνο εκτέλεσης μπορεί να χωριστεί σε δύο περιοχές, όπως φαίνεται στο σχήμα 4.4 που εξειδικεύεται για το μεταγλωττιστή της Ooedsger και το μοντέλο εκτέλεσης `.COM`.

Η πρώτη περιοχή χρησιμοποιείται για την αποθήκευση του τελικού κώδικα που αντιστοιχεί στο αρχικό πρόγραμμα και το μέγεθός της είναι σταθερό και γνωστό κατά τη μεταγλώττιση. Η δεύτερη περιοχή χρησιμοποιείται για την αποθήκευση των ΕΔ, δηλαδή των τελούμενων του αρχικού προγράμματος. Τα ΕΔ τοποθετούνται σ' αυτή την περιοχή κατά την εκτέλεση του προγράμματος. Πρώτα τοποθετείται το ΕΔ του κυρίου προγράμματος. Μετά, κάθε φορά που

[4] Επίσης στην περίπτωση των μεθόδων κατασκευαστών, η θέση αυτή περιέχει μια αναφορά (reference) στην κλάση μέσα στην οποία ορίζονται, αφού στη μέθοδο αυτή δεν μπορεί να χρησιμοποιηθεί το τρέχον αντικείμενο για το οποίο αυτές καλούνται, αφού αυτό δεν έχει ακόμη δημιουργηθεί. Στην περίπτωση των καταστροφέων, το τρέχον αντικείμενο προς καταστροφή περνά ούτως ή άλλως ως παράμετρος μέσω της έκφρασης `destroy` *e*, όπου *e* έκφραση τύπου αναφοράς σε έγκυρο αντικείμενο, οπότε και πάλι η συγκεκριμένη θέση στο ΕΔ διαθέτει μια αναφορά στο αντικείμενο της κλάσης, όπου ορίζεται ο καταστροφέας.



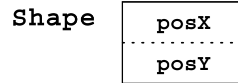
Σχήμα 4.4: Μια διαρρύθμιση της μνήμης που είναι διαθέσιμη για την εκτέλεση ενός προγράμματος.

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, γι' αυτό οργανώνεται ως μια στοίβα που λέγεται *στοίβα εκτέλεσης*.

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. Όταν ο έλεγχος επιστρέφει, το υποπρόγραμμα που κάλεσε αφαιρεί από τη στοίβα εκτέλεσης το τμήμα του ΕΔ 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.

Η διαρρύθμιση της μνήμης που περιγράφηκε παραπάνω μπορεί να τροποποιηθεί έτσι ώστε να υποστηρίζεται η δυναμική παραχώρηση μνήμης, σε γλώσσες όπως η Ooedsger. Για το σκοπό αυτό χρησιμοποιείται ένα τμήμα της ελεύθερης μνήμης που ονομάζεται *σωρός* (heap) και βρίσκεται στο κάτω μέρος της ελεύθερης μνήμης, δηλαδή στο αντίθετο άκρο από τη στοίβα εκτέλεσης. Η βάση του σωρού βρίσκεται αμέσως μετά την τελευταία θέση μνήμης που καταλαμβάνεται από το μεταφρασμένο πρόγραμμα, και ο σωρός μεγαλώνει προς τα πάνω, αντίθετα από τη στοίβα εκτέλεσης. Κατ' αυτό τον τρόπο, η μνήμη μπορεί να εξαντληθεί όταν συναντηθούν οι τρέχοντες δείκτες στο κάτω μέρος της στοίβας και στο πάνω μέρος του σωρού. Για λόγους απλοποίησης δεν έχει ληφθεί πρόνοια για την εμφάνιση κάποιου μηνύματος σφάλματος, αν συμβεί μια τέτοια σύγκρουση σε προσπάθεια δημιουργίας ενός νέου ΕΔ στη στοίβα. Αν πάλι συμβεί στην προσπάθεια δέσμευσης μνήμης του σωρού, τότε στην Ooedsger η εντολή δυναμικής παραχώρησης μνήμης επιστρέφει το μηδενικό δείκτη και είναι ευθύνη του προγράμματος να αντιμετωπίσει αυτό το ενδεχόμενο. Αυτό τον τρόπο αντιμετώπισης επιλέγουν π.χ. και οι υλοποιήσεις της γλώσσας C.

Για τη γέννηση του τελικού κώδικα ακολουθήθηκε ως βάση το πρότυπο του βιβλίου αναφοράς [1] (ενότητα 9.5). Εδώ θα περιγράψουμε τις διαφοροποιήσεις και προσθήκες που χρειάστηκαν λόγω της Ooedsger. Καταρχάς, δεν υπάρχουν μη τοπικά δεδομένα μέσα στις μεθόδους,



Σχήμα 4.5: Παράσταση στη μνήμη των αντικειμένων της κλάσης `Shape`.

όπως αυτά χαρακτηρίζονται στις προστακτικές γλώσσες, αλλά οι μεταβλητές μέσα σε μια μέθοδο είναι είτε τοπικές είτε μεταβλητές υπόστασης ή στατικές κάποιας κλάσης, όπως αναλύθηκε στην ενότητα 3.3. Ως εκ τούτου, δεν χρησιμοποιούνται σύνδεσμοι προσπέλασης. Επίσης, οι στατικές μεταβλητές υλοποιούνται ως μεταβλητές του τελικού προγράμματος `assembly`, ενώ οι μεταβλητές υπόστασης προσπελαύνονται μέσω του αντικειμένου με χρήση των αντίστοιχων αποκλίσεων (offsets) στην παράσταση του αντικειμένου. Η τελευταία περιγράφεται στις επόμενες ενότητες και η τελική της μορφή είναι ένας συνδυασμός της παράστασης του αντικειμένου, όπως αυτή περιγράφεται στις ενότητες 4.6.6 και 4.6.7. Ακόμη, όπως αναφέραμε και στην αρχή του κεφαλαίου 3, δεν υπάρχουν φωλιασμένες συναρτήσεις. Έτσι, κατά την κλήση μιας μεθόδου χρησιμοποιούμε την εντολή **push bp** για τη δέσμευση στη μνήμη της θέσης του συνδέσμου προσπέλασης. Επίσης, η κλήση μιας στατικής μεθόδου δεν διαφέρει στην υλοποίηση από την κλήση μιας απλής συνάρτησης (χωρίς αναζήτηση με συνδέσμους προσπέλασης) σε μια προστακτική γλώσσα προγραμματισμού. Αντίθετα, η κλήση σε μια δυναμική μέθοδο (μέθοδο υπόστασης) για ένα αντικείμενο υλοποιείται μέσα από τον πίνακα ανταπόκρισης μεθόδων του αντικειμένου (με χρήση της κατάλληλης απόκλισης (offset)), όπως περιγράφεται στις ενότητες 4.6.5 ως 4.6.7.

Στη συνέχεια της παρούσας ενότητας παρατίθεται η περιγραφή της υλοποίησης των αντικειμενοστρεφών μηχανισμών της Ooedsger.

4.6.2 Αντικείμενα και μέθοδοι

Όπως προαναφέρθηκε στην ενότητα 2.1, ο τύπος των αντικειμένων εκτός από πεδία δεδομένων, που προσδιορίζουν την κατάσταση ενός αντικειμένου, διαθέτει και μεθόδους που υλοποιούν τη συμπεριφορά του. Η μεταγλώττιση μιας μεθόδου υλοποιείται όπως ακριβώς αν αυτή ήταν μια κοινή διαδικασία ή συνάρτηση και είχε μια επιπλέον παράμετρο κατ' αναφορά: το αντικείμενο στο οποίο εφαρμόζεται. Επίσης, η παράσταση ενός αντικειμένου στη μνήμη δε διαφέρει, κατ' αρχήν, από αυτήν μιας εγγραφής. Απαιτείται χώρος μόνο για τα πεδία δεδομένων.

Για παράδειγμα, έστω το παρακάτω τμήμα ενός προγράμματος Ooedsger, που περιλαμβάνει τον ορισμό της κλάσης `Shape` και τη δήλωση ενός αντικειμένου `s` αυτής της κλάσης.

```
class Shape {
   double posX, posY;

   void move (double dx, double dy) {
      posX = posX + dx;
      posY = posY + dy;
   }
}
...
Shape s;       // μεταβλητή υπόστασης σε κάποια άλλη κλάση
```

Για την παράσταση του `s` και γενικά των αντικειμένων της κλάσης `Shape` στη μνήμη αρκεί να δεσμευθεί χώρος για τα πεδία **posX** και **posY**, όπως φαίνεται στο σχήμα 4.5.

Η μέθοδος **move** υλοποιείται σαν να ήταν η παρακάτω μέθοδος με την επιπλέον παράμετρο **self**, όπου **self** το αντικείμενο για το οποίο καλείται η κλάση. Το όνομα της διαδικασίας

(στον τελικό κώδικα) περιέχει το όνομα της κλάσης, έτσι ώστε να μπορούν περισσότερες κλάσεις να ορίζουν μεθόδους με τα ίδια ονόματα. Επίσης, περιέχει τον ειδικό χαρακτήρα @, που δεν επιτρέπεται να χρησιμοποιείται σε κοινά αναγνωριστικά, οπότε εξασφαλίζεται η μοναδικότητα των ονομάτων. Ακόμη περιέχει έναν μοναδικό φυσικό αριθμό για κάθε κλάση, έτσι ώστε να επιτρέπεται μέσα στην ίδια κλάση η δήλωση πολλών μεθόδων με το ίδιο όνομα (και προφανώς διαφορετικές παραμέτρους - χρήση υπερφόρτωσης). Έτσι σε ψευδοκώδικα παραπλήσιο με Ooedsger θα έχουμε για το παραπάνω παράδειγμα:

```
void Shape@move_1 (byref Shape self, double dx, double dy) {
   self.posX = self.posX + dx;
   self.posY = self.posY + dy;
}
```

Στη συνέχεια, η παρακάτω κλήση αυτής της μεθόδου:

```
s.move(1.0, 2.9);
```

υλοποιείται ως:

```
Shape@move_1(s, 1.0, 2.9);
```

4.6.3 Κατασκευαστές και καταστροφείς

Οι κατασκευαστές και οι καταστροφείς είναι μέθοδοι που καλούνται κατά την κατασκευή και καταστροφή αντικειμένων, αντίστοιχα. Υλοποιούνται όπως και οι υπόλοιπες μέθοδοι, με την αρχική διαφορά ότι δεν περνά κατά την υλοποίηση ως παράμετρος η αναφορά του αντικειμένου, όπως εξηγήθηκε στην ενότητα 4.6.1. Βέβαια, κατά την κλήση τους κρατείται χώρος στο εγγράφημα δραστηριοποίησης για την αναφορά στην κλάση στην οποία ορίζονται, οπότε μπορούν να προσπελάσουν τα πεδία της κλάσης αυτής με χρήση του τελεστή της τελείας, με εκφράσεις όπως:

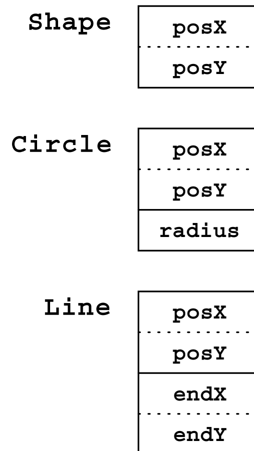
```
this.posX = ...;
```

Έτσι, στην περίπτωση των κατασκευαστών, αρχικά καλείται το υποπρόγραμμα βιβλιοθήκης `_new` για τη δέσμευση της μνήμης που απαιτείται για ένα αντικείμενο της κλάσης. Ο αριθμός των bytes που δεσμεύονται προκύπτει από τον τρόπο αναπαράστασης του αντικειμένου. Αφού γίνει η δέσμευση μνήμης για το αντικείμενο από τη `_new`, επιστρέφεται στην κατάλληλη θέση στο εγγράφημα δραστηριοποίησης (θέση αναφοράς στο τρέχον αντικείμενο) ένας δείκτης (αναφορά) προς τη μνήμη αυτή. Μέσω αυτής της θέσης μπορούν πλέον να προσπελασθούν οι μεταβλητές του αντικειμένου για αρχικοποίηση, με εκφράσεις όπως αυτή του προηγούμενου παραδείγματος. Οι μέθοδοι αυτοί επιστρέφουν ως αποτέλεσμα το αντικείμενο που δημιουργήθηκε (και τυχόν αρχικοποιήθηκε).

Αντίστοιχα, κατά την κλήση των καταστροφέων, που γίνεται μέσω της έκφρασης `destroy` e, όπου e έκφραση τύπου αναφοράς σε έγκυρο αντικείμενο, αρχικά πραγματοποιούνται οι εργασίες που ορίζονται στο σώμα της μεθόδου. Στο τέλος, γίνεται κλήση στο υποπρόγραμμα βιβλιοθήκης `_dispose` για την αποδέσμευση της μνήμης του αντικειμένου, ενώ επιστρέφεται ως αποτέλεσμα μια μηδενική αναφορά (δείκτης), που υλοποιεί τον τύπο `nil` για τις κλάσεις.

4.6.4 Απλή κληρονομικότητα

Για την παράσταση στη μνήμη των αντικειμένων μιας παραγόμενης κλάσης πρέπει να ληφθούν υπόψη τα πεδία που κληρονομούνται από τη βασική κλάση. Επίσης, για να μπορούν τα αντικείμενα της παραγόμενης κλάσης να χρησιμοποιούνται ως αντικείμενα της βασικής κλάσης, θα πρέπει τα κληρονομούμενα πεδία να έχουν τις ίδιες σχετικές διευθύνσεις όπως και στα



Σχήμα 4.6: Παράσταση στη μνήμη των αντικειμένων των κλάσεων `Circle` και `Line` που παράγονται από την κλάση `Shape`.

αντικείμενα της βασικής κλάσης. Αυτό σημαίνει ότι τα κληρονομούμενα πεδία θα πρέπει να τοποθετούνται στην αρχή της μνήμης, που δεσμεύεται για τα αντικείμενα της παραγόμενης κλάσης, και τα υπόλοιπα πεδία να ακολουθούν.

Η κλάση `Shape` του παραδείγματος της ενότητας 4.6.2 χρησιμοποιείται ως βάση μιας ιεραρχίας σχημάτων που κληρονομούν τα χαρακτηριστικά αυτής. Ένα παράδειγμα κλάσης αυτής της ιεραρχίας που κληρονομεί τη `Shape` είναι η κλάση `Circle` που ορίζεται παρακάτω:

```
class Circle extends Shape {
   double radius;
}
```

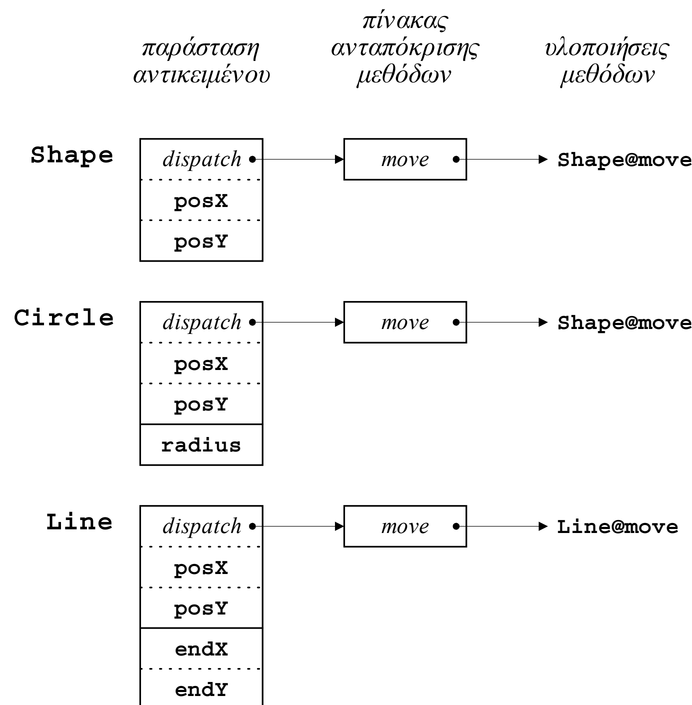
Αυτή, εκτός από τα πεδία `posX`, `posY` και τις μεθόδους της κλάσης `Shape` περιέχει και το πεδίο `radius`. Άλλο παράδειγμα παραγόμενης κλάσης είναι η `Line`:

```
class Line extends Shape {
   double endX, endY;

   void move (double dx, double dy) {
      posX = posX + dx;
      posY = posY + dy;
      endX = endX + dx;
      endY = endY + dy;
   }
}
```

Σε αυτή την κλάση, η μέθοδος `move` επισκιάζει την αντίστοιχη μέθοδο της κλάσης `Shape`. Η παράσταση στη μνήμη των αντικειμένων των κλάσεων `Circle` και `Line` μπορεί να γίνει όπως φαίνεται στο σχήμα 4.6.

4.6.5 Πολυμορφισμός

Στη γλώσσα Ooedsger υποστηρίζεται ο πολυμορφισμός υποτύπων, που ταυτίζεται με το δυναμικό δέσιμο των μεθόδων. Έτσι, για όλες τις μεθόδους, εκτός από τις στατικές και τους κατασκευαστές και καταστροφείς, που είναι χαρακτηριστικές της κλάσης, ο μεταγλωττιστής



Σχήμα 4.7: Παράσταση στη μνήμη των αντικειμένων των κλάσεων `Shape`, `Circle` και `Line` που υποστηρίζει δυναμικό δέσιμο μεθόδων.

μας αποφασίζει ποια μέθοδος πρέπει να κληθεί κατά το χρόνο εκτέλεσης και όχι το χρόνο μεταγλώττισης, όπως εξηγήθηκε στην ενότητα 2.3. Για την υλοποίηση του δυναμικού δεσίματος των μεθόδων είναι αναγκαίο να τροποποιηθεί η παράσταση των αντικειμένων στη μνήμη. Εκτός από τις θέσεις όπου φυλάσσονται τα πεδία δεδομένων των αντικειμένων, χρειάζεται και ένας πίνακας που περιέχει τις διευθύνσεις των δυναμικών μεθόδων. Ο πίνακας αυτός ονομάζεται *πίνακας ανταπόκρισης μεθόδων* (method dispatch table) ή *περιγραφέας κλάσης* (class descriptor), γιατί τα περιεχόμενά του είναι κοινά για κάθε αντικείμενο μιας δεδομένης κλάσης. Στην παράσταση των αντικειμένων κλάσεων που περιέχουν δυναμικές μεθόδους προστίθεται, πριν από τα πεδία δεδομένων, ένας δείκτης προς τον πίνακα ανταπόκρισης μεθόδων της αντίστοιχης κλάσης.

Έστωσαν για παράδειγμα οι κλάσεις `Shape`, `Circle` και `Line`, που ορίζονται στα αντίστοιχα παραδείγματα των ενοτήτων 4.6.2 και 4.6.4. Σε αυτά η μέθοδος `move` της κλάσης `Shape` και η `move` της κλάσης `Line` έχουν ορισθεί ως δυναμικές μέθοδοι. Ο τρόπος παράστασης στη μνήμη των αντικειμένων των κλάσεων `Shape`, `Circle` και `Line` φαίνεται στο σχήμα 4.7.

Έστω επίσης ότι σε κάποια κλάση έχουμε τον ακόλουθο κώδικα (makeLine και makeShape είναι κατασκευαστές για τις κλάσεις `Shape` και `Line` αντίστοιχα):

```
Line l = makeLine(1.5, 2.5, 3.5, 6.5);
Shape p = makeShape(2.5, 3.5);
p = l;
p.move(5.0, 5.0)
```

Ο μεταγλωττιστής επιλύει την τελευταία κλήση παράγοντας τελικό κώδικα ισοδύναμο με τον εξής ψευδοκώδικα (παραπλήσιο με Ooedsger):

```
f = (*p).(*dispatch).move;
(*f)(*p, 5.0, 5.0);
```

Η διεύθυνση της μεθόδου `move` που αντιστοιχεί στο αντικείμενο `*p` υπολογίζεται μέσω του πίνακα ανταπόκρισης μεθόδων για αυτό το αντικείμενο και αποθηκεύεται προσωρινά στο δείκτη

f. Στη συνέχεια, καλείται η αντίστοιχη μέθοδος (πρώτο όρισμα δίνεται το *p, αφού το p είναι αναφορά (άρα δείκτης, στη γλώσσα ψευδοκώδικα) σε κλάση). Στην περίπτωσή μας θα κληθεί λόγω του δυναμικού δεσίματος η move της Line.

Η γλώσσα Ooedsger υποστηρίζει επίσης δυναμικές μεθόδους που δηλώνονται μεν σε κάποια κλάση, ο κώδικάς τους όμως ορίζεται μόνο σε υποκλάσεις αυτής, τις λεγόμενες αφηρημένες μεθόδους. Οι κλάσεις που περιέχουν τέτοιες μεθόδους έχουν καθορισθεί κατά τη σημασιολογική ανάλυση επίσης ως αφηρημένες. Η υλοποίηση των αφηρημένων κλάσεων δεν παρουσιάζει καμία ιδιαιτερότητα. Καθώς δεν είναι δυνατόν να κατασκευαστούν αντικείμενα αυτών των κλάσεων, δε χρειάζεται να κατασκευαστούν και πίνακες ανταπόκρισης μεθόδων για αυτές στη μνήμη. Όμως για τη σωστή κατασκευή των πινάκων ανταπόκρισης μεθόδων των υποκλάσεων, που υλοποιούν τις αφηρημένες μεθόδους, θα πρέπει να κατασκευάζονται εσωτερικά στο μεταγλωττιστή ως δομές οι πίνακες ανταπόκρισης μεθόδων και των αφηρημένων κλάσεων.

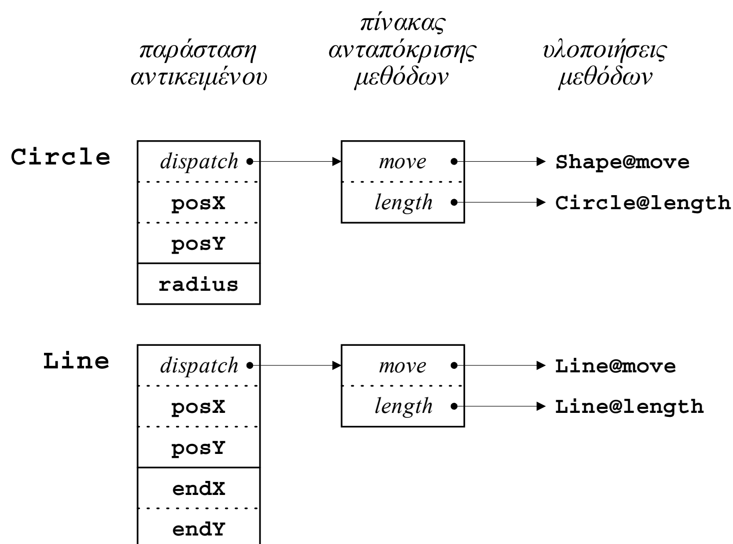
Στο προηγούμενο παράδειγμα προσθέτουμε στην κλάση Shape μια αφηρημένη μέθοδο length, η οποία υπολογίζει το συνολικό μήκος των γραμμών που χρειάζονται για να σχεδιαστεί ένα σχήμα.

```
class Shape {
   ...

   abstract double length ();
}

class Circle extends Shape
   ...

   double length () {
      return 2 * pi * radius;
   }
}

class Line extends Shape {
   ...

   double length () {
      double dx, dy;
      dx = endX - posX;
      dy = endY - posY;
      return System.sqrt(dx * dx + dy * dy);
   }
}
```

Η κλάση Shape είναι πλέον αφηρημένη, οπότε δεν είναι δυνατόν να κατασκευαστούν αντικείμενα αυτής. Η παράσταση στη μνήμη των αντικειμένων των άλλων δύο κλάσεων φαίνεται στο σχήμα 4.8.

4.6.6 Έλεγχος υποτύπων και μετατροπές

Συχνά είναι χρήσιμο να μπορεί ο προγραμματιστής να βρει κατά το χρόνο εκτέλεσης του προγράμματος σε ποια κλάση ανήκει πραγματικά κάποιο αντικείμενο. Για τον σκοπό αυτό μπορεί να χρησιμοποιηθεί η διεύθυνση του πίνακα ανταπόκρισης μεθόδων, που είναι μοναδική για κάθε κλάση. Λίγο πιο δύσκολη είναι η περίπτωση που πρέπει να ελεγχθεί αν ένα αντικείμενο ανήκει ή όχι σε κάποια δεδομένη κλάση ή σε κάποια υποκλάση αυτής. Για την υλοποίησή αυτού του ελέγχου, γνωστού ως ελέγχου υποτύπων, απαιτείται η αποθήκευση πρόσθετων πληροφοριών στον πίνακα ανταπόκρισης μεθόδων, που είναι τώρα σωστότερο να ονομάζεται περιγραφέας κλάσης, καθώς δεν περιέχει μόνο τις διευθύνσεις των μεθόδων. Έστω για παράδειγμα ο ακόλουθος έλεγχος υποτύπων:



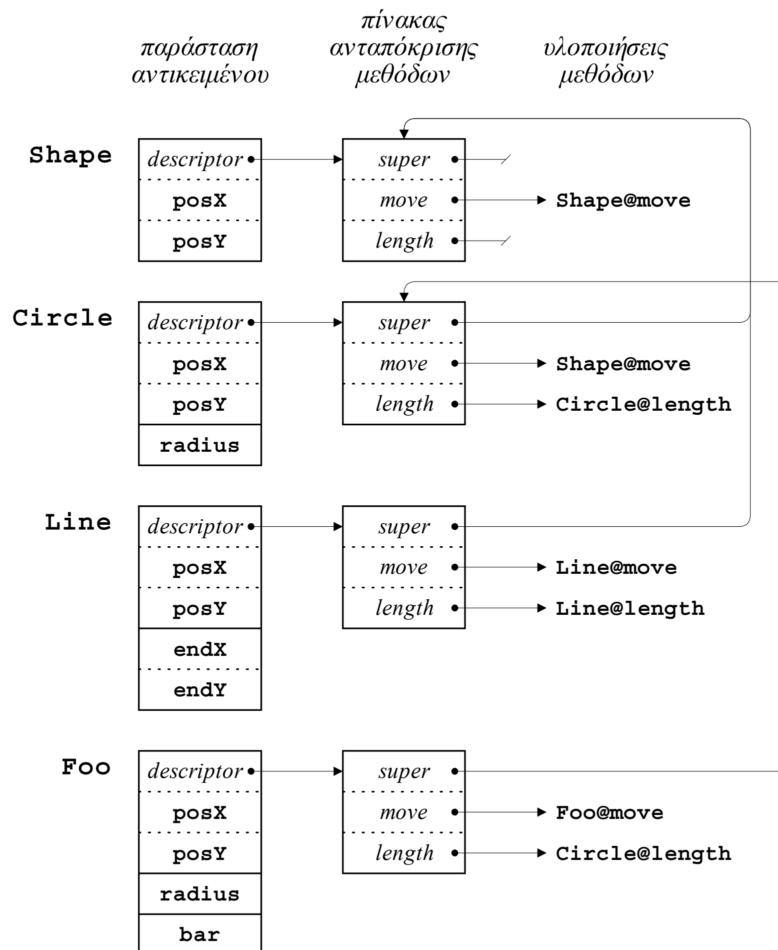
Σχήμα 4.8: Παράσταση στη μνήμη των αντικειμένων των κλάσεων `Circle` και `Line` μετά την προσθήκη της αφηρημένης μεθόδου `length`.

```
Shape p = makeShape(2.5, 3.5);
...
if (p instanceof Circle) {
   ...1...
}
else {
   ...2...
}
```

Ένας τρόπος υλοποίησης αυτού του ελέγχου είναι να αλλάξει η παράσταση των αντικειμένων της ιεραρχίας κλάσεων που ορίζεται στο τελευταίο παράδειγμα της ενότητας 4.6.5, και να μετατραπεί σε αυτήν που φαίνεται στο σχήμα 4.9, όπου έχει προστεθεί μια υποθετική κλάση `Foo` που κληρονομεί τη `Circle`. Το πεδίο `super` στον περιγραφέα κάθε κλάσης είναι ένας δείκτης προς τον περιγραφέα της βασικής κλάσης. Αν δεν υπάρχει βασική κλάση, ο δείκτης αυτός είναι μηδενικός. Ο παραπάνω έλεγχος παράγει τον παρακάτω ισοδύναμο ψευδοκώδικα παραπλήσιο με `Pascal`:

```
   d := p^.descriptor;
loop:
   if d = Circle@descriptor then
   begin
      ...1...
      goto next
   end
   else if d = nil then
   begin
      ...2...
      goto next
   end
   else
      d = d^.super;
   goto loop;
next:
```

όπου `Circle@descriptor` είναι η διεύθυνση του περιγραφέα της κλάσης `Circle`.



Σχήμα 4.9: Παράσταση στη μνήμη που επιτρέπει τον έλεγχο υποτύπων.

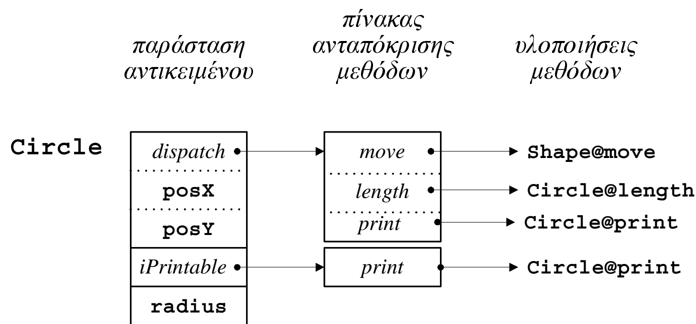
Έστω τώρα για την περίπτωση της αλλαγής τύπου (μετατροπής, casting) ότι μια κλάση `C` είναι υποκλάση της `B`. Είναι πάντα νόμιμο να μεταχειριστεί κανείς ένα αντικείμενο `c` της `C` σαν να ήταν ένα αντικείμενο της `B`. Αυτό ισοδυναμεί σε ψευδοκώδικα με μια μετατροπή του δείκτη του `c` (αναφορά στην κλάση `C`) από τον τύπο `^C` στον τύπο `^B`. Το αντίστροφο όμως, δηλαδή να μεταχειριστεί κανείς ένα αντικείμενο της `B` σαν να ήταν αντικείμενο της `C` δεν επιτρέπεται, όπως προαναφέρθηκε στην ενότητα 3.4.7, παρά μόνο στην περίπτωση που το αντικείμενο της `B` είναι στην πραγματικότητα ένα αντικείμενο της `D` ή μιας υποκλάσης αυτής. Για να εξασφαλιστεί αυτό είναι απαραίτητη η διενέργεια ενός ελέγχου υποτύπων. Στην περίπτωση που ο έλεγχος επιτύχει, τότε πάλι πρέπει να γίνει μια μετατροπή δείκτη από τον τύπο `^B` στον τύπο `^D`.

4.6.7 Διαπροσωπείες

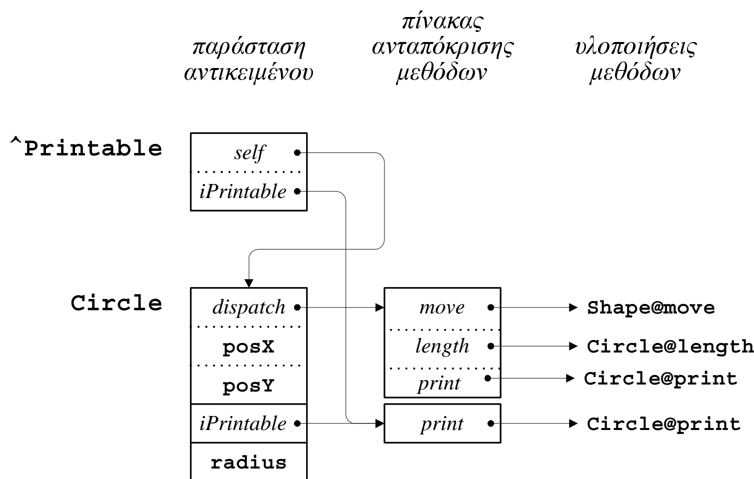
Για την παράσταση στη μνήμη των διαπροσωπειών χρησιμοποιούνται επιπλέον πίνακες ανταπόκρισης μεθόδων. Τα αντικείμενα μιας κλάσης πρέπει να διαθέτουν έναν τέτοιο πίνακα για κάθε διαπροσωπεία που υλοποιεί η κλάση. Τέλος, η παράσταση στη μνήμη των αναφορών σε τύπους διαπροσωπείας μπορεί να γίνει με ένα δείκτη προς το αντικείμενο που υλοποιεί τη διαπροσωπεία και ένα δείκτη προς τον πίνακα ανταπόκρισης μεθόδων της διαπροσωπείας μέσα σε αυτό το αντικείμενο.

Έστω λοιπόν η διαπροσωπεία `Printable` που ορίζεται παρακάτω και υλοποιείται από την κλάση `Circle` του τελευταίου παραδείγματος της ενότητας 4.6.5, που επεκτείνεται:

```
interface Printable {
```



Σχήμα 4.10: Παράσταση στη μνήμη κλάσης που υλοποιεί διαπροσωπεία.



Σχήμα 4.11: Παράσταση στη μνήμη αντικειμένου κλάσης και του αντίστοιχου δείκτη σε διαπροσωπεία.

```
   void print ();
}

class Circle extends Shape implements Printable {
   ...

   void print () {
   ...
   }
}
```

Η παράσταση στη μνήμη των αντικειμένων της κλάσης `Circle` φαίνεται στο σχήμα 4.10. Στη συνέχεια, έστω ο παρακάτω κώδικας:

```
Circle c = makeCircle(2.5, 3.5, 1.0);
Printable p;
...

p = c;
p.print();
```

Η παράσταση των μεταβλητών `c` και `p` στη μνήμη φαίνεται στο σχήμα 4.11. Για τη μεταγλώττιση της εντολής `p = c` υλοποιείται ο παρακάτω ισοδύναμος ψευδοκώδικας παραπλήσιος με `Pascal`:

```
p.self = @c;
p.iPrintable = c.iPrintable;
```

ενώ για τη μεταγλώττιση της `p.print()` υλοποιείται ισοδύναμα:

```
p.iPrintable^.print(p.self)
```

Κεφάλαιο 5

Παραδείγματα

Στην παράγραφο αυτή δίνονται τέσσερα παραδείγματα προγραμμάτων στη γλώσσα Ooedsger. Για κάθε ένα δίνεται ο αρχικός κώδικας, ο ενδιάμεσος κώδικας, η μορφή των εγγραφημάτων δραστηριοποίησης των δομικών μονάδων, η παράσταση στη μνήμη των κλάσεων και των δεικτών σε διαπροσωπείες, καθώς και ο τελικός κώδικας.

5.1 Πες γεια!

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

```
class Hello
{
  static void main ()
  {
    System.writeString("Hello world!\n");
  }
}
```

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

```
1: unit, @class_destruc, -, -
2: endu, @class_destruc, -, -
3: unit, main, -, -
4: par, "Hello world!\n", V, -
5: call, -, System, writeString
6: endu, main, -, -
```

Το εγγράφημα δραστηριοποίησης του προγράμματος, καθώς και η παράσταση στη μνήμη της κλάσης Hello είναι εξαιρετικά απλά και δε θα δοθούν. Ο τελικός κώδικας είναι ο εξής:

```
xseg segment public 'code'
assume cs:xseg, ds:xseg, ss:xseg
org 100h
main proc near
call near ptr Hello@main_1
mov ax, 4C00h
int 21h
main endp


@1:

; 1: unit,@class_destruc,-,-
```

```
Hello@@class_destruc_0 proc near
push bp
mov bp, sp
sub sp, 0

@2:

; 2: endu,@class_destruc,-,-

@Hello@@class_destruc_0:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
```

```
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Hello@@class_destruc_0 endp


@3:

; 3: unit,main,-,-

Hello@main_1 proc near
push bp
mov bp, sp
sub sp, 0

@4:

; 4: par,"Hello world!\n",V,-

lea ax, byte ptr @str_1
push ax

@5:

; 5: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@6:

; 6: endu,main,-,-

@Hello@main_1:
mov sp, bp
pop bp
ret
Hello@main_1 endp

errordc proc near
mov ah, 09h
lea dx, errstr
int 21h
mov ax, 4C00h
int 21h
errordc endp

Hello@descriptor dw 0
dw Hello@@class_destruc_0

extrn __dispose : proc
extrn _writeString : proc

errstr db 13, 10, 'Error in downcasting,
                exiting!$', 13, 10, 0
@str_1 db 'Hello world!', 13, 10, 0

xseg ends
end main
```

5.2 Παράδειγμα επίδειξης κατασκευαστών και καταστροφέων

Το επόμενο πρόγραμμα (`obj.oed`) δείχνει τη χρήση κατασκευαστών και καταστροφέων στην Ooedsger, ενώ περιέχει και κλήση στη δυναμική μέθοδο `move` για τα αντικείμενα της κλάσης `Object` που δημιουργούνται.

```
class Object
{
  char name[32];

  constructor makeObj (char * name)
  {
    System.strcpy(this.name, name);
  }

  void write ()
  {
    System.writeString("I am object ");
    System.writeString(name);
    System.writeString("\n");
  }

  destructor killObj ()
  {
    System.writeString("Help!  Someone\'s killing ");
    System.writeString(name);
```

```
      System.writeString("!\n");
    }
  }

  class Example
  {
    static void main ()
    {
      Object x, y;

      x = makeObj("x the great");
      y = makeObj("y the greatest");

      x.write();
      y.write();

      destroy x;
      destroy y;
    }
  }
```

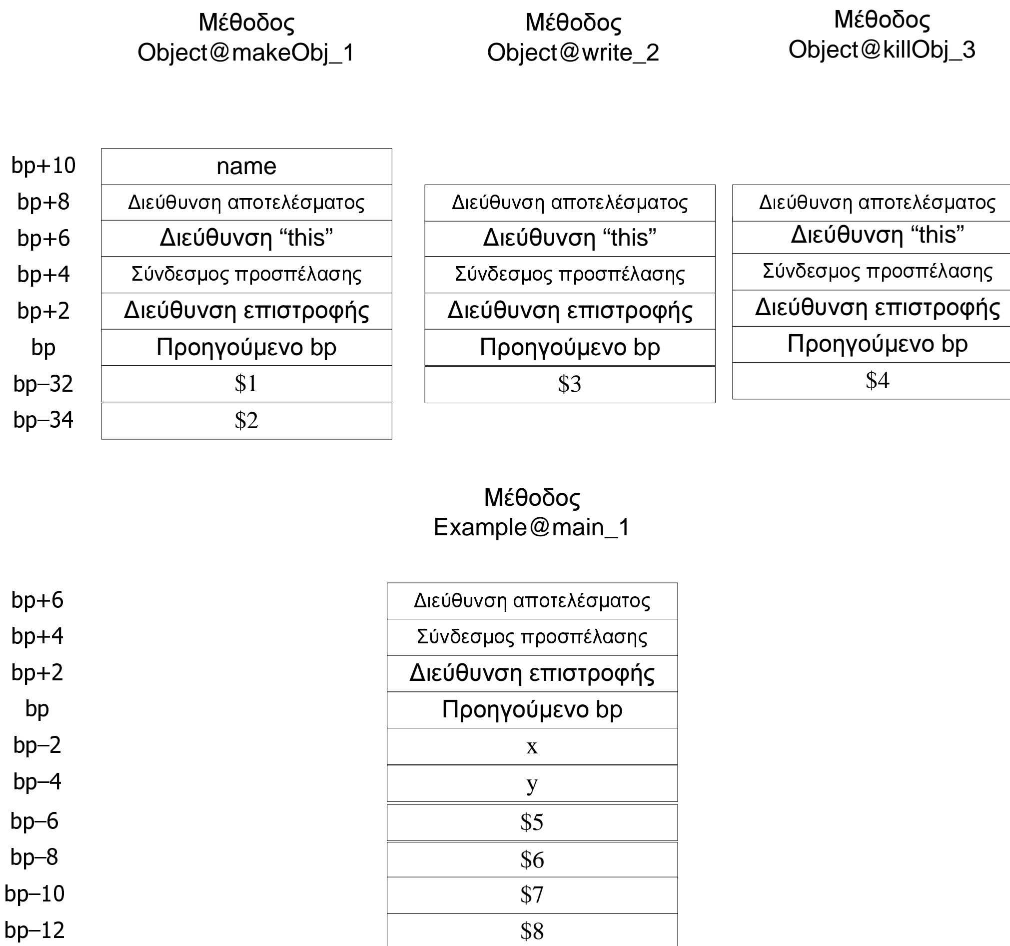
Ο ενδιάμεσος κώδικας που παράγει ο μεταγλωττιστής της Ooedsger είναι ο εξής:

```
 1: unit, @class_destruc, -, -
 2: endu, @class_destruc, -, -
 3: unit, makeObj, -, -
 4: dot, this, name, $1
 5: =, $1, -, $2
 6: par, $2, V, -
 7: par, name, V, -
 8: call, -, System, strcpy
 9: endu, makeObj, -, -
10: unit, write, -, -
11: par, "I am object ", V, -
12: call, -, System, writeString
13: =, name, -, $3
14: par, $3, V, -
15: call, -, System, writeString
16: par, "\n", V, -
17: call, -, System, writeString
18: endu, write, -, -
19: unit, killObj, -, -
20: par, "Help!  Someone\'s killing ",
          V, -
21: call, -, System, writeString
22: =, name, -, $4
23: par, $4, V, -
24: call, -, System, writeString
25: par, "!\n", V, -
26: call, -, System, writeString
27: endu, killObj, -, -
28: unit, main, -, -
29: par, "x the great", V, -
30: par, $5, RET, -
31: call, -, Object, makeObj
32: =, $5, -, x
33: par, "y the greatest", V, -
34: par, $6, RET, -
35: call, -, Object, makeObj
36: =, $6, -, y
37: call, -, x, write
38: call, -, y, write
39: par, $7, RET, -
40: call, -, x, killObj
41: par, $8, RET, -
42: call, -, y, killObj
43: endu, main, -, -
```

Τα εγγραφήματα δραστηριοποίησης, καθώς και η παράσταση στη μνήμη των κλάσεων για το παραπάνω πρόγραμμα φαίνονται στα σχήματα 5.1 και 5.2 αντίστοιχα.

Ο τελικός κώδικας δίνεται παρακάτω:

```
xseg segment public 'code'
assume cs:xseg, ds:xseg, ss:xseg
org 100h
main proc near
call near ptr Example@main_1
mov ax, 4C00h
int 21h
main endp


@1:

; 1: unit,@class_destruc,-,-

Example@@class_destruc_0 proc near
push bp
```



Σχήμα 5.1: Εγγραφήματα δραστηριοποίησης για το πρόγραμμα obj.oed.

```
mov bp, sp
sub sp, 0

@2:

; 2: endu,@class_destruc,-,-

@Example@@class_destruc_0:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Example@@class_destruc_0 endp


@3:

; 3: unit,makeObj,-,-

Object@makeObj_1 proc near
push bp
```

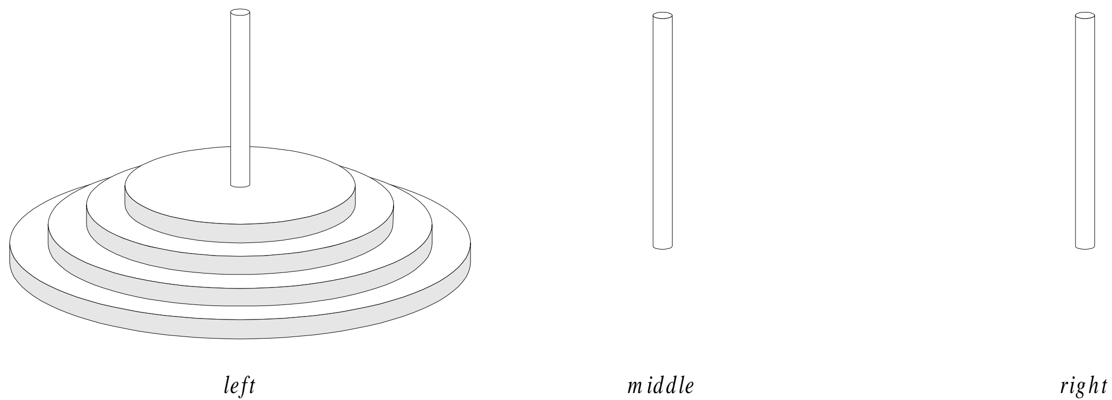
```
mov bp, sp
sub sp, 34
mov ax, 34
push ax
lea si, word ptr [bp + 6]
push si
sub sp, 2
call near ptr __new
add sp, 6
mov di, word ptr [bp + 6]
mov si, word ptr [bp + 8]
mov word ptr [si], di
lea si, word ptr Object@descriptor
mov word ptr [di], si

@4:

; 4: dot,this,name,$1

mov ax, word ptr [bp + 6]
add ax, 2
mov word ptr [bp -32], ax

@5:

; 5: =,$1,-,$2
```

Σχήμα 5.2: Παράσταση στη μνήμη των κλάσεων για το πρόγραμμα obj.oed.

```
mov ax, word ptr [bp -32]
mov word ptr [bp -34], ax

@6:

; 6: par,$2,V,-

mov ax, word ptr [bp -34]
push ax

@7:

; 7: par,name,V,-

mov ax, word ptr [bp +10]
push ax

@8:

; 8: call,-,System,strcpy

sub sp, 2
push bp
call near ptr _strcpy
add sp, 8

@9:

; 9: endu,makeObj,-,-

@Object@makeObj_1:
mov sp, bp
pop bp
ret
Object@makeObj_1 endp


@10:

; 10: unit,write,-,-

Object@write_2 proc near
push bp
mov bp, sp
sub sp, 2

@11:

; 11: par,"I am object ",V,-

lea ax, byte ptr @str_1
push ax

@12:

; 12: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@13:

; 13: =,name,-,$3

mov si, word ptr [bp + 6]
add si, 2
lea ax, word ptr [si]
mov word ptr [bp -2], ax

@14:

; 14: par,$3,V,-

mov ax, word ptr [bp -2]
push ax

@15:

; 15: call,-,System,writeString

sub sp, 2
push bp
```

```
call near ptr _writeString
add sp, 6

@16:

; 16: par,"\n",V,-

lea ax, byte ptr @str_2
push ax

@17:

; 17: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@18:

; 18: endu,write,-,-

@Object@write_2:
mov sp, bp
pop bp
ret
Object@write_2 endp


@19:

; 19: unit,killObj,-,-

Object@killObj_3 proc near
push bp
mov bp, sp
sub sp, 2

@20:

; 20: par,"Help!  Someone\'s killing ",V,-

lea ax, byte ptr @str_3
push ax

@21:

; 21: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@22:

; 22: =,name,-,$4

mov si, word ptr [bp + 6]
add si, 2
lea ax, word ptr [si]
mov word ptr [bp -2], ax

@23:

; 23: par,$4,V,-

mov ax, word ptr [bp -2]
push ax

@24:

; 24: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@25:

; 25: par,"!\n",V,-

lea ax, byte ptr @str_4
push ax

@26:

; 26: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@27:

; 27: endu,killObj,-,-

@Object@killObj_3:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Object@killObj_3 endp


@28:

; 28: unit,main,-,-

Example@main_1 proc near
push bp
mov bp, sp
sub sp, 12

@29:

; 29: par,"x the great",V,-
```

```
lea ax, byte ptr @str_5
push ax

@30:

; 30: par,$5,RET,-

lea ax, word ptr [bp -6]
push ax

@31:

; 31: call,-,Object,makeObj

sub sp, 2
push bp
call near ptr Object@makeObj_1
add sp, 8

@32:

; 32: =,$5,-,x

mov ax, word ptr [bp -6]
mov word ptr [bp -2], ax

@33:

; 33: par,"y the greatest",V,-

lea ax, byte ptr @str_6
push ax

@34:

; 34: par,$6,RET,-

lea ax, word ptr [bp -8]
push ax

@35:

; 35: call,-,Object,makeObj

sub sp, 2
push bp
call near ptr Object@makeObj_1
add sp, 8

@36:

; 36: =,$6,-,y

mov ax, word ptr [bp -8]
mov word ptr [bp -4], ax

@37:

; 37: call,-,x,write

sub sp, 2
mov si, word ptr [bp -2]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 2]
add sp, 6

@38:

; 38: call,-,y,write

sub sp, 2
mov si, word ptr [bp -4]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 2]
add sp, 6

@39:

; 39: par,$7,RET,-

lea ax, word ptr [bp -10]
push ax

@40:

; 40: call,-,x,killObj

mov si, word ptr [bp -2]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@41:

; 41: par,$8,RET,-

lea ax, word ptr [bp -12]
push ax

@42:

; 42: call,-,y,killObj

mov si, word ptr [bp -4]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@43:

; 43: endu,main,-,-

@Example@main_1:
mov sp, bp
pop bp
ret
Example@main_1 endp
```



Σχήμα 5.3: Οι πύργοι του Hanoi.

```
errordc proc near
mov ah, 09h
lea dx, errstr
int 21h
mov ax, 4C00h
int 21h
errordc endp

Object@descriptor dw 0
dw Object@write_2
dw Object@killObj_3
Example@descriptor dw 0
dw Example@@class_destruc_0

extrn __dispose : proc
extrn __new : proc
extrn _strcpy : proc
extrn _writeString : proc

errstr db 13, 10, 'Error in downcasting,
                exiting!$', 13, 10, 0
@str_1 db 'I am object ', 0
@str_2 db 13, 10, 0
@str_3 db 'Help!  Someone', 39, 's killing
                ', 0
@str_4 db '!', 13, 10, 0
@str_5 db 'x the great', 0
@str_6 db 'y the greatest', 0

xseg ends
end main
```

5.3 Οι πύργοι του Hanoi

Το πρόγραμμα που ακολουθεί λύνει το πρόβλημα των πύργων του Hanoi. Μια σύντομη περιγραφή του προβλήματος δίνεται παρακάτω.

Υπάρχουν τρεις στύλοι, στον πρώτο από τους οποίους είναι περασμένοι n το πλήθος δακτύλιοι. Οι εξωτερικές διάμετροι των δακτυλίων είναι διαφορετικές και αυτοί είναι περασμένοι από κάτω προς τα πάνω σε φθίνουσα σειρά εξωτερικής διαμέτρου, όπως φαίνεται στο σχήμα 5.3. Ζητείται να μεταφερθούν οι δακτύλιοι από τον πρώτο στον τρίτο στύλο (χρησιμοποιώντας το δεύτερο ως βοηθητικό χώρο), ακολουθώντας όμως τους εξής κανόνες:

- Κάθε φορά επιτρέπεται να μεταφερθεί ένας μόνο δακτύλιος, από κάποιο στύλο σε κάποιον άλλο στύλο.
- Απαγορεύεται να τοποθετηθεί δακτύλιος με μεγαλύτερη διάμετρο πάνω από δακτύλιο με μικρότερη διάμετρο.

Το πρόγραμμα στη γλώσσα Ooedsger που λύνει αυτό το πρόβλημα δίνεται παρακάτω. Η μέθοδος `hanoi` είναι αναδρομική.

```
class Pile
{
  char name [10];

  constructor makePile (char * n)
  {
```

```
    System.strcpy(name, n);
  }

  void write ()
  {
    System.writeString(name);
  }
}

class Hanoi
{
  static void main ()
  {
    Pile source, target, auxiliary;
    int rings;
    int moves;

    System.writeString("Please, give the number of rings: ");
    rings = System.readInteger();
    source = makePile("source");
    target = makePile("target");
    auxiliary = makePile("auxiliary");

    moves = hanoi(rings, source, target, auxiliary);

    System.writeString("Total moves: ");
    System.writeInteger(moves);
    System.writeString("\n");

    destroy source;
    destroy target;
    destroy auxiliary;
  }

  static int hanoi (int n, Pile source, Pile target, Pile auxiliary)
  {
    if (n > 0) {
      int moves = 0;
      moves += hanoi(n-1, source, auxiliary, target);
      move(source, target);
      moves++;
      moves += hanoi(n-1, auxiliary, target, source);
      return moves;
    }
    else
      return 0;
  }

  static void move (Pile source, Pile target)
  {
    System.writeString("Move from ");
    source.write();
    System.writeString(" to ");
    target.write();
    System.writeString("\n");
  }
}
```

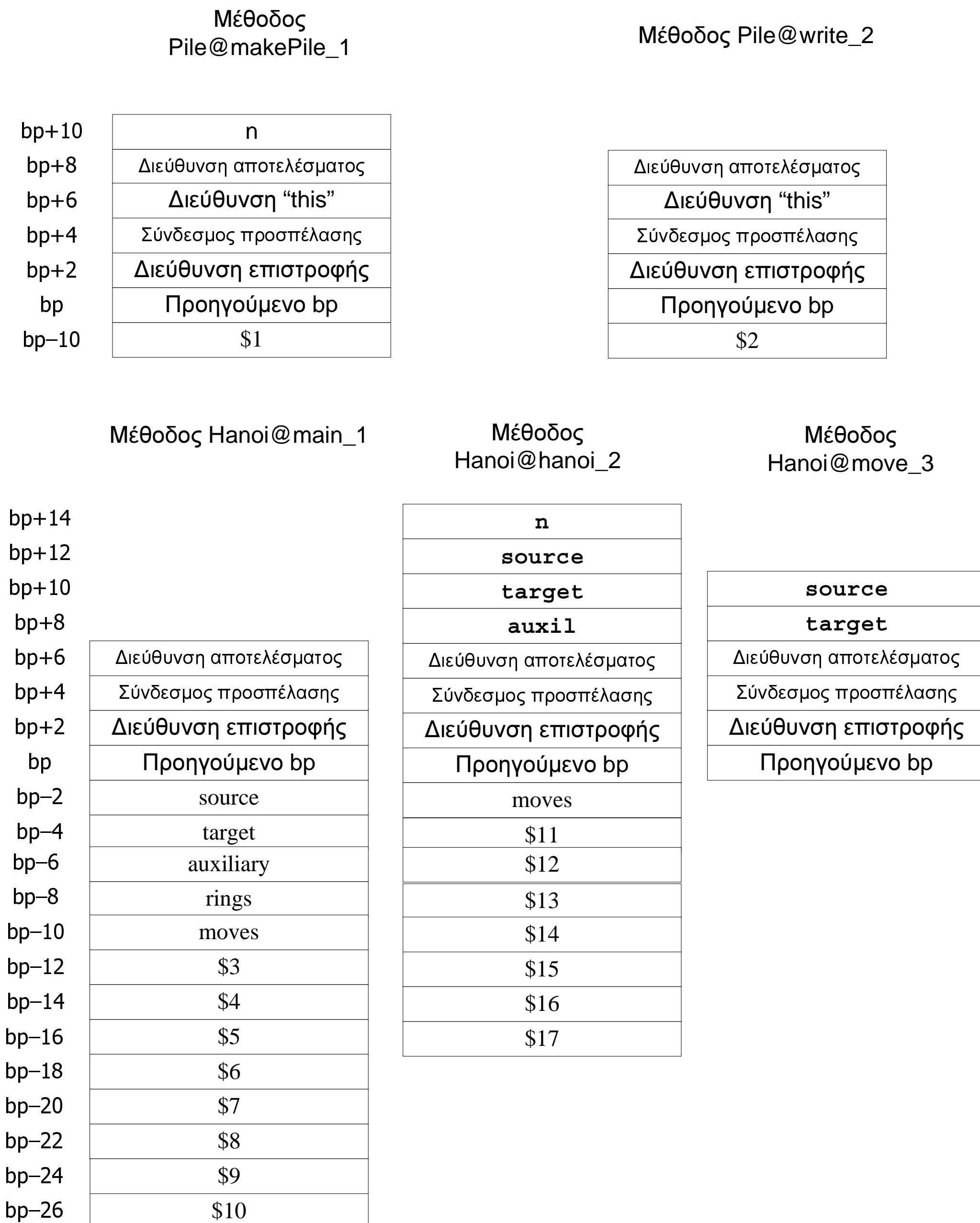
Ο ενδιάμεσος κώδικας που παράγει ο μεταγλωττιστής της Ooedsger είναι ο εξής:

```
 1: unit, @class_destruc, -, -
 2: endu, @class_destruc, -, -
 3: unit, @class_destruc, -, -
 4: endu, @class_destruc, -, -
 5: unit, makePile, -, -
 6: =, name, -, $1
 7: par, $1, V, -
 8: par, n, V, -
 9: call, -, System, strcpy
10: endu, makePile, -, -
11: unit, write, -, -
12: =, name, -, $2
13: par, $2, V, -
14: call, -, System, writeString
15: endu, write, -, -
16: unit, main, -, -
17: par, "Please, give the number of
         rings: ", V, -
18: call, -, System, writeString
19: par, $3, RET, -
20: call, -, System, readInteger
21: =, $3, -, rings
22: par, "source", V, -
23: par, $4, RET, -
24: call, -, Pile, makePile
25: =, $4, -, source
26: par, "target", V, -
27: par, $5, RET, -
28: call, -, Pile, makePile
29: =, $5, -, target
30: par, "auxiliary", V, -
31: par, $6, RET, -
32: call, -, Pile, makePile
33: =, $6, -, auxiliary
34: par, rings, V, -
35: par, source, V, -
36: par, target, V, -
37: par, auxiliary, V, -
38: par, $7, RET, -
39: call, -, this, hanoi
40: =, $7, -, moves
41: par, "Total moves: ", V, -
42: call, -, System, writeString
43: par, moves, V, -
44: call, -, System, writeInteger
45: par, "\n", V, -
46: call, -, System, writeString
47: par, $8, RET, -
48: call, -, source, @class_destruc
49: par, $9, RET, -
50: call, -, target, @class_destruc
51: par, $10, RET, -
52: call, -, auxiliary, @class_destruc
53: endu, main, -, -
54: unit, hanoi, -, -
55: >, n, 0, 57
56: jump, -, -, 84
57: =, 0, -, moves
58: -, n, 1, $11
59: par, $11, V, -
60: par, source, V, -
61: par, auxiliary, V, -
62: par, target, V, -
63: par, $12, RET, -
64: call, -, this, hanoi
65: +, moves, $12, $13
66: =, $13, -, moves
67: par, source, V, -
68: par, target, V, -
69: call, -, this, move
70: +, moves, 1, $14
71: =, $14, -, moves
72: -, n, 1, $15
73: par, $15, V, -
74: par, auxiliary, V, -
75: par, target, V, -
76: par, source, V, -
77: par, $16, RET, -
78: call, -, this, hanoi
79: +, moves, $16, $17
80: =, $17, -, moves
81: =, moves, -, $$
82: ret, -, -, -
83: jump, -, -, 86
84: =, 0, -, $$
85: ret, -, -, -
86: endu, hanoi, -, -
87: unit, move, -, -
88: par, "Move from ", V, -
89: call, -, System, writeString
90: call, -, source, write
91: par, " to ", V, -
92: call, -, System, writeString
93: call, -, target, write
94: par, "\n", V, -
95: call, -, System, writeString
96: endu, move, -, -
```

Τα εγγραφήματα δραστηριοποίησης, καθώς και η παράσταση στη μνήμη των κλάσεων για το παραπάνω πρόγραμμα φαίνονται στα σχήματα 5.4 και 5.5 αντίστοιχα.

Ο τελικός κώδικας δίνεται παρακάτω:

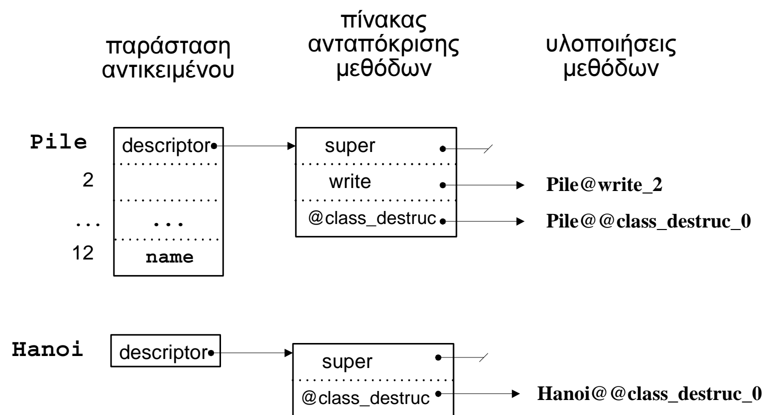
```
xseg segment public 'code'
assume cs:xseg, ds:xseg, ss:xseg
org 100h
main proc near
call near ptr Hanoi@main_1
mov ax, 4C00h
int 21h
main endp
```



Σχήμα 5.4: Εγγραφήματα δραστηριοποίησης για τους Πύργους του Hanoi.

```
@1:

; 1: unit,@class_destruc,-,-

Pile@@class_destruc_0 proc near
push bp
mov bp, sp
sub sp, 0

@2:

; 2: endu,@class_destruc,-,-
```

```
@Pile@@class_destruc_0:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Pile@@class_destruc_0 endp


@3:
```



Σχήμα 5.5: Παράσταση στη μνήμη των κλάσεων για τους Πύργους του Hanoi.

```
; 3: unit,@class_destruc,-,-

Hanoi@@class_destruc_0 proc near
push bp
mov bp, sp
sub sp, 0

@4:

; 4: endu,@class_destruc,-,-

@Hanoi@@class_destruc_0:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Hanoi@@class_destruc_0 endp


@5:

; 5: unit,makePile,-,-

Pile@makePile_1 proc near
push bp
mov bp, sp
sub sp, 2
mov ax, 12
push ax
lea si, word ptr [bp + 6]
push si
sub sp, 2
call near ptr __new
add sp, 6
mov di, word ptr [bp + 6]
mov si, word ptr [bp + 8]
mov word ptr [si], di
lea si, word ptr Pile@descriptor
mov word ptr [di], si

@6:

; 6: =,name,-,$1

mov si, word ptr [bp + 6]
add si, 2
lea ax, word ptr [si]
mov word ptr [bp -2], ax

@7:

; 7: par,$1,V,-

mov ax, word ptr [bp -2]
push ax

@8:

; 8: par,n,V,-

mov ax, word ptr [bp +10]
push ax

@9:

; 9: call,-,System,strcpy

sub sp, 2
push bp
call near ptr _strcpy
add sp, 8

@10:

; 10: endu,makePile,-,-

@Pile@makePile_1:
mov sp, bp
pop bp
ret
Pile@makePile_1 endp
```

```
@11:

; 11: unit,write,-,-

Pile@write_2 proc near
push bp
mov bp, sp
sub sp, 2

@12:

; 12: =,name,-,$2

mov si, word ptr [bp + 6]
add si, 2
lea ax, word ptr [si]
mov word ptr [bp -2], ax

@13:

; 13: par,$2,V,-

mov ax, word ptr [bp -2]
push ax

@14:

; 14: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@15:

; 15: endu,write,-,-

@Pile@write_2:
mov sp, bp
pop bp
ret
Pile@write_2 endp


@16:

; 16: unit,main,-,-

Hanoi@main_1 proc near
push bp
mov bp, sp
sub sp, 26

@17:

; 17: par,"Please, give the number of
                rings: ", V,-

lea ax, byte ptr @str_1
push ax

@18:

; 18: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@19:

; 19: par,$3,RET,-

lea ax, word ptr [bp -12]
push ax

@20:

; 20: call,-,System,readInteger

push bp
call near ptr _readInteger
add sp, 4

@21:

; 21: =,$3,-,rings

mov ax, word ptr [bp -12]
mov word ptr [bp -8], ax

@22:

; 22: par,"source",V,-

lea ax, byte ptr @str_2
push ax

@23:

; 23: par,$4,RET,-

lea ax, word ptr [bp -14]
push ax

@24:

; 24: call,-,Pile,makePile

sub sp, 2
push bp
call near ptr Pile@makePile_1
add sp, 8

@25:

; 25: =,$4,-,source

mov ax, word ptr [bp -14]
mov word ptr [bp -2], ax

@26:

; 26: par,"target",V,-
```

```
lea ax, byte ptr @str_3
push ax

@27:

; 27: par,$5,RET,-

lea ax, word ptr [bp -16]
push ax

@28:

; 28: call,-,Pile,makePile

sub sp, 2
push bp
call near ptr Pile@makePile_1
add sp, 8

@29:

; 29: =,$5,-,target

mov ax, word ptr [bp -16]
mov word ptr [bp -4], ax

@30:

; 30: par,"auxiliary",V,-

lea ax, byte ptr @str_4
push ax

@31:

; 31: par,$6,RET,-

lea ax, word ptr [bp -18]
push ax

@32:

; 32: call,-,Pile,makePile

sub sp, 2
push bp
call near ptr Pile@makePile_1
add sp, 8

@33:

; 33: =,$6,-,auxiliary

mov ax, word ptr [bp -18]
mov word ptr [bp -6], ax

@34:

; 34: par,rings,V,-

mov ax, word ptr [bp -8]
push ax

@35:

; 35: par,source,V,-

mov ax, word ptr [bp -2]
push ax

@36:

; 36: par,target,V,-

mov ax, word ptr [bp -4]
push ax

@37:

; 37: par,auxiliary,V,-

mov ax, word ptr [bp -6]
push ax

@38:

; 38: par,$7,RET,-

lea ax, word ptr [bp -20]
push ax

@39:

; 39: call,-,this,hanoi

push bp
call near ptr Hanoi@hanoi_2
add sp, 12

@40:

; 40: =,$7,-,moves

mov ax, word ptr [bp -20]
mov word ptr [bp -10], ax

@41:

; 41: par,"Total moves: ",V,-

lea ax, byte ptr @str_5
push ax

@42:

; 42: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@43:

; 43: par,moves,V,-
```

```
mov ax, word ptr [bp -10]
push ax

@44:

; 44: call,-,System,writeInteger

sub sp, 2
push bp
call near ptr _writeInteger
add sp, 6

@45:

; 45: par,"\n",V,-

lea ax, byte ptr @str_6
push ax

@46:

; 46: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@47:

; 47: par,$8,RET,-

lea ax, word ptr [bp -22]
push ax

@48:

; 48: call,-,source,@class_destruc

mov si, word ptr [bp -2]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@49:

; 49: par,$9,RET,-

lea ax, word ptr [bp -24]
push ax

@50:

; 50: call,-,target,@class_destruc

mov si, word ptr [bp -4]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@51:

; 51: par,$10,RET,-

lea ax, word ptr [bp -26]
push ax

@52:

; 52: call,-,auxiliary,@class_destruc

mov si, word ptr [bp -6]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@53:

; 53: endu,main,-,-

@Hanoi@main_1:
mov sp, bp
pop bp
ret
Hanoi@main_1 endp


@54:

; 54: unit,hanoi,-,-

Hanoi@hanoi_2 proc near
push bp
mov bp, sp
sub sp, 16

@55:

; 55: >,n,0,57

mov ax, word ptr [bp +14]
mov dx, 0
cmp ax, dx
jg @57

@56:

; 56: jump,-,-,84

jmp @84

@57:

; 57: =,0,-,moves

mov ax, 0
mov word ptr [bp -2], ax

@58:
```

```
; 58: -,n,1,$11

mov ax, word ptr [bp +14]
mov dx, 1
sub ax, dx
mov word ptr [bp -4], ax

@59:

; 59: par,$11,V,-

mov ax, word ptr [bp -4]
push ax

@60:

; 60: par,source,V,-

mov ax, word ptr [bp +12]
push ax

@61:

; 61: par,auxiliary,V,-

mov ax, word ptr [bp +8]
push ax

@62:

; 62: par,target,V,-

mov ax, word ptr [bp +10]
push ax

@63:

; 63: par,$12,RET,-

lea ax, word ptr [bp -6]
push ax

@64:

; 64: call,-,this,hanoi

push bp
call near ptr Hanoi@hanoi_2
add sp, 12

@65:

; 65: +,moves,$12,$13

mov ax, word ptr [bp -2]
mov dx, word ptr [bp -6]
add ax, dx
mov word ptr [bp -8], ax

@66:

; 66: =,$13,-,moves

mov ax, word ptr [bp -8]
mov word ptr [bp -2], ax

@67:

; 67: par,source,V,-

mov ax, word ptr [bp +12]
push ax

@68:

; 68: par,target,V,-

mov ax, word ptr [bp +10]
push ax

@69:

; 69: call,-,this,move

sub sp, 2
push bp
call near ptr Hanoi@move_3
add sp, 8

@70:

; 70: +,moves,1,$14

mov ax, word ptr [bp -2]
mov dx, 1
add ax, dx
mov word ptr [bp -10], ax

@71:

; 71: =,$14,-,moves

mov ax, word ptr [bp -10]
mov word ptr [bp -2], ax

@72:

; 72: -,n,1,$15

mov ax, word ptr [bp +14]
mov dx, 1
sub ax, dx
mov word ptr [bp -12], ax

@73:

; 73: par,$15,V,-

mov ax, word ptr [bp -12]
push ax

@74:

; 74: par,auxiliary,V,-
```

```
mov ax, word ptr [bp +8]
push ax

@75:

; 75: par,target,V,-

mov ax, word ptr [bp +10]
push ax

@76:

; 76: par,source,V,-

mov ax, word ptr [bp +12]
push ax

@77:

; 77: par,$16,RET,-

lea ax, word ptr [bp -14]
push ax

@78:

; 78: call,-,this,hanoi

push bp
call near ptr Hanoi@hanoi_2
add sp, 12

@79:

; 79: +,moves,$16,$17

mov ax, word ptr [bp -2]
mov dx, word ptr [bp -14]
add ax, dx
mov word ptr [bp -16], ax

@80:

; 80: =,$17,-,moves

mov ax, word ptr [bp -16]
mov word ptr [bp -2], ax

@81:

; 81: =,moves,-,$$

mov ax, word ptr [bp -2]
mov si, word ptr [bp +6]
mov word ptr [si], ax

@82:

; 82: ret,-,-,-

jmp @Hanoi@hanoi_2

@83:

; 83: jump,-,-,86

jmp @86

@84:

; 84: =,0,-,$$

mov ax, 0
mov si, word ptr [bp +6]
mov word ptr [si], ax

@85:

; 85: ret,-,-,-

jmp @Hanoi@hanoi_2

@86:

; 86: endu,hanoi,-,-

@Hanoi@hanoi_2:
mov sp, bp
pop bp
ret
Hanoi@hanoi_2 endp


@87:

; 87: unit,move,-,-

Hanoi@move_3 proc near
push bp
mov bp, sp
sub sp, 0

@88:

; 88: par,"Move from ",V,-

lea ax, byte ptr @str_7
push ax

@89:

; 89: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@90:

; 90: call,-,source,write

sub sp, 2
mov si, word ptr [bp +10]
push si
push bp
```

```
mov si, word ptr [si]
call word ptr [si + 2]
add sp, 6

@91:

; 91: par," to ",V,-

lea ax, byte ptr @str_8
push ax

@92:

; 92: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@93:

; 93: call,-,target,write

sub sp, 2
mov si, word ptr [bp +8]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 2]
add sp, 6

@94:

; 94: par,"\n",V,-

lea ax, byte ptr @str_6
push ax

@95:

; 95: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@96:

; 96: endu,move,-,-

@Hanoi@move_3:
mov sp, bp
pop bp
ret
Hanoi@move_3 endp

errordc proc near
mov ah, 09h
lea dx, errstr
int 21h
mov ax, 4C00h
int 21h
errordc endp

Pile@descriptor dw 0
dw Pile@write_2
dw Pile@@class_destruc_0
Hanoi@descriptor dw 0
dw Hanoi@@class_destruc_0

extrn __dispose : proc
extrn __new : proc
extrn _strcpy : proc
extrn _writeString : proc
extrn _readInteger : proc
extrn _writeInteger : proc

errstr db 13, 10, 'Error in downcasting,
                exiting!$', 13, 10, 0
@str_1 db 'Please, give the number
                of rings: ', 0
@str_2 db 'source', 0
@str_3 db 'target', 0
@str_4 db 'auxiliary', 0
@str_5 db 'Total moves: ', 0
@str_6 db 13, 10, 0
@str_7 db 'Move from ', 0
@str_8 db ' to ', 0

xseg ends
end main
```

5.4 Εκτενές παράδειγμα επίδειξης αντικειμενοστρεφών χαρακτηριστικών

Το επόμενο πρόγραμμα (`extended.oed`) δείχνει εκτενέστερα τη λειτουργία των αντικειμενοστρεφών μηχανισμών της Ooedsger, όπως φαίνεται και στα μηνύματα που πρόκειται να τυπωθούν στην οθόνη. Συγκεκριμένα, βλέπουμε τη χρήση των αντικειμενοστρεφών μηχανισμών των διαπροσωπειών, του ελέγχου υποτύπων, της κληρονομικότητας, των μετατροπών (upcasting, downcasting), του πολυμορφισμού (με τις κλήσεις δυναμικών μεθόδων), καθώς και της δημιουργίας αντικειμένων με κλήσεις στους κατασκευαστές.

```
class Hello
{
```

```
  static void main ()
  {
    Implementor i = impl_make("impl");
    i.hello_impl();

    System.writeString("-------interfaces------------\n");
    Helloable h;
    h = i;
    h.hello_impl();
    System.writeString("-------elegxos ypotypwn------------\n");
    if (i instanceof Implementor)
       System.writeString("Ok, ok you found me!\n");
    else
       System.writeString("Nope!\n");
    System.writeString("-------klhronomikothta------------\n");
    Yac y = yac_make("yac");
    y.write();
    System.writeString("-------downcasting------------\n");
    if (i instanceof Yac)
       System.writeString("Ok, ok you found me!\n");
    else
       System.writeString("Nope!\n");
    /*y = i;*/ // θα δώσει κατάλληλο μήνυμα σφάλματος
    System.writeString("-------upcasting------------\n");
    System.writeString("before casting\n");
    i.write();
    i = y;
    System.writeString("after casting\n");
    i.write();
    if (i instanceof Yac)
       System.writeString("Ok, ok you found me!\n");
    else
       System.writeString("Nope!\n");
    System.writeString("-------downcasting (allowed)------------\n");
    System.writeString("before casting\n");
    y.write();
    y = i;
    System.writeString("after casting\n");
    y.write();
    if (y instanceof Implementor)
       System.writeString("Ok, ok you found me!\n");
    else
       System.writeString("Nope!\n");
  }
}

class Yac extends Implementor
{
  constructor yac_make(char *name)
  {
    System.strcpy(this.name, name);
  }
}

class Implementor implements Helloable
{
  char name[30];
  constructor impl_make(char *name)
```

```
  {
    System.strcpy(this.name, name);
  }

  void hello_impl()
  {
    System.writeString("Xaire kosme!\n");
  }

  void write()
  {
    System.writeString(name);
    System.writeString("\n");
  }
}

interface Helloable
{
  void hello_impl();
}
```

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

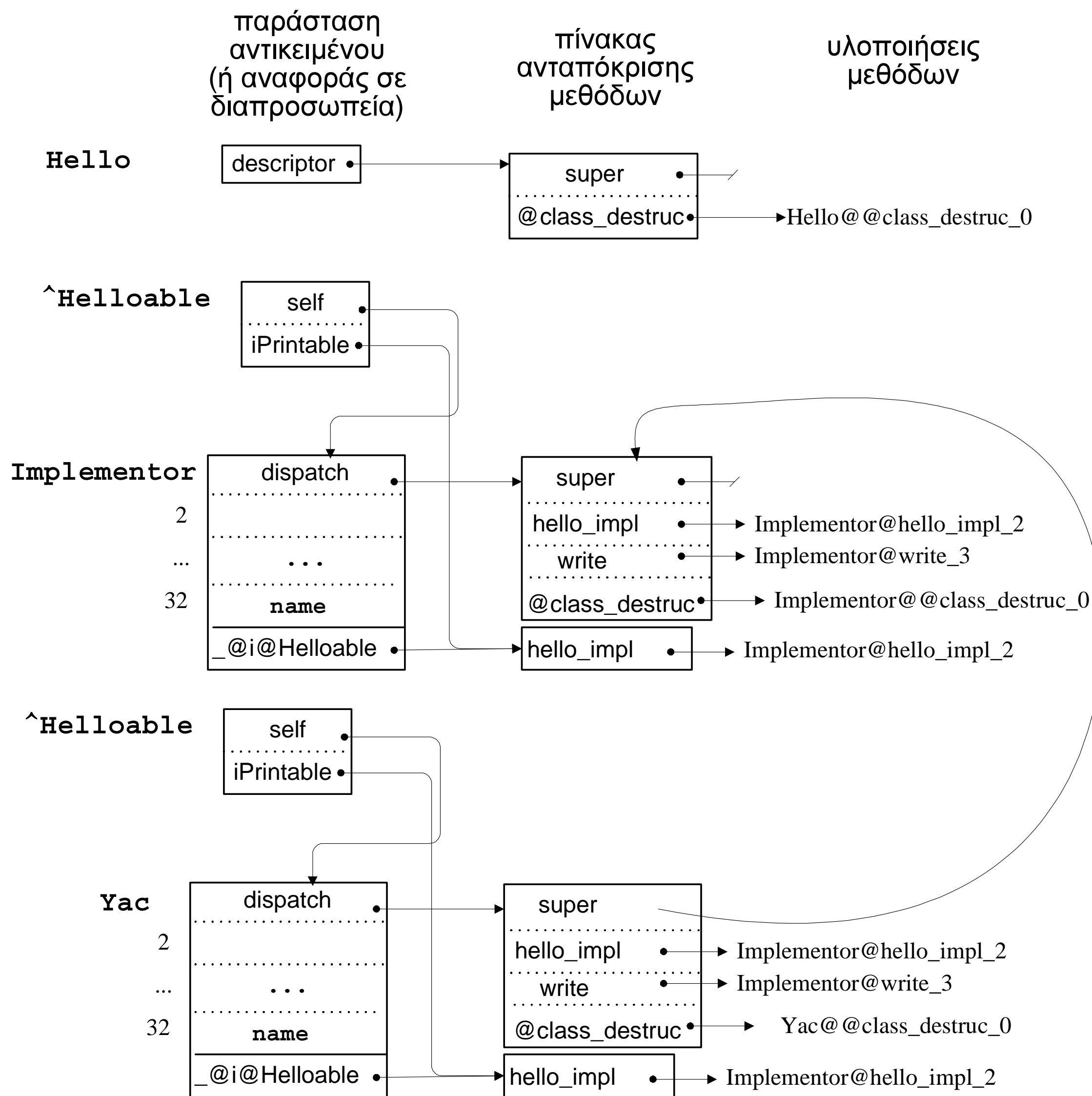
```
 1: unit, @class_destruc, -, -
 2: endu, @class_destruc, -, -
 3: unit, @class_destruc, -, -
 4: endu, @class_destruc, -, -
 5: unit, @class_destruc, -, -
 6: endu, @class_destruc, -, -
 7: unit, main, -, -
 8: par, "impl", V, -
 9: par, $1, RET, -
10: call, -, Implementor, impl_make
11: =, $1, -, i
12: call, -, i, hello_impl
13: par, "-------interfaces------------
          \n", V, -
14: call, -, System, writeString
15: =, i, -, h
16: call, -, h, hello_impl
17: par, "-------elegxos ypotypwn-------
          -----\n", V, -
18: call, -, System, writeString
19: instof, i, Implementor, 21
20: jump, -, -, 24
21: par, "Ok, ok you found me!\n", V, -
22: call, -, System, writeString
23: jump, -, -, 26
24: par, "Nope!\n", V, -
25: call, -, System, writeString
26: par, "-------klhronomikothta--------
          ----\n", V, -
27: call, -, System, writeString
28: par, "yac", V, -
29: par, $2, RET, -
30: call, -, Yac, yac_make
31: =, $2, -, y
32: call, -, y, write
33: par, "-------downcasting------------
          \n", V, -
34: call, -, System, writeString
35: instof, i, Yac, 37
36: jump, -, -, 40
37: par, "Ok, ok you found me!\n", V, -
38: call, -, System, writeString
39: jump, -, -, 42
40: par, "Nope!\n", V, -
41: call, -, System, writeString
42: par, "-------upcasting------------
          \n", V, -
43: call, -, System, writeString
44: par, "before casting\n", V, -
45: call, -, System, writeString
46: call, -, i, write
47: =, y, -, i
48: par, "after casting\n", V, -
49: call, -, System, writeString
50: call, -, i, write
51: instof, i, Yac, 53
52: jump, -, -, 56
53: par, "Ok, ok you found me!\n", V, -
54: call, -, System, writeString
55: jump, -, -, 58
56: par, "Nope!\n", V, -
57: call, -, System, writeString
58: par, "-------downcasting (allowed)-
          -----------\n", V, -
59: call, -, System, writeString
60: par, "before casting\n", V, -
61: call, -, System, writeString
62: call, -, y, write
63: =, i, -, y
64: par, "after casting\n", V, -
```

Σχήμα 5.6: Εγγραφήματα δραστηριοποίησης για το πρόγραμμα `extended.oed`.

```
65: call, -, System, writeString
66: call, -, y, write
67: instof, y, Implementor, 69
68: jump, -, -, 72
69: par, "Ok, ok you found me!\n", V, -
70: call, -, System, writeString
71: jump, -, -, 74
72: par, "Nope!\n", V, -
73: call, -, System, writeString
74: endu, main, -, -
75: unit, yac_make, -, -
76: dot, this, name, $3
77: =, $3, -, $4
78: par, $4, V, -
79: par, name, V, -
80: call, -, System, strcpy
81: endu, yac_make, -, -
82: unit, impl_make, -, -
83: dot, this, name, $5
84: =, $5, -, $6
85: par, $6, V, -
86: par, name, V, -
87: call, -, System, strcpy
88: endu, impl_make, -, -
89: unit, hello_impl, -, -
90: par, "Xaire kosme!\n", V, -
91: call, -, System, writeString
92: endu, hello_impl, -, -
93: unit, write, -, -
94: =, name, -, $7
95: par, $7, V, -
96: call, -, System, writeString
97: par, "\n", V, -
98: call, -, System, writeString
99: endu, write, -, -
```

Τα εγγραφήματα δραστηριοποίησης, καθώς και η παράσταση στη μνήμη των κλάσεων και των δεικτών στη διαπροσωπεία `Helloable` για το παραπάνω πρόγραμμα φαίνονται στα σχήματα 5.6 και 5.7 αντίστοιχα.

Ο τελικός κώδικας δίνεται παρακάτω:



Σχήμα 5.7: Παράσταση στη μνήμη των κλάσεων για το πρόγραμμα `extended.oed`.

```
xseg segment public 'code'
assume cs:xseg, ds:xseg, ss:xseg
org 100h
main proc near
call near ptr Hello@main_1
mov ax, 4C00h
int 21h
main endp


@1:

; 1: unit,@class_destruc,-,-

Hello@@class_destruc_0 proc near
push bp
mov bp, sp
sub sp, 0

@2:
```

```
; 2: endu,@class_destruc,-,-

@Hello@@class_destruc_0:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Hello@@class_destruc_0 endp


@3:

; 3: unit,@class_destruc,-,-
```

```
Implementor@@class_destruc_0 proc near
push bp
mov bp, sp
sub sp, 0

@4:

; 4: endu,@class_destruc,-,-

@Implementor@@class_destruc_0:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Implementor@@class_destruc_0 endp


@5:

; 5: unit,@class_destruc,-,-

Yac@@class_destruc_0 proc near
push bp
mov bp, sp
sub sp, 0

@6:

; 6: endu,@class_destruc,-,-

@Yac@@class_destruc_0:
mov si, word ptr [bp + 6]
push si
sub sp, 4
call near ptr __dispose
add sp, 6
mov si, word ptr [bp + 8]
mov word ptr [si], 0
mov sp, bp
pop bp
ret
Yac@@class_destruc_0 endp


@7:

; 7: unit,main,-,-

Hello@main_1 proc near
push bp
mov bp, sp
sub sp, 12

@8:

; 8: par,"impl",V,-
lea ax, byte ptr @str_1
push ax

@9:

; 9: par,$1,RET,-

lea ax, word ptr [bp -4]
push ax

@10:

; 10: call,-,Implementor,impl_make

sub sp, 2
push bp
call near ptr Implementor@impl_make_1
add sp, 8

@11:

; 11: =,$1,-,i

mov ax, word ptr [bp -4]
mov word ptr [bp -2], ax

@12:

; 12: call,-,i,hello_impl

sub sp, 2
mov si, word ptr [bp -2]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 2]
add sp, 6

@13:

; 13: par,"-------interfaces------------\n",
          V, -

lea ax, byte ptr @str_2
push ax

@14:

; 14: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@15:

; 15: =,i,-,h

lea si, word ptr [bp -2]
mov ax, word ptr [si]
mov si, word ptr [si]
```

```
mov bx, word ptr [si + 32]
mov word ptr [bp -8], bx
mov word ptr [bp -6], ax

@16:

; 16: call,-,h,hello_impl

sub sp, 2
lea si, word ptr [bp -8]
add si, 2
push si
push bp
mov si, word ptr [si - 2]
call word ptr [si + 0]
add sp, 6

@17:

; 17: par,"-------elegxos ypotypwn--------
                ----\n", V,-

lea ax, byte ptr @str_3
push ax

@18:

; 18: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@19:

; 19: instof,i,Implementor,21

lea ax, word ptr Implementor@descriptor
lea si, word ptr [bp -2]
_@_@again1: mov si, word ptr [si]
or si, si
jz _@_@nfound1
cmp si, ax
jnz _@_@again1
jz @21
_@_@nfound1:

@20:

; 20: jump,-,-,24

jmp @24

@21:

; 21: par,"Ok, ok you found me!\n",V,-

lea ax, byte ptr @str_4
push ax

@22:

; 22: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@23:

; 23: jump,-,-,26

jmp @26

@24:

; 24: par,"Nope!\n",V,-

lea ax, byte ptr @str_5
push ax

@25:

; 25: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@26:

; 26: par,"-------klhronomikothta----------
                --\n",V,-

lea ax, byte ptr @str_6
push ax

@27:

; 27: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@28:

; 28: par,"yac",V,-

lea ax, byte ptr @str_7
push ax

@29:

; 29: par,$2,RET,-

lea ax, word ptr [bp -12]
push ax

@30:

; 30: call,-,Yac,yac_make
```

```
sub sp, 2
push bp
call near ptr Yac@yac_make_1
add sp, 8

@31:

; 31: =,$2,-,y

mov ax, word ptr [bp -12]
mov word ptr [bp -10], ax

@32:

; 32: call,-,y,write

sub sp, 2
mov si, word ptr [bp -10]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@33:

; 33: par,"-------downcasting------------
                \n", V,-

lea ax, byte ptr @str_8
push ax

@34:

; 34: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@35:

; 35: instof,i,Yac,37

lea ax, word ptr Yac@descriptor
lea si, word ptr [bp -2]
_@_@again2: mov si, word ptr [si]
or si, si
jz _@_@nfound2
cmp si, ax
jnz _@_@again2
jz @37
_@_@nfound2:

@36:

; 36: jump,-,-,40

jmp @40

@37:

; 37: par,"Ok, ok you found me!\n",V,-

lea ax, byte ptr @str_4
push ax

@38:

; 38: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@39:

; 39: jump,-,-,42

jmp @42

@40:

; 40: par,"Nope!\n",V,-

lea ax, byte ptr @str_5
push ax

@41:

; 41: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@42:

; 42: par,"-------upcasting-----------\n",
                V,-

lea ax, byte ptr @str_9
push ax

@43:

; 43: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@44:

; 44: par,"before casting\n",V,-

lea ax, byte ptr @str_10
push ax

@45:
```

```
; 45: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@46:

; 46: call,-,i,write

sub sp, 2
mov si, word ptr [bp -2]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@47:

; 47: =,y,-,i

mov ax, word ptr [bp -10]
mov word ptr [bp -2], ax

@48:

; 48: par,"after casting\n",V,-

lea ax, byte ptr @str_11
push ax

@49:

; 49: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@50:

; 50: call,-,i,write

sub sp, 2
mov si, word ptr [bp -2]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@51:

; 51: instof,i,Yac,53

lea ax, word ptr Yac@descriptor
lea si, word ptr [bp -2]
_@_@again3: mov si, word ptr [si]
or si, si
jz _@_@nfound3
cmp si, ax
jnz _@_@again3
jz @53
_@_@nfound3:

@52:

; 52: jump,-,-,56

jmp @56

@53:

; 53: par,"Ok, ok you found me!\n",V,-

lea ax, byte ptr @str_4
push ax

@54:

; 54: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@55:

; 55: jump,-,-,58

jmp @58

@56:

; 56: par,"Nope!\n",V,-

lea ax, byte ptr @str_5
push ax

@57:

; 57: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@58:

; 58: par,"-------downcasting (allowed)----
                --------\n",V,-

lea ax, byte ptr @str_12
push ax

@59:

; 59: call,-,System,writeString

sub sp, 2
push bp
```

```
call near ptr _writeString
add sp, 6

@60:

; 60: par,"before casting\n",V,-

lea ax, byte ptr @str_10
push ax

@61:

; 61: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@62:

; 62: call,-,y,write

sub sp, 2
mov si, word ptr [bp -10]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@63:

; 63: =,i,-,y

lea ax, word ptr Yac@descriptor
lea si, word ptr [bp -2]
_@_@again4: mov si, word ptr [si]
or si, si
jz _@_@nfound4
cmp si, ax
jnz _@_@again4
jmp _@_@next1
_@_@nfound4:
call near ptr errordc
_@_@next1:
mov ax, word ptr [bp -2]
mov word ptr [bp -10], ax

@64:

; 64: par,"after casting\n",V,-

lea ax, byte ptr @str_11
push ax

@65:

; 65: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@66:

; 66: call,-,y,write

sub sp, 2
mov si, word ptr [bp -10]
push si
push bp
mov si, word ptr [si]
call word ptr [si + 4]
add sp, 6

@67:

; 67: instof,y,Implementor,69

lea ax, word ptr Implementor@descriptor
lea si, word ptr [bp -10]
_@_@again5: mov si, word ptr [si]
or si, si
jz _@_@nfound5
cmp si, ax
jnz _@_@again5
jz @69
_@_@nfound5:

@68:

; 68: jump,-,-,72

jmp @72

@69:

; 69: par,"Ok, ok you found me!\n",V,-

lea ax, byte ptr @str_4
push ax

@70:

; 70: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@71:

; 71: jump,-,-,74

jmp @74

@72:

; 72: par,"Nope!\n",V,-

lea ax, byte ptr @str_5
push ax
```

```
@73:

; 73: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@74:

; 74: endu,main,-,-

@Hello@main_1:
mov sp, bp
pop bp
ret
Hello@main_1 endp


@75:

; 75: unit,yac_make,-,-

Yac@yac_make_1 proc near
push bp
mov bp, sp
sub sp, 32
mov ax, 34
push ax
lea si, word ptr [bp + 6]
push si
sub sp, 2
call near ptr __new
add sp, 6
mov di, word ptr [bp + 6]
mov si, word ptr [bp + 8]
mov word ptr [si], di
lea si, word ptr Yac@descriptor
mov word ptr [di], si

@76:

; 76: dot,this,name,$3

mov ax, word ptr [bp + 6]
add ax, 2
mov word ptr [bp -30], ax

@77:

; 77: =,$3,-,$4

mov ax, word ptr [bp -30]
mov word ptr [bp -32], ax

@78:

; 78: par,$4,V,-

mov ax, word ptr [bp -32]
push ax

@79:

; 79: par,name,V,-

mov ax, word ptr [bp +10]
push ax

@80:

; 80: call,-,System,strcpy

sub sp, 2
push bp
call near ptr _strcpy
add sp, 8

@81:

; 81: endu,yac_make,-,-

@Yac@yac_make_1:
mov sp, bp
pop bp
ret
Yac@yac_make_1 endp


@82:

; 82: unit,impl_make,-,-

Implementor@impl_make_1 proc near
push bp
mov bp, sp
sub sp, 32
mov ax, 34
push ax
lea si, word ptr [bp + 6]
push si
sub sp, 2
call near ptr __new
add sp, 6
mov di, word ptr [bp + 6]
mov si, word ptr [bp + 8]
mov word ptr [si], di
lea si, word ptr Implementor@descriptor
mov word ptr [di], si
lea si, word ptr _@i@Helloable
mov word ptr [di + 32], si

@83:

; 83: dot,this,name,$5

mov ax, word ptr [bp + 6]
add ax, 2
mov word ptr [bp -30], ax

@84:

; 84: =,$5,-,$6

mov ax, word ptr [bp -30]
```

```
mov word ptr [bp -32], ax

@85:

; 85: par,$6,V,-

mov ax, word ptr [bp -32]
push ax

@86:

; 86: par,name,V,-

mov ax, word ptr [bp +10]
push ax

@87:

; 87: call,-,System,strcpy

sub sp, 2
push bp
call near ptr _strcpy
add sp, 8

@88:

; 88: endu,impl_make,-,-

@Implementor@impl_make_1:
mov sp, bp
pop bp
ret
Implementor@impl_make_1 endp


@89:

; 89: unit,hello_impl,-,-

Implementor@hello_impl_2 proc near
push bp
mov bp, sp
sub sp, 0

@90:

; 90: par,"Xaire kosme!\n",V,-

lea ax, byte ptr @str_13
push ax

@91:

; 91: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@92:

; 92: endu,hello_impl,-,-

@Implementor@hello_impl_2:
mov sp, bp
pop bp
ret
Implementor@hello_impl_2 endp


@93:

; 93: unit,write,-,-

Implementor@write_3 proc near
push bp
mov bp, sp
sub sp, 2

@94:

; 94: =,name,-,$7

mov si, word ptr [bp + 6]
add si, 2
lea ax, word ptr [si]
mov word ptr [bp -2], ax

@95:

; 95: par,$7,V,-

mov ax, word ptr [bp -2]
push ax

@96:

; 96: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@97:

; 97: par,"\n",V,-

lea ax, byte ptr @str_14
push ax

@98:

; 98: call,-,System,writeString

sub sp, 2
push bp
call near ptr _writeString
add sp, 6

@99:

; 99: endu,write,-,-
```

```
@Implementor@write_3:
mov sp, bp
pop bp
ret
Implementor@write_3 endp

errordc proc near
mov ah, 09h
lea dx, errstr
int 21h
mov ax, 4C00h
int 21h
errordc endp

Hello@descriptor dw 0
dw Hello@@class_destruc_0
Implementor@descriptor dw 0
dw Implementor@hello_impl_2
dw Implementor@write_3
dw Implementor@@class_destruc_0
_@i@Helloable dw Implementor@hello_impl_2
Yac@descriptor dw Implementor@descriptor
dw Implementor@hello_impl_2
dw Implementor@write_3
dw Yac@@class_destruc_0
_@i@Helloable dw Implementor@hello_impl_2

extrn __dispose : proc
extrn _writeString : proc
extrn __new : proc
extrn _strcpy : proc

errstr db 13, 10, 'Error in downcasting,
                exiting!$', 13,
                10, 0
@str_1 db 'impl', 0
@str_2 db '-------interfaces-----------
                -', 13, 10, 0
@str_3 db '-------elegxos ypotypwn-----
                -------', 13, 10, 0
@str_4 db 'Ok, ok you found me!', 13,
                10, 0
@str_5 db 'Nope!', 13, 10, 0
@str_6 db '-------klhronomikothta------
                ------', 13, 10, 0
@str_7 db 'yac', 0
@str_8 db '-------downcasting----------
                --', 13, 10, 0
@str_9 db '-------upcasting------------
                ', 13, 10, 0
@str_10 db 'before casting', 13, 10, 0
@str_11 db 'after casting', 13, 10, 0
@str_12 db '-------downcasting (allowed)
                ------------', 13, 10, 0
@str_13 db 'Xaire kosme!', 13, 10, 0
@str_14 db 13, 10, 0

xseg ends
end main
```

Κεφάλαιο 6

Συμπεράσματα και επεκτάσεις

Τα συμπεράσματα της παρούσας εργασίας συνοψίζονται στα παρακάτω:

- Υλοποιήθηκε ένας μεταγλωττιστής για μια γλώσσα αντικειμενοστρεφούς προγραμματισμού, που βασίζεται στο πρότυπο της Java και στο προστακτικό μοντέλο. Η υλοποίηση του μεταγλωττιστή περιελάμβανε τα στάδια του λεκτικού, συντακτικού και σημασιολογικού αναλυτή, καθώς και του γεννήτορα ενδιάμεσου και τελικού κώδικα. Μέσα από τα στάδια αυτά αποκομίσθηκε γόνιμη γνώση σχετικά με την εσωτερική υλοποίηση των αντικειμενοστρεφών (και μη) προγραμματιστικών δομών μέσα σε ένα μεταγλωττιστή.

- Έμφαση δόθηκε στην υποστήριξη ευέλικτων, αντικειμενοστρεφών χαρακτηριστικών, τα σημαντικότερα από τα οποία είναι: αντικείμενα, απλή κληρονομικότητα, δυνατότητα ορισμού κατασκευαστών με διαφορετικά ονόματα (μέσα στην ίδια κλάση), πολυμορφισμός υποτύπων (δυναμικό δέσιμο μεθόδων), έλεγχος υποτύπων, μετατροπές (upcasting, downcasting με κάποιους περιορισμούς), διαπροσωπείες. Παράλληλα, περιλαμβάνει χρήσιμες δομές προερχόμενες από το προστακτικό μοντέλο, όπως είναι οι τύποι δεικτών, η υπερφόρτωση μεθόδων, το πέρασμα παραμέτρων κατ' αναφορά. Ως γλώσσα προγραμματισμού για την υλοποίηση χρησιμοποιήθηκε η C.

Όπως είναι φυσικό, υπάρχουν πολλά περιθώρια επέκτασης του μεταγλωττιστή με νέα στοιχεία προς την κατεύθυνση της αύξησης της λειτουργικότητας και πρακτικότητάς του. Ενδεικτικά ορισμένα από τα σημαντικότερα είναι:

- Χρήση τεχνικών βελτιστοποίησης στον ενδιάμεσο ή/και στον τελικό κώδικα για αύξηση της ταχύτητας του μεταγλωττιστή και μείωση των απαιτήσεων στη μνήμη.

- Υλοποίηση αυτόματου *συλλέκτη σκουπιδιών* (garbage collector) για την αποδέσμευση μνήμης από αντικείμενα μη χρησιμοποιούμενα, δηλαδή από αντικείμενα που δεν έχουν ενεργές αναφορές (references) προς αυτά. Έτσι, δε θα χρειάζεται να απελευθερώνει τη μνήμη ο χρήστης-προγραμματιστής με κλήσεις μεθόδων καταστροφέων, αφού η εργασία αυτή θα γίνεται αυτόματα.

- Επέκταση των μετατροπών και στην περίπτωση που συμμετέχουν διαπροσωπείες με προς τα κάτω μετατροπή (downcasting), για ακόμα μεγαλύτερη ευελιξία.

- Υποστήριξη πολλαπλής κληρονομικότητας μεταξύ των κλάσεων για την διεύρυνση της κληρονομούμενης συμπεριφοράς με αύξηση βέβαια της πολυπλοκότητας της υλοποίησης του μεταγλωττιστή (και κατάργηση διαπροσωπειών). Εναλλακτικά και πιο απλά, μπορεί να προστεθεί η πολλαπλή κληρονομικότητα στην ιεραρχία των διαπροσωπειών.

- Δυνατότητα ομαδοποίησης σχετικών κλάσεων και διασυνδέσεων σε *πακέτα* (packages), οπότε ομάδες κλάσεων μπορούν να διατίθενται μόνο αν χρειάζονται, και έτσι εξαλείφονται πιθανές διενέξεις ανάμεσα σε ονόματα κλάσεων σε διάφορες ομάδες κλάσεων. Παράλληλα, μπορεί να προστεθεί ο έλεγχος προσπέλασης (δημόσια, προστατευμένη, ιδιωτική) για μεθόδους και μεταβλητές, επιτρέποντας ενθυλάκωση στα δεδομένα μιας κλάσης.

- Υποστήριξη *εξαιρέσεων* (exceptions) για την ασφαλή αντιμετώπιση λαθών ή απρόβλεπτων καταστάσεων, που καθιστούν αδύνατη τη συνέχιση του προγράμματος. Επίσης χρήσιμη θα ήταν η εισαγωγή του *ταυτοχρονισμού* (concurrency) στην Ooedsger, πιθανώς με χρήση *νημάτων* (threads), ώστε να δίνεται η δυνατότητα παράλληλης εκτέλεσης διαφόρων ενεργειών από διαφορετικά αντικείμενα-νήματα.

Βιβλιογραφία

[1] Ν. Σ. Παπασπύρου και Ε. Σ. Σκορδαλάκης. *Μεταγλωττιστές*. Εκδόσεις Συμμετρία, 2002.

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

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

[4] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998. Άλλες εκδόσεις του βιβλίου χρησιμοποιούν τις γλώσσες Java και ML.

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

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

[7] R. Cadenhead and L. Lemay. *Teach Yourself Java in 21 days*. Sams Publishing, 2003.

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

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

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

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

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

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