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

Υλοποίηση μιας Γλώσσας Χαμηλού Επιπέδου με Υποστήριξη Αυτομεταβαλλόμενου Κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΙΩΡΓΟΣ Α. ΦΟΥΡΤΟΥΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Αθήνα, Νοέμβριος 2005

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

Υλοποίηση μιας Γλώσσας Χαμηλού Επιπέδου με Υποστήριξη Αυτομεταβαλλόμενου Κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΙΩΡΓΟΣ Α. ΦΟΥΡΤΟΥΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Δεκεμβρίου 2005.

.....	.....	.....
Νικόλαος Παπασπύρου Λέκτορας Ε.Μ.Π.	Ευστάθιος Ζάχος Καθηγητής Ε.Μ.Π.	Ανδρέας-Γεώργιος Σταφυλοπάτης Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2005

.....

Γιώργος Α. Φουρτούνης

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

Περίληψη

Σκοπός της διπλωματικής αυτής εργασίας είναι η δημιουργία μιας προστακτικής γλώσσας χαμηλού επιπέδου με δυνατότητα συγγραφής σε αυτήν αυτομεταβαλλόμενου κώδικα (self modifying code) καθώς και ενός πρότυπου διερμηνέα (interpreter) για αυτήν. Η γλώσσα αυτή ονομάζεται AMK και σχεδιάστηκε από την αρχή με σκοπό να αποτελέσει ένα μικρό υποσύνολο μίας κλασικής προστακτικής (imperative) γλώσσας, στην οποία έχει προστεθεί αυτό το χαρακτηριστικό, χωρίς όμως να χάνει σε ικανότητα υπολογισμού και σε ασφάλεια.

Ως αυτομεταβαλλόμενος ορίζεται ο κώδικας ο οποίος έχει τη δυνατότητα να μεταβάλλει την αναπαράστασή του στη μνήμη τη στιγμή που εκτελείται χρησιμοποιώντας ειδικές εντολές της γλώσσας AMK. Η μεταβολή αυτή επιτυγχάνεται με την αντιμετώπιση του κώδικα ως δεδομένου από το πρόγραμμα και το χειρισμό του μέσω μεταβλητών που τον περιέχουν.

Σημαντική παράμετρο στο σχεδιασμό της γλώσσας AMK και του διερμηνέα της αποτέλεσε η απαίτηση τα προγράμματα που μπορούν να γραφούν σε αυτή να είναι ασφαλή όσον αφορά το χειρισμό του κώδικα. Έγινε μια προσπάθεια να αντιμετωπιστούν κάποια προβλήματα που συναντώνται στις μέχρι τώρα απόπειρες συγγραφής αυτομεταβαλλόμενου κώδικα σε γλώσσες χαμηλού επιπέδου (γλώσσα μηχανής, συμβολικές γλώσσες), ώστε να παρέχονται εγγυήσεις για τη συμπεριφορά των προγραμμάτων σε AMK.

Επίσης γίνεται μια αναδρομή σε συστήματα που έχουν ήδη κυκλοφορήσει και υποστηρίζουν ορισμένα ή όλα τα χαρακτηριστικά που θα μελετηθούν σε αυτήν την εργασία.

Τέλος η εργασία συμπληρώνεται από ορισμένες εφαρμογές σε AMK ώστε με αυτόν τον τρόπο να δειχτεί μία άλλη προσέγγιση σε γνωστούς αλγόριθμους.

Λέξεις κλειδιά

Αυτομεταβαλλόμενος κώδικας, AMK, γεννήτορας κώδικα, ασφάλεια, OCaml.

Abstract

The purpose of this diploma dissertation is the design of a low-level imperative language which supports self modifying code, as well as the implementation of a prototype interpreter. This language is called AMK and was designed from scratch, aiming to implement a rather small subset of the instruction set of a classic imperative language, enhanced with the ability of self modification. This ability however should not come at the price of losing computation power or security.

The term "self modifying code" refers to code that is able to modify its representation in memory at run-time, using special instructions of the AMK language. This modification is achieved by treating code as data and manipulating it through ordinary variables that contain it.

Keeping the manipulation of code as safe as possible has been a strict design goal of the AMK language and its interpreter. An effort has been made to minimize the usual problems that occur in self modifying code, written until now in low-level languages (machine code, assembly), in such a way that safe execution of self modifying code in AMK is guaranteed.

This dissertation also includes a brief survey of existing systems currently in use that support some or all language characteristics studied here.

In the end, some example programs in AMK are shown that demonstrate a new perspective on some old algorithms.

Key words

Self modifying code, AMK, code generator, security, OCaml.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή αυτής της διπλωματικής εργασίας κ. Νίκο Παπασπύρου για το χρόνο που μου αφιέρωσε κατά τη διάρκεια της συγγραφής της και την πολύτιμη βοήθειά του στα ζητήματα που προέκυψαν κατά τη μελέτη του αντικειμένου της. Επίσης θα ήθελα να ευχαριστήσω τους γονείς μου που μου πρόσφεραν πολύτιμη στήριξη όλα αυτά τα χρόνια της φοίτησής μου στο Ε.Μ.Π.

Γιώργος Α. Φουρτούνης

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-8-05, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Νοέμβριος 2005.

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

Περιεχόμενα

Πίνακες

Σχήματα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σκοπός αυτής της διπλωματικής εργασίας είναι η κατασκευή μιας γλώσσας αυτομεταβαλλόμενου κώδικα και ενός πρότυπου διερμηνέα για αυτήν. Η γλώσσα αυτή (AMK) είναι υποσύνολο μιας τυπικής προστακτικής γλώσσας, έμφαση όμως δε θα δοθεί σε αυτά τα χαρακτηριστικά της που άλλωστε έχουν μελετηθεί διεξοδικά σε άλλες γλώσσες τις τελευταίες δεκαετίες. Αντίθετα, η παρούσα μελέτη θα επικεντρωθεί σε χαρακτηριστικά της που επιτρέπουν την παρέμβαση στον κώδικα ενός προγράμματος από το ίδιο το πρόγραμμα τη στιγμή που εκτελείται, συμπεριφορά που συγκριτικά έχει μελετηθεί λιγότερο. Η δυνατότητα αυτή ονομάζεται αυτομεταβαλλόμενος κώδικας και εμφανίζει ιδιαίτερες απαιτήσεις, τόσο από τη γλώσσα που χρησιμοποιείται για την εφαρμογή του, όσο και από τον προγραμματιστή που θα τον χρησιμοποιήσει. Στη συνέχεια θα γίνει μια περιγραφή της φύσης του και οι ιδιαίτερες ιδιότητες και απαιτήσεις που συνεπάγεται η χρήση του θα μελετηθούν στην πράξη μέσω κάποιου κατάλληλου διερμηνέα που έχει κατασκευαστεί για αυτήν την περίσταση. Φυσικά θα αιτιολογηθεί ο τρόπος κατασκευής της γλώσσας AMK και του διερμηνέα της, ώστε να συναχθούν κάποια χρήσιμα συμπεράσματα για τις δυνατότητες της χρήσης αυτομεταβαλλόμενου κώδικα σε πραγματικές, μη τετριμμένες, εφαρμογές. Για αυτό το λόγο, αρκετά σημαντική κρίνεται η επίδειξη κάποιων παραδειγμάτων που να δοκιμάζουν τις ιδιότητες αυτές.

1.2 Κίνητρο, ιστορική αναδρομή και εφαρμογές

Όπως φαίνεται από την ιστορία των γλωσσών προγραμματισμού, η πρόοδος των μεθόδων ανάπτυξης και μελέτης των προγραμμάτων που γράφονται σε αυτές, δείχνει μία κοινή πορεία με την ανάπτυξη του υλικού (hardware) και των περιβαλλόντων στα οποία αναπτύσσονταν (λειτουργικά συστήματα). Η πορεία από τη γλώσσα μηχανής σε συμβολική γλώσσα και στη συνέχεια σε γλώσσες ολοένα και υψηλότερου επιπέδου (προστακτικές, αντικειμενοστρεφείς, συναρτησιακές, κλπ.) είχε ως αποτέλεσμα να δημιουργηθεί ένα πλήθος νέων χαρακτηριστικών των γλώσσών και μία καλύτερη κατανόηση των αρχών που διέπουν τη σωστή λειτουργία ενός προγράμματος επομένως και την εγγύηση που παρέχεται στον προγραμματιστή για την ποιότητα των δημιουργημάτων του.

Η ασφάλεια αυτή δεν ήταν πάντα το πρώτο μέλημα ενός σχεδιαστή μίας γλώσσας προγραμματισμού. Σε παλαιότερες εποχές, όταν το υλικό των υπολογιστών είχε αρκετά μικρότερες δυνατότητες από σήμερα και τα μέσα για την ανάπτυξη ενός προγράμματος ήταν πιο περιορισμένα (μικρή μνήμη και ισχύς, λιγότερα προγραμματιστικά εργαλεία) πολλές φορές επιλεγόταν μία γλώσσα χαμηλού επιπέδου για τη συγγραφή του, ώστε να γίνεται αποδοτικότερη χρήση του υλικού. Μια ενδιαφέρουσα επίπτωση αυτής της γενικευμένης χρήσης γλωσσών χαμηλού επιπέδου ήταν η συνειδητοποίηση από μέρους του προγραμματιστή της εγγενούς σχέσης δεδομένων και κώδικα και η χρήση μεθόδων αρχικά προοριζόμενων για τα πρώτα στον δεύτερο και αντίστροφα. Οι υπολογιστές ακολουθούν το πρότυπο του von Neumann, σύμφωνα με το οποίο τα περιεχόμενα της μνήμης του υπολογιστή δεν παριστάνουν είτε ξεχωριστά κώδικα είτε

δεδομένα αλλά εξαρτάται από τον τρόπο που θα τα χειριστεί κάποιος, δηλαδή υπάρχει κοινή αναπαράσταση και των δύο. Το γεγονός αυτό αποκαλύπτεται στις γλώσσες χαμηλού επιπέδου από τη δυνατότητα ενός προγράμματος να διαβάζει οποιοδήποτε στοιχείο της μνήμης και να γράψει σε αυτό ανεξάρτητα των περιεχομένων του. Έτσι κάποιος θα μπορούσε να γράψει ένα πρόγραμμα που διαβάζει από τη μνήμη την ίδια του την αναπαράσταση και μεταβάλλει οποιοδήποτε τμήμα της είναι επιθυμητό. Όμως η πρακτική αυτή, ο αυτομεταβαλλόμενος κώδικας, αν και ήταν γνωστή και γινόταν χρήση της, ποτέ δεν απέκτησε θεωρητικά θεμέλια και συνήθως εφαρμοζόταν μόνο σε ειδικές περιπτώσεις, λόγω της αποδοσής της. Μια τέτοια περίπτωση ήταν η συγγραφή προγραμμάτων για συστήματα με μικρή μνήμη, όπου ο AMK έδινε τη δυνατότητα σε ένα μικρό πρόγραμμα να πάρει μόνο τη μορφή που θα του ήταν χρήσιμη για μία συγκεκριμένη εκτέλεσή του και άρα να διαγράψει τα τμήματα που θα έπιαναν άδικα χώρο. Επίσης χρησιμοποιήθηκε σε προγράμματα που έκαναν συμπίεση στον εαυτό τους (packers, εφαρμογή συγγενής με την προηγούμενη) ή σε εφαρμογές προστασίας εμπορικών εφαρμογών, ώστε να δυσκολέψουν την ανάλυση ενός τμήματος κώδικα με μεθόδους αντίστροφης μηχανικής (reverse engineering) από τη γλώσσα μηχανής του.

Τέτοιες εφαρμογές όμως έμειναν ως επι το πλείστον περιθωριακές και πολύ λίγη επιρροή είχαν στην εξέλιξη των γλωσσών προγραμματισμού, οι οποίες μάλιστα και παρέλειψαν να προσθέσουν τέτοιες ικανότητες στα χαρακτηριστικά τους. Η παράλειψη αυτή οφείλεται στη χαμηλή αξιοπιστία του αυτομεταβαλλόμενου κώδικα όταν αυτός γράφεται σε γλώσσα μηχανής ή συμβολική γλώσσα. Είναι γνωστό ότι παρά την ισχύ τους τέτοιες γλώσσες αποκαλύπτουν στον προγραμματιστή όλο το υλικό χωρίς να τον διευκολύνουν σε θέματα διαχείρισής του και δεν τον προστατεύουν από λάθος χρήση του, με συνέπεια να είναι επιρρεπής σε λάθη. Έτσι ο αυτομεταβαλλόμενος κώδικας δε μελετήθηκε όπως άλλα γλωσσικά χαρακτηριστικά, αντιμετωπίστηκε ως ένας πρόχειρος τρόπος να υλοποιηθούν ορισμένα προγράμματα όταν δεν υπήρχε εναλλακτική λύση και αποθαρρύνθηκε η χρήση του. Είναι ενδεικτική η στάση του E. W. Dijkstra που αναφερόμενος στον αυτομεταβαλλόμενο κώδικα λέει ότι ήταν ένα αστείο που ευθύνεται για μία δεκαετία σύγχυσης [1].

Στη συνέχεια η ανάπτυξη νέων γλωσσών και η μελέτη άλλων χαρακτηριστικών έθεσε τελείως στο περιθώριο το χαρακτηριστικό του αυτομεταβαλλόμενου κώδικα. Οι νέες γλώσσες είναι πιο υψηλού επιπέδου, μακριά από το υλικό και σκοπό είχαν να διευκολύνουν τον προγραμματιστή στο χειρισμό των δεδομένων και να ενθαρρύνουν πιο ασφαλείς και αυστηρές μεθόδους ανάπτυξης. Ο κώδικας πιά δεν είναι προσβάσιμος στο πρόγραμμα, αφού ο μεταγλωττιστής ή διερμηνέας που ήταν υπεύθυνος για τη δημιουργία του, τον προστατεύει από μη συνεπείς αλλαγές που δεν προβλέπονται από το πλαίσιο της γλώσσας για την οποία έχει γραφεί. Οι επεξεργαστικοί πόροι δεν αποτελούν τόσο μεγάλο πρόβλημα και βρέθηκαν άλλες μέθοδοι για την αντιμετώπιση των προβλημάτων του παρελθόντος. Για παράδειγμα η μικρή μνήμη αντιμετωπίστηκε με αρθρωτά συστήματα (modular systems), στα οποία ο κώδικας φορτώνεται στη μνήμη μόνο τη στιγμή που χρειάζεται από δυναμικές βιβλιοθήκες. Εξαίρεση αποτελούν εφαρμογές, όπως οι αυτομεταβαλλόμενοι ιοί (μία προσπάθεια να αλλοιωθεί η χαρακτηριστική υπογραφή τους ώστε να μην εντοπίζονται από τα αντιβιοτικά) ή η υλοποίηση διερμηνέων όπως της Haskell [18] που μετέβαλλαν τον κώδικα στη μνήμη που καλούσε μία συνάρτηση όταν μία τιμή της υπολογιζόταν, ώστε να υπάρχει ένα κέρδος στην ταχύτητα εκτέλεσης.

Όμως ο αυτομεταβαλλόμενος κώδικας, ως μοντέλο είναι αρκετά ισχυρός και με τα σημερινά δεδομένα πρέπει να γίνει μία απόπειρα επανεξέτασής του. Έχει μελετηθεί θεωρητικά [11, 8] και η δυνατότητα ενσωμάτωσής του σε κάποια γλώσσα πιο υψηλού επιπέδου με ασφαλή όμως τρόπο είναι θεμιτή και θα μελετηθεί στη συνέχεια της εργασίας αυτής.

Το μειονέκτημά του στον παρελθόν αποτέλεσε η στενή του σχέση με το υλικό σε χαμηλό επίπεδο, σχέση που υπαγορεύθηκε σε μεγάλο βαθμό από τη σχεδίαση της μηχανής και τους περιορισμούς των τότε υπολογιστών. Η απάντηση σε αυτό είναι η μεταφορά του σε ένα περιβάλλον υψηλότερου επιπέδου, που περιγράφεται αυστηρά από μια γλώσσα και υπακούει σε ορισμένες απαιτήσεις. Τέτοια περιβάλλοντα, εντελώς αποκομμένα από το υλικό είναι οι ει-

κονικές μηχανές (Virtual Machines, VMs ή sandboxes) που όπως έχουν αποδείξει σε πολλές περιπτώσεις (γλώσσες Java, C#) μπορούν να παρέχουν μία πλήρως αφηρημένη αλλά αποδοτική εικόνα του υλικού με τον κώδικα να τρέχει σε μια ειδική μορφή (bytecodes) που διατηρεί την επιθυμητή ασφάλεια και συνέπεια.

1.3 Σύνοψη

Σε αυτό το κεφάλαιο έγινε μια μικρή εισαγωγή στο ζήτημα της επανεξέτασης της χρήσης του αυτομεταβαλλόμενου κώδικα σε εφαρμογές και μια ιστορική αναδρομή προσπάθησε να σκιαγραφήσει τους λόγους για τους οποίους σήμερα πρέπει να μελετηθεί.

- Στο Κεφάλαιο 2 θα μελετηθεί εκτενέστερα η φύση του αυτομεταβαλλόμενου κώδικα και τα ιδιαίτερα χαρακτηριστικά του. Θα εξεταστούν σχετικές εφαρμογές σε κάποια πεδία της πληροφορικής και θα βρεθούν τα προαπαιτούμενα ώστε ένα σύστημα να μπορεί να θεωρηθεί ότι τον υποστηρίζει με κάποια μορφή.
- Στο Κεφάλαιο 3 θα γίνει μία πλήρης περιγραφή της γλώσσας που θα χρησιμοποιηθεί σε αυτήν την εργασία: της AMK.
- Στο Κεφάλαιο 4 σειρά έχει η περιγραφή της υλοποίησης του πρότυπου διερμηνέα (amk) για τη γλώσσα αυτή. Θα αναφερθούν και θα εξηγηθούν οι εσωτερικές δομές του και ο τρόπος με τον οποίο λειτουργεί.
- Στο Κεφάλαιο 5 θα παρουσιαστούν κάποια παραδείγματα που επιδεικνύουν την αλλαγή στον τρόπο διατύπωσης ορισμένων προγραμμάτων υπό το πρίσμα των δυνατοτήτων αυτομεταβαλλόμενου κώδικα της γλώσσας AMK.
- Στο Κεφάλαιο 6 θα αναφερθούν τα συμπεράσματα που έχουν εξαχθεί από αυτήν τη μελέτη.

Κεφάλαιο 2

Αυτομεταβαλλόμενος κώδικας

2.1 Χαρακτηριστικά μίας γλώσσας αυτομεταβαλλόμενου κώδικα

Η χρήση αυτομεταβαλλόμενου κώδικα είναι άμεσα συνδεδεμένη με το ζήτημα της δημιουργίας δυναμικού κώδικα από ένα πρόγραμμα στο χρόνο εκτέλεσής του. Χωρίς αυτήν τη βασική δυνατότητα, ένα πρόγραμμα δε θα μπορούσε παρά μόνο να μεταβάλλει τα ίδια τα δεδομένα της μνήμης χρησιμοποιώντας αποκλειστικά κώδικα που αυτή ήδη έχει. Αυτό οδηγεί σε ένα πιο περιορισμένο μοντέλο αυτομεταβαλλόμενου κώδικα με σαφώς μικρότερη ισχύ που επιβάλλει ισχυρότερους περιορισμούς στη μορφή ενός προγράμματος.

Επομένως κρίσιμο σημείο για την εξέταση του τρόπου με τον οποίο λειτουργεί ο αυτομεταβαλλόμενος κώδικας είναι η εξοικείωση με τον τρόπο δημιουργίας νέων εντολών από ένα πρόγραμμα. Οι τρόποι με τους οποίους είναι ικανό ένα πρόγραμμα να δημιουργεί και στη συνέχεια να εκτελεί δυναμικά νέα τμήματα κώδικα καθώς εκτελείται είναι οι εξής:

- Δημιουργία προγραμμάτων(program generation) και τρέξιμο αυτών. Αυτή η δυνατότητα συνήθως ταυτίζεται με την υποστήριξη από την πλευρά της γλώσσας μιας εντολής που δέχεται κώδικα σε κάποια συγκεκριμένη μορφή και τον εκτελεί, όπως η εντολή eval() της Perl. Όμως γενικά μπορεί να δειχτεί ότι η δυνατότητα αυτή παρέχεται και στις γλώσσες που δεν έχουν τέτοια εντολή στο συντακτικό τους αλλά αρκεί να υποστηρίζουν το δυναμική φόρτωση κώδικα τη στιγμή της εκτέλεσης, για παράδειγμα μέσω δυναμικών βιβλιοθηκών (Dynamic Linked Libraries, DLLs) [6].

- Τη γενικότερη ικανότητα όχι μόνο να δημιουργεί όπως παραπάνω νέα τμήματα κώδικα και να τα τρέχει αλλά να μπορεί να αλλάξει την ίδια την αναπαράστασή του στη μνήμη και να εκτελέσει πάλι κάποιο τμήμα του που είχε ήδη εκτελεσθεί, αυτή τη φορά τροποποιημένο. Αυτή η δυνατότητα είναι προφανώς υπερσύνολο της προηγούμενης γιατί ένα πρόγραμμα που μεταβάλλει κάποιο τμήμα του κώδικά του που δεν έχει ακόμα εκτελεσθεί αλλά ακολουθεί αμέσως μετά τις εντολές που έχουν ως συνέπεια αυτή τη μεταβολή, είναι ισοδύναμο με ένα πρόγραμμα που δημιουργεί κάτι και στη συνέχεια το εκτελεί.

  Επίσης φαίνεται ότι και το αντίστροφο είναι αληθές, δηλαδή μία γλώσσα με δυνατότητα ΑΜΚ μπορεί να προσομοιωθεί από μια γλώσσα με ικανότητα δημιουργίας κώδικα, αν θεωρήσουμε ότι κάθε φορά ο νεος κώδικας δημιουργείται έτσι ώστε να είναι αντίγραφο του ήδη υπάρχοντος στη μνήμη και στη συνέχεια τροποποιείται και εκτελείται. Βέβαια προκύπτει το ζήτημα αν ένα πρόγραμμα μπορεί να έχει την ικανότητα να χειριστεί τον ισοδύναμο κώδικα που αντιστοιχεί στη λειτουργικότητά του. Στην πράξη υπάρχει μια κατηγορία προγραμμάτων (quines) που μπορούν να εκτυπώνουν τον πηγαίο τους κώδικα υπό τη μορφή συμβολοσειράς. Όμως οι περισσότερες εντολές που υποστηρίζουν την εντολή eval(), θεωρούν ότι ο κώδικας δίνεται σε αυτές με τη μορφή κειμένου, άρα τελικά υπάρχουν προγράμματα που μπορούν να εκτελέσουν αντίγραφα του εαυτού τους που τα παράγουν μόνα τους σε κάποια ενδιάμεση μορφή (εδώ συμβολοσειρές) τη στιγμή της εκτέλεσής τους.

Επομένως αποδεικνύεται ότι τα δύο παραπάνω μοντέλα είναι διαφορετικές προσεγγίσεις στο ίδιο υπολογιστικό μοντέλο. Για απλότητα και για συμφωνία με το περιεχόμενο της εργασίας, αυτό το μοντέλο θα ονομάζεται στη συνέχεια μοντέλο AMK και θα εννοεί κυρίως τη δεύτερη προσέγγιση στο θέμα του δυναμικού κώδικα. Η πρώτη προσέγγιση θα σημειώνεται επιπλέον όταν πρόκειται να χρησιμοποιηθεί.

2.1.1 Σύστημα χρόνου εκτέλεσης

Οι γλώσσες προγραμματισμού που υποστηρίζουν αυτομεταβαλλόμενο κώδικα είναι συνήθως βασισμένες σε διερμηνείς (SNOBOL4, LISP). Η ίδια η φύση του αυτομεταβαλλόμενου κώδικα επιβάλλει μια υψηλού επιπέδου αλληλεπίδραση με το σύστημα εκτέλεσης (run-time system) λόγω της απαίτησης για παραγωγή δυναμικού κώδικα τη στιγμή που ζητείται (on the fly) και όχι κάποια προγενέστερη στιγμή, όπως στην περίπτωση μιας μεταγλωττιζόμενης γλώσσας. Προσπάθειες για αυτομεταβαλλόμενο κώδικα έχουν γίνει και σε μεταγλωττιστές βέβαια (Clipper, Spitbol) αλλά είναι πιο δύσκολο να υλοποιηθούν λόγω της στατικής φύσης του κώδικα που δημιουργούν. Η απαίτηση υποστήριξης χαρακτηριστικών αυτομεταβαλλόμενου κώδικα ενός συστήματος με χρήση κάποιας γλώσσας υψηλού επιπέδου (άρα αποκλείονται οι συμβολικές γλώσσες) έχει φυσικό επακόλουθο την απαίτηση ύπαρξης ενός ισχυρού συστήματος εκτέλεσης. Το σύστημα αυτό πρέπει να μπορεί να αποκαλύπτει στον προγραμματιστή τα εργαλεία για τη δημιουργία νέου κώδικα και την τοποθέτησή του σε μια θέση στη μνήμη και να του επιτρέπει πρόσβαση σε όσο περισσότερες δυνατότητες τροποποίησης ενός προγράμματος από τον ίδιο του τον εαυτό. Αυτή η δυνατότητα όμως δεν πρέπει να το αφήνει εκτεθειμένο σε μη έγκυρες η επικίνδυνες τροποιήσεις της μνήμης.

Οι μέθοδοι με τις οποίες γίνεται ο χειρισμός του κώδικα από την ίδια τη γλώσσα είναι κυρίως δυο, δυναμική παραγωγή νέων ατομικών εντολών και ο χειρισμός του προγράμματος στη μνήμη.

Δυναμική παραγωγή νέων εντολών

Η υποστήριξη δυναμικής παραγωγής νέων ατομικών εντολών στη διάρκεια του προγράμματος με τη χρήση παραμετρικών εντολών κατασκευαστών [14] για αυτές. Με τους κατασκευαστές, το πρόγραμμα είναι ικανό να παράγει οποιαδήποτε εντολή και να την τοποθετήσει στη μνήμη, ανάλογα με την παλιά μέθοδο της γλώσσας μηχανής να τοποθετεί bytes στη μνήμη που αντιστοιχούν σε συγκεκριμένες εντολές. Η διαφορά βέβαια αυτής της μεθόδου με τη γλώσσα μηχανής είναι ότι είναι ασφαλέστερη επειδή οι κατασκευαστές είναι συγκεκριμένοι και άρα δεν είναι δυνατόν κάποιος να δημιουργήσει λάθος ή ελλιπείς εντολές που δεν προβλέπονται από τη γλώσσα. Έτσι κάθε πρόγραμμα ανάγεται σε έναν δημιουργό κώδικα (program generator) που έχει τη δυνατότητα να τροποποιεί οποιοδήποτε τμήμα του είναι επιθυμητό.

Μεταβολή στη μνήμη

Η υποστήριξη μεταβολής του προγράμματος στη μνήμη μέσω συγκεκριμένου μηχανισμού τροποποίησης του προγράμματος σε μορφή υψηλού επιπέδου. Η προσέγγιση αυτή θα ακολουθηθεί στην παρούσα εργασία και είναι πιο εύκολο να μελετηθεί γιατί τα χαρακτηριστικά που επιτρέπουν κώδικα που μεταβάλλει τον εαυτό του είναι λιγότερα σε αριθμό. Σε αυτήν την περίπτωση η γλώσσα είναι ένα υποσύνολο μιας τυπικής προστακτικής γλώσσας με επιπλέον εντολές που είτε ορίζουν έτοιμα τμήματα κώδικα, είτε τα χρησιμοποιούν για να κατασκευάσουν στη μνήμη το ζητούμενο πρόγραμμα. Εδώ η έμφαση δε δίνεται τόσο στη δημιουργία των εντολών, όσο στο χειρισμό υπάρχουσων εντολών για τη δημιουργία κατευθείαν στη μνήμη νέου κώδικα.

Το πρόγραμμα σε αυτήν την περίπτωση αναπαρίσταται στη μνήμη με μια μορφή υψηλού επιπέδου (συνήθως κάποιο συντακτικό δέντρο) και οι εντολές έχουν τη δυνατότητα να μεταβάλλουν τους κόμβους του, μεταφέροντας μεταξύ αυτών υποδένδρα-τμήματα κώδικα. Επιπλέον είναι δυνατή η σημείωση ορισμένων κόμβων ως σημεία εισαγωγής για νέα τμήματα εντολών.

Οι δυνατότητες που προσφέρει αυτή η γλώσσα προφανώς είναι ίδιες με την προηγούμενη γιατί οι κατασκευαστές των εντολών μπορούν να προσομοιωθούν με ισοδύναμα έτοιμα τμήματα κώδικα, ένα για κάθε εντολή και με σημεία εισαγωγής των παραμέτρων που χρειάζονται κάθε φορά. Σε τελική ανάλυση είναι γνωστό ότι ο αυτομεταβαλλόμενος κώδικας δεν έχει παραπάνω δυνατότητες από το συνηθισμένο μη-μεταβαλλόμενο κώδικα, διότι είναι ένα πλήρες κατά Turing υπολογιστικό μοντέλο (Turing complete).

2.1.2 Τροποποίηση μνήμης

Στην περίπτωση της γλώσσας AMK και κάθε γλώσσας που επιτρέπει την τροποποίηση της αναπαράστασης του προγράμματος στη μνήμη, τίθεται το πρόβλημα της τροποποίησης κώδικα που εκτελείται αλλά βρίσκεται σε κάποιο ενδιάμεσο στάδιο της εκτέλεσής του.

Πιο συγκεκριμένα, έστω το παράδειγμα κάποιας γλώσσας που υποστηρίζει αυτομεταβαλλόμενο κώδικα και συναρτήσεις (functions) στα προγράμματά της. Ένα πρόγραμμα σε αυτή θα είναι ένα σύνολο εντολών σε χαμηλό επίπεδο που θα εκτελούνται σειριακά, μια εντολή κάθε φορά. Η σειρά εκτέλεσής τους ορίζεται από κάποιο μετρητή προγράμματος, όπως ο καταχωρητής IP για τη γλώσσα μηχανής των επεξεργαστών αρχιτεκτονικής x86 της Intel, ο αριθμός εντολής για ορισμένες παλιές εκδόσεις της BASIC ή κάποιο άλλο πιο σύνθετο μέγεθος, μοναδικό για κάθε εντολή στη μνήμη (όπως ο αύξων αριθμός του αντίστοιχου για την κάθε εντολή φύλλου του συντακτικού δένδρου). Έστω ότι το πρόγραμμα ορίζει δύο συναρτήσεις f, g και έστω ότι η f καλεί την g στο μέσο της εκτέλεσής της. Αν η g με κάποια κατάλληλη εντολή έχει τη δυνατότητα να μεταβάλλει το περιεχόμενο της f στη μνήμη, τότε προφανώς μπορεί να προκαλέσει καταστάσεις δύσκολες να προβλεφθούν. Μπορεί να αλλάξει μεταβλητές από τις οποίες εξαρτάται και η ίδια, να διαγράψει το σημείο της f από το οποίο κλήθηκε ή να καταστρέψει το περιβάλλον της ώστε να είναι αδύνατο να τερματίσει και να επιστρέψει στην f με έγκυρα αποτελέσματα.

Να σημειωθεί ότι η εγγραφή στη μνήμη λάθος εντολών (με μη έγκυρη μορφή) υποτίθεται ότι δε μπορεί να συμβεί γιατί μελετάται μόνο ο αυτομεταβαλλόμενος κώδικας μέσα από μια γλώσσα που να διατηρεί την ακεραιότητα των εντολών στη μνήμη και να μην επιτρέπει την κατασκευή τέτοιων εντολών ή την καταστροφή τους.

Σε κάθε περίπτωση είναι εύλογο να επιδιώκεται ο περιορισμός τέτοιων συμπεριφορών που μπορούν να περιπλέξουν την ανάλυση της συμπεριφοράς ενός αντίστοιχου προγράμματος. Οι λύσεις που γενικά προτείνονται έχουν δύο μορφές:

- Να απαγορεύεται η τροποποίηση κώδικα που εξαρτάται από τον κώδικα που την προκαλεί. Σε αυτήν την περίπτωση δε μπορεί για παράδειγμα κάποια εντολή που βρίσκεται σε ένα βρόχο να αλλάξει τα όριά του, γιατί αυτό θα προκαλούσε κάποια αβεβαιότητα όσον αφορά ποιά εντολή του προγράμματος θα εκτελεστεί αμέσως μετά. Η οπτική αυτή είναι και η πιο διαδεδομένη στις υπάρχουσες υλοποιήσεις αυτομεταβαλλόμενου κώδικα και μία έκδοσή της είναι η περιορισμένη υποστήριξή του από μια γλώσσα με την εντολή eval() που προαναφέρθηκε. Η eval() εκτελείται σε ένα επίπεδο πιο εσωτερικό από αυτό στο οποίο κλήθηκε και άρα δε μπορεί να επηρεάσει το πρόγραμμα όπως προαναφέρθηκε. Αυτή η συμπεριφορά ακολουθείται για παράδειγμα από τη γλώσσα προγραμματισμού Perl όπως θα γίνει φανερό στη συνέχεια.

  Ως στρατηγική η παραπάνω είναι συντηρητική γιατί θεωρεί εκ των προτέρων λάθος μια μεγάλη κλάση ακίνδυνων προγραμμάτων που, ενω μεταβάλλουν κώδικα μέσα από αυτόν, δεν μπορούν να εμφανίσουν ασταθή χαρακτήρα. Στο παραπάνω παράδειγμα, αν η εντολή αντικαταστήσει τα όρια του βρόχου με τα ίδια, δεν υπάρξει δηλαδή πραγματικά αλλαγή, είναι προφανές ότι δεν υπάρχει κάποιο πρόβλημα στη εκτέλεση του προγράμματος.

- Να επιτρέπεται η τροποίηση τέτοιων ευαίσθητων τμημάτων κώδικα αλλά με τρόπο που να εγγυάται στον προγραμματιστή ότι η συμπεριφορά του κώδικά του είναι προβλέψιμη και

ασφαλής. Στην παρούσα διπλωματική εργασία έγινε προσπάθεια να ακολουθηθεί αυτός ο τροπος σκέψης, ώστε να μπορούν να γραφούν όσο το δυνατό περισσότερα προγράμματα στη γλώσσα AMK. Η αυτομεταβολή κώδικα επομένως επιτρέπεται αλλά διατηρεί την προβλεψιμότητα της θέσης του σημείου εκτέλεσης ενώ προϋποθέτει την επισήμανση μιας θέσης στη μνήμη ως θέσης υποψήφιας για μεταβολή.

2.2 Γνωστές εφαρμογές

2.2.1 Perl

Η γλώσσα Perl υποστηρίζει τη δημιουργία και την εκτέλεση κώδικα κατά την εκτέλεση ενός προγράμματος γραμμένου σε αυτή, μέσω της εντολής eval() [12]. Η eval() μπορεί να δέχεται ένα από τα ακόλουθα δυο είδη παραμέτρων:

Μια συμβολοακολουθία, που μπορεί να είναι γνωστή κατά το χρόνο της μεταγλώττισης, μπορεί όμως και όχι. Η περίπτωση αυτή ονομάζεται δυναμική αποτίμηση εκφράσεων (dynamic expression evaluation) και χρησιμοποιείται κυρίως όταν ο ζητούμενος κώδικας που πρόκειται να εκτελεστεί δίνεται από το χρήστη (περίπτωση κατασκευής μικρών διερμηνέων σε Perl) ή μπορεί να κατασκευαστεί με τον κατάλληλο χειρισμό συμβολοακολουθιών.

Ένα σώμα κώδικα, περίπτωση κατά την οποία είναι προφανές ότι ο κώδικας είναι γνωστός από το χρόνο μεταγλώττισης και ελέγχεται από το μεταγλωττιστή της Perl. Σε αυτήν την περίπτωση, το σύστημα εκτέλεσης της Perl είναι υπεύθυνο για τον εντοπισμό των σφαλμάτων χρόνου εκτέλεσης που μπορούν να προκύψουν από αυτό.

Αξίζει να αναφερθεί ότι ο κώδικας που εκτελείται δυναμικά δεν είναι ανεξάρτητος του προγράμματος αλλά εκτελείται στο ίδιο περιβάλλον(context) με αυτό. Αυτό σημαίνει ότι στον νέο κώδικα είναι διαθέσιμες όλες οι συναρτήσεις που έχουν οριστεί στο πρόγραμμα που τον καλεί, καθώς επίσης και οι μεταβλητές του. Έτσι ο νέος κώδικας μπορεί να επηρεάσει την εκτέλεσή του, να μεταβάλλει το περιεχόμενο των μεταβλητών του και να καλέσει κώδικα από αυτό. Επιπλέον, επειδή ο κώδικας αυτός θεωρείται στο επίπεδο της Perl ως σώμα (block), μπορεί να περιέχει τοπικές μεταβλητές, που δεν είναι προσβάσιμες από το υπόλοιπο πρόγραμμα και να επιστρέφει κάποια τιμή μετά την εκτέλεσή του.

Στην περίπτωση που ο κώδικας που δίνεται στην eval() είναι συντακτικά λανθασμένος, τότε ο μεταγλωττιστής της Perl που είναι υπεύθυνος για τη μετατροπή του σε γλώσσα χαμηλού επιπέδου πριν την εκτέλεσή του, εμφανίζει μήνυμα λάθους (Compilation error). Επίσης, αν ο κώδικας είναι συντακτικά σωστός αλλά κατά την εκτέλεσή του προκαλέσει κάποιο λάθος, το σύστημα εκτέλεσης θα εμφανίσει ένα μήνυμα λάθους χρόνου εκτέλεσης (Runtime error).

Τέλος, επειδή η Perl χρησιμοποιείται σε περιβάλλοντα στα οποία οι απαιτήσεις ασφάλειας είναι αυξημένες (system scripts, CGI scripts), η εντολή eval() πολλές φορές θεωρείται παράγοντας κινδύνου και είτε χρησιμοποιείται ο μηχανισμός ελέγχου μόλυνσης (taint-checking) της Perl που δεν επιτρέπει τη χρήση εξωγενούς πηγών (για παράδειγμα αρχείων) ως είσοδο στην eval(), είτε η εκτέλεσή της γίνεται με τη βοήθεια του περιβάλλοντος που παρέχει το άρθρωμα (module) Safe, κατά παρόμοιο τρόπο με το ασφαλές περιβάλλον που παρέχει ένας browser για την εκτέλεση δυναμικών σελίδων (Java, Javascript).

2.2.2 Προστασία προγραμμάτων

Μια από τις πιο διαδεδομένες χρήσεις του αυτομεταβαλλόμενου κώδικα είναι η προστασία του προγράμματος που τον χρησιμοποιεί. Η προστασία αυτή δεν είναι κάποια ιδιότητα που παρέχεται από οποιοδήποτε σύστημα προβλέπει αυτομεταβαλλόμενο κώδικα αλλά μόνο όταν αυτός υλοποιείται σε μία δυσνόητη γλώσσα (κατά προτίμηση γλώσσα μηχανής ή συμβολική γλώσσα). Σε αυτήν την περίπτωση, ένα πρόγραμμα προσπαθεί να προστατέψει τον κώδικά του από τρίτους που μπορούν να χρησιμοποιούν μεθόδους όπως η απομεταγλώττιση (decompile) και η εισαγωγή κώδικα (code injection) μέσω της δημιουργίας πολυπλοκού και δυσνόητου κώδικα.

Υπάρχουν έτοιμα εργαλεία που χρησιμοποιούνται για αυτόν το σκοπό, όπως οι obfuscators και οι συμπιεστές (packers), στα οποία δίνεται ένα πρόγραμμα και δημιουργείται το ισοδύναμό του σε πολύ πιο περίπλοκη όμως μορφή. Ο αυτομεταβαλλόμενος κώδικας χρησιμοποιείται συνήθως σε αυτό το στάδιο για την παραγωγή κώδικα που είναι δύσκολο να παρακολουθηθεί η ροή της εκτέλεσής του και το αρχείο του δεν αναλύεται εύκολα από τον άνθρωπο.

Οι δύο πιο γνωστές εφαρμογές των παραπάνω είναι:

- στα συστήματα προστασίας προγραμμάτων (copy protection) που κλειδώνουν ένα πρόγραμμα με τρόπο που μόνο αν κάποιος εκτελέσει μία συγκεκριμένη ακολουθία βημάτων, οριζόμενη από τον προγραμματιστή τους, να μπορεί να τα εκτελέσει

- στους ιούς, που για να αποφύγουν το εντοπισμό τους μεταβάλλουν τον εαυτό τους στη μνήμη, ώστε να διαφέρει ως προς συγκεκριμένες ακολουθίες (υπογραφές, signatures), για τις οποίες ερευνούν τα προγράμματα ανίχνευσης και αντιμετώπισης ιών (antivirus programs).

Προφανώς η παραπάνω χρήση δεν αποτελεί πραγματικό μέτρο ασφάλειας αλλά περισσότερο ένα δείγμα της αποκαλούμενης "ασφάλειας μέσω ιδιομορφίας" (security through obscurity). Εαν στη θέση της γλώσσας χαμηλού επιπέδου χρησιμοποιηθεί κάποια άλλη υψηλού επιπέδου με παρόμοιες δυνατότητες όσον αφορά τη αυτομεταβολή (για παράδειγμα η AMK της εργασίας), τότε το πρόγραμμα που θα προκύψει θα χαρακτηρίζεται από ένα σταθερό και μη προστατευμένο τμήμα: το σύστημα εκτέλεσης της γλώσσας αυτής. Το τμήμα αυτό είναι εύκολο να ανιχνευθεί και να τροποποιηθεί από τρίτους με σκοπό την ανάκτηση του κώδικα που υποτίθεται ότι προστατεύει. Άρα η όλη ασφάλεια της παραπάνω μεθόδου βασίζεται μάλλον σε ακραία χρήση της γλώσσας μηχανής και σε μεθόδους μετασχηματισμού προγραμμάτων που υλοποιούνται μόνο σε αυτή.

Επιπλέον είναι χαρακτηριστικό ότι η παραπάνω μέθοδος για τη συγγραφή προστατευμένων προγραμμάτων μπορεί να αποβεί προβληματική για αυτά, αφού δυσχεραίνει την ανάλυσή τους σε χαμηλό επίπεδο για λόγους απόδοσης (profiling) και αποκλείει ή κάνει ευάλωτες κάποιες μεθόδους που δουλεύουν σε χαμηλό επίπεδο για λόγους ασφάλειας. Παράδειγμα του τελευταίου είναι η χρήση τυχαία παραγόμενου κώδικα για την αποφυγή εισαγωγής ξένου κώδικα σε ένα πρόγραμμα [4].

2.2.3 Λειτουργικά συστήματα

Πρέπει να σημειωθεί ότι η πρακτική της συγγραφης αυτομεταβαλλόμενου κώδικα δεν είναι διαδεδομένη στα πιο πολλά λειτουργικά συστήματα. Στο μεγαλύτερο πλήθος αυτών(Linux, Windows, DOS) η παρουσία του συνήθως προϋποθέτει κάποια επαφή με την αντίστοιχη συμβολική γλώσσα της πλατφόρμας για την οποία προορίζεται και δεν υποστηρίζεται από το λειτουργικό σύστημα. Επιπλέον, λόγω της έλλειψης ασφάλειας που χαρακτηρίζει τέτοιες απόπειρες συγγραφής του, υπάρχουν περιβάλλοντα που απαγορεύουν την ύπαρξή του, όπως το λειτουργικό σύστημα OpenBSD που μέσω του μηχανισμού γράψε-ή-εκτέλεσε (W^X, write or execute) δεν επιτρέπει σε κάποια περιοχή της μνήμης να μπορεί να γράφεται από κάποιο πρόγραμμα και να περιέχει εκτελέσιμο κώδικα.

Εξαίρεση στα παραπάνω αποτελεί η ανάπτυξη του πυρήνα Synthesis [9] για το ομώνυμο λειτουργικό σύστημα τύπου UNIX. Αποτελείται από ένα συνθέτη κώδικα (code synthesizer) στο εσωτερικό του πυρήνα που αναλαμβάνει να δημιουργεί νέο κώδικα για εξειδικευμένες ρουτίνες του όταν συντρέχουν συγκεκριμένες συνθήκες. Με αυτόν τον τρόπο δημιουργούνται μικρά και γρήγορα τμήματα κώδικα, η χρήση των οποίων έχει αποδειχτεί ότι προσφέρει σημαντική αύξηση των επιδόσεων. Η δημιουργία των τμημάτων αυτών υποκαθιστά τον περίπλοκο εξωτερικό κώδικα που καλείται σε άλλα λειτουργικά συστήματα όταν απαιτείται να γίνει μια πράξη που δεν καλύπτεται από τις ήδη υπάρχουσες κλήσεις πυρήνα. Ειδικά όταν η ανάγκη για τέτοιες κλήσεις είναι συχνή, ο νέος κώδικας που έχει δημιουργηθεί από το συνθετητή προσφέρει σημαντικά

στην επίδοση του όλου συστήματος. Η θεωρητική μελέτη του πυρήνα Synthesis και η χρήση του από τον προγραμματιστή γίνεται σύμφωνα με το υπολογιστικό μοντέλο της συνθετικής μηχανής (synthetic machine) που προσομοιώνει ένα εικονικό πολυεπεξεργαστικό σύστημα με υποστήριξη ταυτοχρονισμού (concurrent programming). Το σύστημα Synthesis θεωρείται μια από τις πιο πλήρεις προσπάθειες μέχρι στιγμής να εφαρμοστεί η ιδέα του αυτομεταβαλλόμενου κώδικα σε επίπεδο λειτουργικού συστήματος με αποδοτικό τρόπο.

2.2.4 Βελτιστοποίηση κώδικα σε ML

Ο αυτομεταβαλλόμενος κώδικας έχει χρησιμοποιηθεί και στην περίπτωση κατασκευής ενός αποδοτικού μεταγλωττιστή για ένα υποσύνολο της γλώσσας προγραμματισμού ML. Έχει παρατηρηθεί ότι το αρχείο που προκύπτει από την απλή μεταγλώττιση ενός αρχείου πηγαίου κώδικα σε ML δεν είναι το βέλτιστο – υπάρχουν περιθώρια για βελτιώσεις, που όμως μπορούν να αξιοποιηθούν μόνο τη στιγμή της εκτέλεσής του.

Η κατασκευή του μεταγλωττιστή FABIUS [5] αντιμετωπίζει αυτό το πρόβλημα με την κατασκευή ενος προγράμματος υπεύθυνου για τη δημιουργία του τελικού προγράμματος. Ο κώδικας σε ML μεταγλωττίζεται σε κώδικα μηχανής για την εκάστοτε πλατφόρμα υλικού (native code) κώδικα αλλά αυτό που προκύπτει δεν είναι το τελικό αλλά ένα ενδιάμεσο πρόγραμμα που όταν εκτελεστεί θα είναι υπεύθυνο για τη δημιουργία του τελικού κώδικα, έχοντας υπολογίσει τις συνθήκες, κάτω από τις οποίες θα εκτελεστεί. Με αυτόν τον τρόπο προκύπτει το πρόγραμμα σε κώδικα μηχανής με τις επιπλέον βελτιώσεις που θα ήταν αδύνατο να προβλεφθούν κατά το χρόνο της αρχικής μεταγλώττισης. Χαρακτηριστικές υλοποιήσεις προγραμμάτων σε ML που μεταγλωττίζονται με αυτό το σύστημα δείχνουν ότι είναι αποδοτικότερο από έναν απλό μεταγλωττιστή για την αντίστοιχη έκδοση της ML και ορισμένες φορές αποδοτικότερο από το αντίστοιχο αρχείο γραμμένο σε C (παράγοντας αποδοτικότητας 4x).

2.2.5 Γενετικοί αλγόριθμοι

Οι γενετικοί αλγόριθμοι βασίζουν τη λειτουργία τους σε τμήματα κώδικα που μεταβάλλονται με σκοπό την τελική παραγωγή του επιθυμητού κώδικα από μια κατάλληλη αρχική μορφή. Κατά την εκτέλεσή τους δημιουργούν διάφορα ενδιάμεσα στάδια αυτού του κώδικα με επιλογή από ένα αρχικό σύνολο δεδομένων (pool) και κάνοντας χρήση πιθανοτικών μεθόδων διαδοχικών μεταλλάξεων αυτών. Επίσης εξετάζεται η χρήση ντετερμινιστικών μεθόδων για τη δημιουργία του μεταλλαγμένου κώδικα. Στη συνέχεια ο κώδικας αυτός που έχει παραχθεί ελέγχεται από κάποιο κριτήριο που βαθμολογεί την απόδοσή του, το κατά πόσο δηλαδή πλησιάζει το ζητούμενο πρόγραμμα.

Οι γενετικοί αλγόριθμοι είναι αρκετά διαδομένοι σήμερα και αποτελούν πεδίο μελέτης με έντονη ερευνητική δραστηριότητα. Χρησιμοποιούνται σε περιπτώσεις που οι συνηθισμένοι αλγόριθμοι συγκλίνουν πολύ πιο αργά στην κατάλληλη λύση ή δε μπορούν να την προσεγγίσουν εύκολα. Σε αυτό το σημείο η επιθετική στρατηγική διερεύνησης ενός ευρύτερου πεδίου τιμών λόγω της πιθανοτικής διάσχισής του και η χρήση των κατάλληλων κριτηρίων μπορούν να απαλείψουν γρήγορα ολόκληρες κλάσεις περιττών δεδομένων [2].

Η μετάλλαξη πουπροαναφέρθηκε καθώς και η εφαρμογή του κριτηρίου καταλληλότητας των παραγόμενων τμημάτων μπορεί να γίνει με δυο τρόπους:

- από κάποιο εξωτερικό σύστημα που επιβλέπει αυτή τη διαδικασία, περίπτωση στην οποία πρόκειται για κάποιο γενετικό σύστημα διαχείρισης δεδομένων
- από τον ίδιο τον κώδικα που αυτομεταβάλλεται με τη βοήθεια της γλώσσας στην οποία είναι γραμμένος, αν αυτή το υποστηρίζει

Ενδιαφέρουσα είναι κυρίως η δεύτερη προσέγγιση, για τους σκοπούς της διπλωματικής αυτής εργασίας, γιατί αφορά ένα σημαντικό πεδίο της σημερινής έρευνας πάνω στο γενετικό

προγραμματισμό. Οι γενετικοί αλγόριθμοι, σε συνδυασμό με τις κατάλληλες δομές δεδομένων και τη γλώσσα ή σύστημα στο οποίο αναπτύσσονται, μπορούν να προσομοιώσουν φυσιολογικά συστήματα, να βοηθήσουν στην κατανόησή τους και να χρησιμοποιηθούν αντί αυτών (για παράδειγμα στην Τεχνητή Νοημοσύνη). Τέτοια προγράμμα, επηρεασμένα από το βιολογικό φαινόμενο της εξέλιξης, ονομάζονται εξελικτικά προγράμματα (evolution programs [7]). Αξίζει να τονιστεί ότι κάθε πρόγραμμα σε αυτομεταβαλλόμενο κώδικα με την έννοια που του έχει αποδοθεί από αυτήν την εργασία μπορεί να αναχθεί σε κάποιο εξελικτικό πρόγραμμα γιατί η γλώσσα ΑΜΚ που θα αναπτυχθεί στη συνέχει διευκολύνει την έκφραση τέτοιων κριτηρίων και παρεμβάσεων σε ένα πρόγραμμα τη στιγμή που εκτελείται.

2.2.6 Pure Data

Η χρήση του αυτομεταβαλλόμενου κώδικα δεν περιορίζεται μόνο σε γλώσσες στις οποίες το πρόγραμμα δίνεται στο μεταγλωττιστή ή στο διερμηνέα υπό μορφή κειμένου. Χαρακτηριστικό παράδειγμα αποτελεί η χρήση του στη γλώσσα οπτικού προγραμματισμού Pure Data, στην οποία τα προγράμματα παριστάνονται με τη μορφή κατευθυνόμενων γράφων αντικειμένων(patches), με τις μεταξύ τους συνδεσεις να περιγράφουν τη ροή των μηνυμάτων μεταξύ αυτών. Αξίζει να σημειωθεί ότι η γλώσσα αυτή είναι πλήρης κατά Turing και άρα οποιοδήποτε άλλο πρόγραμμα μπορεί να γραφεί σε οπτική μορφή όπως αυτή που μόλις περιγράφηκε.

Σε αυτή τη γλώσσα η δυνατότητα αυτομεταβαλλόμενου κώδικα υποστηρίζεται από δύο ειδικά αντικείμενα (dyn, dyn~ [3]), τα οποία δέχονται μηνύματα και αναλαμβάνουν να δημιουργήσουν ή να καταστρέψουν άλλα αντικείμενα του προγράμματος, καθώς και να δημιουργήσουν ή να διαγράψουν τις συνδέσεις μεταξύ τους. Με αυτόν τον τρόπο κάθε πρόγραμμα είναι ικανό να δημιουργεί άλλα προγράμματα ή να μεταβάλλει τον εαυτό του με τις ίδιες δυνατότητες που έχει και ο προγραμματιστής όταν το δημιουργεί.

2.2.7 Objective C

Η γλώσσα προγραμματισμού Objective C αποτελεί μια αντικειμενοστρεφή έκδοση της C σε μια προσπάθεια να προσεγγιστούν μέσα από αυτή χαρακτηριστικά και ιδέες της SmallTalk. Στην πιο ευρέως διαθέσιμη έκδοσή της, όπως υλοποιείται από το πρότυπο OPENSTEP μέσω του μεταγλωττιστή gcc, υποστηρίζει τη δυναμική φόρτωση μεταγλωττισμένου κώδικα από αρχεία (bundles), που να μπορεί να αντικαθιστά αντίστοιχο κώδικα σε κάποιο πρόγραμμα τη στιγμή που εκτελείται. Ένα πρόγραμμα, μπορεί καθώς εκτελείται δηλαδή να αντικαταστήσει μεθόδους ή πεδία κάποιων αντικειμένων του, αφού του δίνεται η ικανότητα να τροποποιεί τα αντικείμενα τη στιγμή που εκτελούνται, ορίζοντάς τα ως μέλη μιας συγκεκριμένης διαπροσωπείας (interface).

Η παραπάνω πρακτική βέβαια δεν ικανοποιεί τον ορισμό του αυτομεταβαλλόμενου κώδικα που έχει δοθεί επειδή πρόκειται για ήδη μεταγλωττισμένο κώδικα, που δεν έχει δημιουργηθεί στο επίπεδο της γλώσσας. Είναι όμως άξια αναφοράς δύο χαρακτηριστικά των παραπάνω δυνατοτήτων της Objective C:

- Ένα πρόγραμμα είναι ικανό να μεταβάλλει τον εαυτό του, έστω και με προ-μεταγλωττισμένα τμήματα κώδικα. Η δυνατότητα αυτή δεν πρέπει να υποτιμηθεί γιατί με μελετημένη χρήση αρκετών τέτοιων τμημάτων και με προσεκτικό σχεδιασμό της ιεραρχίας των κλάσεων που χησιμοποιούνται, ένα πρόγραμμα μπορεί σε επίπεδο αντικειμένων να εμφανίσει συμπεριφορά παρόμοια με αυτή ενός προγράμματος που υποστηρίζει αυτομεταβαλλόμενο κώδικα. Σε επίπεδο συστήματος εκτέλεσης ένα τέτοιο πρόγραμμα θα εμφανίζεται να μπορεί να δημιουργεί και να διαγράφει αντικείμενα και άρα έμμεσα να έχει δυνατότητες τροποποίησης του εαυτού του.

- Η χρήση δυναμικού κώδικα που φορτώνεται στην εκτέλεση όχι απλώς ως ένα άρθρωμα όπως οι βιβλιοθήκες αλλά ως τμήμα κώδικα που αλληλεπιδρά με τον ήδη υπάρχοντα βρί-

σκει χρήση σε συστήματα που χρησιμοποιούνται από αρκετούς χρήστες. Τέτοια συστήματα είναι το GNUSTEP [17, 13] και το MacOSX, στο οποίο κυκλοφορούν και εργαλεία [13] για την παρέμβαση σε οποιοδήποτε πρόγραμμα τη στιγμή που εκτελείται, με σκοπό την αλλαγή της συμπεριφοράς του με τρόπο ασφαλή και συνεπή ως προς τη λογισμική αρχιτεκτονική του λειτουργικού συστήματος.

2.3 Χρησιμότητα αυτομεταβαλλόμενου κώδικα: εξοικονόμηση πόρων

Μία από τις σημαντικότερες δυνατότητες του AMK και μία από τις μέχρι σχετικά πρόσφατα χρήσεις του ήταν η εξοικονόμηση πόρων σε αδύναμα συστήματα. Ένα πρόγραμμα που μεταβάλλει τον εαυτό του έχει τη δυνατότητα (σε ορισμένες εφαρμογές) να καταναλώσει μικρότερη ποσότητα μνήμης (RAM, χώρος σε κάποιο αποθηκευτικό μέσο, εσωτερική μνήμη επεξεργαστή) ή λιγότερες εντολές της Κεντρικής Μονάδας Επεξεργασίας (ΚΜΕ) από ένα αντίστοιχο πρόγραμμα που δε χρησιμοποιεί τέτοιες πρακτικές. Για παράδειγμα, έστω το στοιχειώδες πρόγραμμα που μπορεί να γραφεί με τις εξής δυο ισοδύναμες μορφές:

```
(μη-αυτομεταβαλλόμενος, παράδειγμα 1)

for i=1 to 9 { κώδικας... }

for i=1 to 4 { κώδικας... }

(μη-αυτομεταβαλλόμενο, παράδειγμα 2)

f(x) := for i=1 to x { κώδικας... }

f(9)

f(4)
```

Στα παραπάνω θεωρείται ότι ο κώδικας σημειωμένος με *κώδικας...* είναι ο ίδιος.

Στην πρώτη περίπτωση το πρόγραμμα θα εκτελεστεί γρήγορα, αλλά θα δαπανηθεί αρκετή μνήμη για αυτό λόγω της διπλής ύπαρξης του σώματος εντολών *κώδικας...*, που αν είναι μεγάλο, μπορεί να οδηγήσει σε σημαντική αύξηση του μεγέθους του τελικού κώδικα.

Στην δεύτερη περίπτωση ο κώδικας δίνεται μόνο μια φορά και αποφεύγεται η σπατάλη μνήμης αλλά προκύπτει η επιβάρυνση (overhead) από τις κλήσεις της συνάρτησης f. Κάθε κλήση πρέπει να αποθηκεύσει και να επαναφέρει το περιβάλλον που την κάλεσε, σε κάθε κλήση και τερματισμό της αντίστοιχα.

Έστω τώρα το παρακάτω τμήμα κώδικα:

```
(αυτομεταβαλλόμενος, παράδειγμα 3)

F: for i=1 to 9 { κώδικας... }

store(F, for i=1 to 4)

run(F)
```

Εδώ υποτίθεται ότι υπάρχει κάποιος μηχανισμός απόδοσης ονομάτων σε θέσεις μνήμης κατειλημμένες από εντολές με την ανάθεση ετικετών σε αυτές, όπως η F. Επίσης η ψευδογλώσσα

στην οποία είναι γραμμένη υποστηρίζει μια εντολή store που δέχεται μια ετικέττα και τοποθετεί το δεύτερο όρισμά της στη συνθήκη του περιεχομένου της ετικέττας.

Σε αυτήν την περίπτωση υπάρχουν και οι δυο βελτιώσεις που ήταν χαρακτηριστικό των παραπάνω παραδειγμάτων. Στην πρώτη κλήση υπάρχει το κέρδος ότι δε χρειάζεται κάποια ανάθεση (όπως στο παράδειγμα 2) ή επανάληψη τμήματος κώδικα (όπως στο παράδειγμα 1). Όμως ο κέρδος, ειδικά για το 2ο (χρόνος που σπαταλάται από την ΚΜΕ), είναι οριακό. Το 1ο προσφέρει καλύτερες δυνατότητες αλλά δεν προκύπτει στην πραγματικότητα, εκτός και αν μεσολάβεί μεταγλωττιστής με δυνατότητα ξεδιπλώματος μικρών βρόχων (loop-unrolling) στο 2ο πρόβλημα (οπότε τα 1 και 2 είναι ισοδύναμα).

Στα σημερινά συστήματα η δυνατότητα βελτίωσης του κώδικα (optimization) σε τέτοιο επίπεδο με τη χρήση αυτομεταβαλλόμενου κώδικα είναι μικρή και γίνεται κυρίως από μεταγλωττιστές, άρα σε επίπεδο συμβολικής γλώσσας, αφού δεν έχει προκύψει η γενικευμένη ανάγκη υλοποίησης ενός κατάλληλου συστήματος που να τον υποστηρίζει και να αποτρέπει το χρήστη από λάθη (αφού δεν υπάρχει χρήστης, μόνο μεταγλωττιστής). Βέβαια, όπως αναφέρθηκε και σε προηγούμενο κεφάλαιο, η σωστή υλοποίηση ενός συστήματος για ασφαλή συγγραφή αυτομεταβαλλόμενου κώδικα σε αυτό, προϋποθέτει την ύπαρξη ενός συστήματος χρόνου εκτέλεσης που να μεσολαβεί μεταξύ του προγράμματος και του συστήματος επιτρέποντας το χειρισμό της αναπαράστασης των εντολών στη μνήμη.

Κεφάλαιο 3

Η γλώσσα AMK

3.1 Το συντακτικό της γλώσσας

Το συντακτικό της γλώσσας σε EBNF μορφή φαίνεται στον πίνακα 3.1.

Στη συνέχεια περιγράφεται η σύνταξη και η λειτουργία κάθε εντολής της γλώσσας. Οι τυπογραφικές συμβάσεις που ακολουθούνται είναι οι εξής:

- κανονική γραφή
  Συμβολίζονται τα ονόματα των μεταβλητών και των ετικεττών.
- *έντονη γραφή*
  Συμβολίζονται τα δεσμευμένα ονόματα των εντολών και συμβόλων της γλώσσας.

3.1.1 Εντολή fetch

Σύνταξη:

fetch (var label)

Λειτουργία:

Η εντολή fetch διαβάζει τα περιεχόμενα της ετικέττας με όνομα label και τα τοποθετεί στη μεταβλητή με όνομα var. Αν η μεταβλητή var δεν υπάρχει, δημιουργείται εκείνη τη στιγμή.

3.1.2 Εντολή store

Σύνταξη:

store (label var)

Λειτουργία:

Η εντολή store αντικαθιστά το περιεχόμενο της ετικέττας με όνομα label με το περιεχόμενο της μεταβλητής var. Αν η ετικέττα label δε βρίσκεται στο μονοπάτι που συνδέει την εντολή store με τη ρίζα του δέντρου του προγράμματος, η εκτέλεση συνεχίζει με την επόμενη εντολή που ακολουθεί την εντολή store. Αν όμως αυτό δεν ισχύει, δηλαδή αν η ετικέττα label είναι πρόγονος της εντολής store, τότε το ζήτημα της εύρεσης της επόμενης εντολής που θα εκτελεστεί είναι περίπλοκο γιατί μπορεί να έχει αλλάξει το ίδιο το σώμα του κώδικα που την περιέχει. Σε αυτήν την περίπτωση, κάποιος πολύπλοκος αλγόριθμος θα μπορούσε να αναλύσει τον νέο κώδικα που προκύπτει και να υπολογίσει την επόμενη εντολή προς εκτέλεση, όμως η συμπεριφορά αυτή δεν είναι εύκολο να γίνει κατανοητή από τον προγραμματιστή και να προβλεφθεί με ασφάλεια το αποτέλεσμα μιας εντολής store σε οποιοδήποτε περιβάλλον προγράμματος. Έτσι, σε αυτήν την περίπτωση, η επόμενη εντολή προς εκτέλεση επιλέγεται να είναι η πρώτη εντολή

Πίνακας 3.1: Συντακτικό της γλώσσας AMK.

⟨program⟩ ::= “**label**” ⟨ID⟩ “{” ⟨stmt-body⟩ “}”
| “**test**”

⟨char⟩ ::= “**A**” | . . . | “**Z**” | “**a**” | . . . | “**z**”

⟨digit⟩ ::= “**0**” | . . . | “**9**”

⟨punct⟩ ::= “.” | “,” | “ ” | “=” | “+” | “−” | “(” | “)” | “{” | “}”

⟨ID⟩ ::= ⟨char⟩$^+$

⟨NUM⟩ ::= ⟨digit⟩$^+$ [“.” ⟨digit⟩$^+$]

⟨label-block⟩ ::= “**label**” ⟨ID⟩ “{” ⟨stmt-body⟩ “}”
| “**vlabel**” ⟨ID⟩ “{” “}”

⟨stmt-body⟩ ::= ϵ | ⟨stmt-body⟩ ⟨stmt⟩

⟨stmt⟩ ::= “**fetch**” “(” ⟨ID⟩ ⟨ID⟩ “)”
| “**store**” “(” ⟨ID⟩ ⟨ID⟩ “)”
| “**store2**” “(” ⟨ID⟩ ⟨ID⟩ “)”
| “**readv**” “(” ⟨ID⟩ ⟨ID⟩ “)”
| “**writev**” “(” ⟨ID⟩ ⟨ID⟩ “)”
| “**read**” “(” ⟨ID⟩ “)”
| “**readcode**” “(” ⟨ID⟩ “)”
| “**write**” “(” ⟨ID⟩ “)”
| “**writeln**”
| “**writesp**”
| “**if**” ⟨ID⟩ “{” ⟨stmt-body⟩ “}” [“**else**” “{” ⟨stmt-body⟩ “}”]
| “+” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “−” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “∗” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “/” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “>” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “<” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “:=” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “**concat**” ⟨ID⟩ ⟨ID⟩ ⟨ID⟩
| “**head**” ⟨ID⟩ ⟨ID⟩
| “**cons**” ⟨ID⟩ ⟨ID⟩
| ⟨NUM⟩
| ⟨label-block⟩ “**for**” ⟨ID⟩ “{” ⟨stmt-body⟩ “}”
| “**while**” ⟨ID⟩ “{” ⟨stmt-body⟩ “}”
| “**set**” ⟨ID⟩ “{” ⟨stmt-body⟩ “}”
| “**rnd**” ⟨ID⟩
| “"”([*punct*] [*char*] [*digit*])$^+$ “"”
| “[”([*punct*] [*char*] [*digit*])$^+$ “]”

της ετικέττας label, γίνεται δηλαδή επανεκκίνηση του τμήματος του προγράμματος που είναι ύποπτο για εσφαλμένη συμπεριφορά.

3.1.3 Εντολή store2

Σύνταξη:

store2 (label var)

Λειτουργία:

Η εντολή store2 χρησιμοποιείται για τη δημιουργία δομών εντολών στη μνήμη. Αντικαθιστά την εντολή της ετικέττας με όνομα label με τα περιεχόμενα της μεταβλητής var. Δεν παρουσιάζει τη συμπεριφορά της εντολής store όταν αντικαθιστά ετικέττα που είναι πρόγονός της. Αυτό δεν οφείλεται τόσο σε κάποιο τεχνικό πρόβλημα στην υλοποίησή της, όσο στην παρότρυνση προς τον προγραμματιστή να μην επιδιώκει την κατασκευή σύνθετων δομών στον ίδιο τον κώδικα που τρέχει, διαδικασία που πολύ εύκολα μπορεί να οδηγήσει σε δυσνόητα προγράμματα και λάθος συμπεριφορές τους. Αν αυτό είναι απαραίτητο, τότε η επιθυμητή δυναμική δομή μπορεί να κατασκευαστεί πρώτα κάπου αλλού, εκτός του μονοπατιού που συνδέει την εντολή store2 με τη ρίζα του προγράμματος και στη συνέχεια με τη βοήθεια εντολών fetch και store, το περιεχόμενό της να μεταφερθεί όπου χρειάζεται.

Η εντολή store2 χρησιμοποιείται για την κατασκευή προγραμμάτων στη μνήμη μέσω του μηχανισμού των δυναμικών εντολών. Πριν τοποθετήσει τον κώδικα στην ετικέττα με όνομα label, ο κώδικας φροντίζει να αντικαταστήσει κάθε εντολή vlabel σε αυτόν με την ισοδύναμη ετικέττα (εντολή label). Έτσι ο παρακάτω κώδικας:

```
set x { 6 8 9 }
vlabel one { }
for f {
        vlabel two { }
        write (x)
}
```

μετασχηματίζεται ως εξής πριν την τοποθέτησή του:

```
set x { 6 8 9 }
label one { }
for f {
        label two { }
        write (x)
}
```

Η συμπεριφορά αυτή χρησιμοποιείται όταν ένα τμήμα κώδικα αντικαθιστά μία ετικέττα με όνομα x και το ίδιο περιέχει μια vlabel x { }. Με αυτόν τον τρόπο, χρησιμοποιώντας μόνο το μηχανισμό των ετικεττών, είναι δυνατό να κατασκευαστούν αναδρομικά δομές στη μνήμη, σε οποιοδήποτε σώμα εντολών.

3.1.4 Εντολή '+'

Σύνταξη:

+ var1 var2 var3

Λειτουργία:

Αν οι δύο μεταβλητές var2 και var3 περιέχουν ακριβώς έναν αριθμό η κάθε μία, η μεταβλητή var1 τίθεται ίση με το άθροισμά τους. Σε άλλη περίπτωση ο διερμηνέας επιστρέφει μήνυμα λάθους. Αν η μεταβλητή var1 δεν υπάρχει, δημιουργείται εκείνη τη στιγμή.

3.1.5 Εντολή '-'

Σύνταξη:

- var1 var2 var3

Λειτουργία:

Αν οι δύο μεταβλητές var2 και var3 περιέχουν ακριβώς έναν αριθμό η κάθε μία, η μεταβλητή var1 τίθεται ίση με τη διαφορά τους. Σε άλλη περίπτωση ο διερμηνέας επιστρέφει μήνυμα λάθους. Αν η μεταβλητή var1 δεν υπάρχει, δημιουργείται εκείνη τη στιγμή.

3.1.6 Εντολή '*'

Σύνταξη:

* var1 var2 var3

Λειτουργία:

Αν οι δύο μεταβλητές var2 και var3 περιέχουν ακριβώς έναν αριθμό η κάθε μία, η μεταβλητή var1 τίθεται ίση με το γινόμενό τους. Σε άλλη περίπτωση ο διερμηνέας επιστρέφει μήνυμα λάθους. Αν η μεταβλητή var1 δεν υπάρχει, δημιουργείται εκείνη τη στιγμή.

3.1.7 Εντολή '/'

Σύνταξη:

/ var1 var2 var3

Λειτουργία:

Αν οι δύο μεταβλητές var2 και var3 περιέχουν ακριβώς έναν αριθμό η κάθε μία, η μεταβλητή var1 τίθεται ίση με το πηλίκο τους. Αν η μεταβλητή var3 περιέχει τον αριθμό 0 τότε ο διερμηνέας διακόπει την εκτέλεση του προγράμματος με μήνυμα λάθους ή αν κάποια από τις μεταβλητές var2 και var3 δεν περιέχει 1 αριθμό, ο διερμηνέας επιστρέφει μήνυμα λάθους. Αν η μεταβλητή var1 δεν υπάρχει, δημιουργείται εκείνη τη στιγμή.

3.1.8 Εντολή '>'

Σύνταξη:

> var1 var2 var3

Λειτουργία:

Αν οι δύο μεταβλητές var2 και var3 περιέχουν ακριβώς έναν αριθμό η κάθε μία, η μεταβλητή

var1 τίθεται ίση με τη 1 αν το περιεχόμενο της πρώτης είναι μεγαλύτερο από αυτό της δεύτερης. Αν το περιεχόμενο της πρώτης είναι μικρότερο από αυτό της δεύτερης, τοποθετείται η τιμή 0 στη μεταβλητή var. Σε άλλη περίπτωση ο διερμηνέας επιστρέφει μήνυμα λάθους. Αν η μεταβλητή var1 δεν υπάρχει, δημιουργείται εκείνη τη στιγμή.

3.1.9 Εντολή '<'

Σύνταξη:

< var1 var2 var3

Λειτουργία:

Αν οι δύο μεταβλητές var2 και var3 περιέχουν ακριβώς έναν αριθμό η κάθε μία, η μεταβλητή var1 τίθεται ίση με τη 0 αν το περιεχόμενο της πρώτης είναι μεγαλύτερο από αυτό της δεύτερης. Αν το περιεχόμενο της πρώτης είναι μικρότερο από αυτό της δεύτερης, τοποθετείται η τιμή 1 στη μεταβλητή var. Σε άλλη περίπτωση ο διερμηνέας επιστρέφει μήνυμα λάθους. Αν η μεταβλητή var1 δεν υπάρχει, δημιουργείται εκείνη τη στιγμή.

3.1.10 Εντολή label

Σύνταξη:

label name { εντολές... }

Λειτουργία:

Η εντολή label ορίζει ένα σημείο στο πρόγραμμα, τα περιεχόμενα του οποίου μπορούν να τροποποιηθούν από μια εντολή store ή store2. Αξίζει να σημειωθεί ότι αυτός είναι ο μοναδικός τρόπος να οριστεί ένα μεταβλητό σημείο σε ένα πρόγραμμα.

3.1.11 Εντολή vlabel

Σύνταξη:

vlabel name { }

Λειτουργία:

Η εντολή vlabel χρησιμοποιείται για να συμβολίσει το σημείο το οποίο πρόκειται να αντικατασταθεί από μια πραγματική εντολή ετικέττας όταν το σώμα κώδικα στο οποίο βρίσκεται δοθεί σε μια εντολή store2. Επειδή ο ρόλος της είναι απλοκλειστικά ως ένα σύμβολο, το σώμα της είναι κενό - ακόμη και αν δοθούν εντολές της γλώσσας AMK, αυτές θα αγνοηθούν.

3.1.12 Εντολή write

Σύνταξη:

write(var)

Λειτουργία:

Αν η μεταβλητή var περιέχει έναν αριθμό ή μια συμβολοσειρά, τότε αυτό τυπώνεται στην

οθόνη. Σε διαφορετική περίπτωση η μεταβλητή θεωρείται ότι περιέχει κώδικα και τυπώνεται στην οθόνη το δέντρο που αντιστοιχεί σε αυτόν.

3.1.13 Εντολή writeln

Σύνταξη:

writeln

Λειτουργία:

Στέλνει ένα χαρακτήρα αλλαγής γραμμής στην οθόνη.

3.1.14 Εντολή writesp

Σύνταξη:

writesp

Λειτουργία:

Εκτυπώνει ένα χαρακτήρα κενού στην οθόνη.

3.1.15 Εντολή if

Σύνταξη:

if var { ... }

if var { ... } *else* { }

Λειτουργία:

Αν το περιεχόμενο της μεταβλητής var είναι αριθμός, τότε αν είναι ίσο με το μηδέν, τότε εκτελείται το πρώτο σώμα των εντολών ενώ αν δεν είναι μηδέν, εκτελείται το δεύτερο σώμα εντολών (αν δίνεται). Σε άλλη περίπτωση, η εντολή if δεν κάνει τίποτα και ακολυθεί η εκτέλεση της επόμενης εντολής μετά από αυτή.

3.1.16 Εντολή ':='

Σύνταξη:

:= var1 var2

Λειτουργία:

Αντιγράφει το περιεχόμενο της μεταβλητής var2 στη μεταβλητή var1.

3.1.17 Εντολή set

Σύνταξη:

set var { ... }

Λειτουργία:

Αντικαθιστά το περιεχόμενο της μεταβλητής var με το σώμα των εντολών που ακολουθούν. Οι εντολές δεν εκτελούνται αλλά το δέντρο του κώδικα, που αυτές αποτελούν και είχε δημιουργηθεί κατά τη διαδικασία της συντακτικής ανάλυσης, τοποθετείται στη μεταβλητή var. Με αυτόν τον τρόπο κατασκευάζεται ένα τμήμα κώδικα, ο λεκτικός και συντακτικός έλεγχος του οποίου γίνεται πριν εκτελεστεί το πρόγραμμα και άρα έχει στατικό χαρακτήρα.

3.1.18 Εντολή read

Σύνταξη:

read (var)

Λειτουργία:

Διαβάζει έναν αριθμό ή μία συμβολοσειρά από την είσοδο (πληκτρολόγιο) και ανάλογα με το είδος της δημιουργεί τα αντίστοιχα δεδομένα που τα τοποθετεί στη μεταβλητή var.

3.1.19 Εντολή readcode

Σύνταξη:

readcode (var)

Λειτουργία:

Διαβάζει ένα ρεύμα χαρακτήρων από την είσοδο και το αναλύει λεκτικά και σημασιολογικά ώστε να δημιουργήσει το αντίστοιχο πρόγραμμα σε AMK. Η είσοδος δηλαδή πρέπει να αποτελεί ένα έγκυρο πρόγραμμα της γλώσσας αλλιώς ο διερμηνέας εκτυπώνει μήνυμα λάθους. Στη συνέχεια, το πρόγραμμα αυτό τοποθετείται στη μεταβλητή var σαν να είχε τοποθετηθεί σε αυτή στατικά από το ίδιο το πρόγραμμα μέσω κάποιας εντολής set.

3.1.20 Εντολή rnd

Σύνταξη:

rnd (var)

Λειτουργία:

Τοποθετεί στη μεταβλητή var την τιμή 0 ή την τιμή 1 με ίση πιθανότητα για την κάθε μία από αυτές.

3.1.21 Εντολή while

Σύνταξη:

while var { ... }

Λειτουργία:

Εαν η μεταβλητή var έχει τιμή διάφορη του 0 τότε εκτελείται το σώμα της εντολής while. Κάθε φορά που ολοκληρώνεται η εκτέλεση του σώματος των εντολών, ο έλεγχος αυτός γίνεται εκ νέου. Αν επιτύχει ακολουθεί η ίδια διαδικασία ενώ αν αποτύχει, το πρόγραμμα συνεχίζει με την εκτέλεση της επόμενης εντολής της while.

3.1.22 Εντολή for

Σύνταξη:

for var { ... }

Λειτουργία:

Η εντολή for εκτελεί τόσες φορές το σώμα της, όσος είναι ο αριθμός που βρίσκεται στη μεταβλητή var, στρογγυλοποιημένος προς τον πλησιέστερο ακέραιο. Από τη στιγμή που εκτελείται για πρώτη φορά ο βρόχος, ο αριθμός των επαναλήψεών του είναι συγκεκριμένος και δεν είναι δυνατό να αλλάξει, ακόμα και αν μεταβληθεί το περιεχόμενο της μεταβλητής var.

Αν η μεταβλητή var δεν περιέχει έναν αριθμό, τότε η εντολή αγνοεί το σώμα των εντολών της.

3.1.23 Εντολή concat

Σύνταξη:

concat var1 var2 var3

Λειτουργία:

Συνενώνει τις λίστες των εντολών που που υπάρχουν στις μεταβλητές var2 και var3 και τοποθετεί το αποτέλεσμα της πράξης αυτής στη μεταβλητή var1. Η εντολή αυτή δεν είναι απαραίτητη για τη γλώσσα: μπορεί να προσομοιωθεί με την τοποθέτηση μιας εντολής vlabel στο τέλος του σώματος της πρώτης μεταβλητής, την τοποθέτησή της μέσω μιας εντολής store2 σε μια ετικέττα και την τοποθέτηση της δεύτερης μεταβλητής με μια εντολή store στο τέλος της. Στη συνέχεια ο κώδικας που έχει προκύψει στην ετικέττα αυτή μπορεί να διαβαστεί πίσω με τη βοήθεια μιας εντολής fetch. Η διαδικασία αυτή όμως είναι αρκετά επίπονη και αφορά μια συχνή διαδικασία κατά τον προγραμματισμό, για αυτόν το λόγο και έγινε η επιλογή να προστεθεί η εντολή concat στη γλώσσα.

3.1.24 Εντολή cons

Σύνταξη:

cons var1 var2

Λειτουργία:

Η εντολή cons τοποθετεί στη μεταβλητή var1 τη λίστα των εντολών που βρίσκονται στη μεταβλητή var2 χωρίς το πρώτο της στοιχείο. Αν η μεταβλητή var2 είναι κενή ή περιέχει μόνο ένα στοιχείο, τότε στη μεταβλητή var1 τοποθετείται η κενή λίστα.

3.1.25 Εντολή head

Σύνταξη:

head var1 var2

Λειτουργία:

Η εντολή head τοποθετεί στη μεταβλητή var1 το πρώτο στοιχείο της λίστας των εντολών που βρίσκονται στη μεταβλητή var2. Αν η μεταβλητή var2 είναι κενή ή περιέχει μόνο ένα στοιχείο, τότε στη μεταβλητή var1 τοποθετείται η κενή λίστα.

3.1.26 Εντολή readv

Σύνταξη:

readv (index, dest)

Λειτουργία:

Τοποθετεί στη μεταβλητή με όνομα dest το περιεχόμενο της θέσης με δείκτη index του αποθηκευτικού χώρου της ταινίας. Επειδή η ταινία περιέχει μόνο αριθμούς, η μεταβλητή που θα δημιουργηθεί θα έχει ως τιμή αριθμό. Αν η μεταβλητή dest δεν υπάρχει, τότε δημιουργείται εκείνη τη στιγμή.

3.1.27 Εντολή writev

Σύνταξη:

writev (index src)

Λειτουργία:

Γράφει στη θέση της ταινίας με δείκτη index το περιεχόμενο της μεταβλητής src. Η μεταβλητή src πρέπει να έχει ως περιεχόμενο υποχρεωτικά κάποιον αριθμό.

3.1.28 Σχόλια

Σύνταξη:

[... σχόλια ...]

Λειτουργία:

Με τα σχόλια δηλώνεται ένα κείμενο που αγνοείται από το διερμηνέα. Στις αγκύλες των σχολίων μπορεί να περικλειστεί οποιαδήποτε ακολουθία των εξής συμβόλων:

- αριθμών από 0 ως 9
- γραμμάτων από 'A' ως 'Z" και από 'a' ως 'z'
- σημείων στίξης: τελεία ("."), κόμμα (","), κενό (" "), ίσον ("="), συν ("+"), μείον ("-"), παρενθέσεις ("(" και ")") και αγκύλες ("{" και "}")

3.1.29 Πραγματικός αριθμός

Σύνταξη:

πραγματικός αριθμός

Λειτουργία:

Η παρουσία ενός πραγματικού αριθμού ως εντολή στο πρόγραμμα αγνοείται από τη γλώσσα. Χρησιμοποιείται από τις κατάλληλες εντολές της γλώσσας (πράξεις, εκτύπωση στην οθόνη) μόνο όταν τοποθετηθεί σε μια μεταβλητή.

3.1.30 Συμβολοσειρά

Σύνταξη:

" συμβολοσειρά "

Λειτουργία:

Η παρουσία μιας συμβολοσειράς ως εντολή στο πρόγραμμα αγνοείται από τη γλώσσα. Χρησιμοποιείται από τις κατάλληλες εντολές της γλώσσας (πράξεις, εκτύπωση στην οθόνη) μόνο όταν τοποθετηθεί σε μια μεταβλητή. Οι εγκυροι χαρακτήρες που μπορεί να περιλαμβάνει μια συμβολοσειρά είναι οι ίδιοι με αυτούς που αναερθηκαν παραπάνω για τη δήλωση των σχολίων.

3.2 Χειρισμός αναπαράστασης στη μνήμη

Το κύριο χαρακτηριστικό της AMK είναι η υποστήριξη από μέρους της εντολών για το διάβασμα και την τροποποίηση των ίδιων των εντολών ενός προγράμματος, καθώς και για την κατασκευή καινούριων. Η υποστήριξη αυτή προσφέρεται από το ίδιο το επίπεδο της γλώσσας και δεν υπάρχει η ανάγκη ο προγραμματιστής να αναζητήσει την αναπαράσταση των εντολών σε χαμηλότερο επίπεδο, όπως συμβαίνει στη συμβολική γλώσσα που αναγκαστικά γράφει και διαβάζει από τη μνήμη bytes σε γλώσσα μηχανής.

Το πρόγραμμα παριστάνεται στη μνήμη σε μορφή όχι διαφορετική δομικά από το αρχείο πηγαίου του κώδικα. Κάθε σώμα κώδικα είναι και μια λίστα από εντολές(<stmt_body> σύμφωνα με το EBNF συντακτικό διάγραμμα) και ανήκει σε κάποια εντολή label, set, for, while, if, if ... else, η εντολή αυτή είναι δηλαδή πατέρας της. Επομένως κάθε πρόγραμμα είναι ένα δένδρο στη μνήμη με μη τελικούς κόμβους τις εντολές αυτές και τελικούς κόμβους τις υπόλοιπες. Στη ρίζα του θα βρίσκεται η βασική ετικέττα του προγράμματος, η πιο εξωτερική, η παρουσία της οποίας είναι υποχρεωτική.

Από τις εντολές που περιέχουν ένα σώμα εντολών, δύο είναι ιδιαίτερα σημαντικές, η label και η set. Όπως αναφέρθηκε και στο εγχειρίδιο των εντολών της γλώσσας AMK που προηγήθηκε, η εντολή label ορίζει ένα μνημονικό όνομα για τη θέση ενός σώματος κώδικα στη μνήμη και η εντολή set τοποθετεί ένα τμήμα κώδικα της AMK σε μια μεταβλητή. Πρέπει να παρατηρηθεί ότι αυτός ο κώδικας είναι υποχρεωτικά σωστός επειδή γίνεται έλεγχός του από το συντακτικό αναλυτή σαν να ήταν απλά κώδικας του προγράμματος και ας μην εκτελείται. Με αυτόν τον τρόπο είναι σίγουρο ότι ένα πρόγραμμα που μόλις ξεκίνησε την εκτέλεσή του θα έχει στη διάθεσή του μόνο έγκυρο κώδικα σε μεταβλητές για να κατασκευάσει ό,τι θέλει στη μνήμη.

Βασικό ρόλο στην αυτομεταβολή του κώδικα είναι και οι εντολές για να γίνει το πραγματικό διάβασμα και τροποποίηση της μνήμης, οι εντολές fetch, store και store2.

- Η εντολή fetch διαβάζει το περιεχόμενο της ετικέττας που της δίνεται, δηλαδή το υποδένδρο του προγράμματος που αντιστοιχεί σε αυτή και το τοποθετεί σε μια μεταβλητή για να το χειριστεί το πρόγραμμα. Η ανάθεση αυτή είναι αντιγραφή, άρα δεν επηρεάζεται το περιεχόμενο της ετικέττας αν αλλάξει το περιεχόμενο της μεταβλητής.

- Η εντολή store κάνει το ακριβώς αντίθετο από τη fetch, αντικαθιστά το υποδένδρο που αρχίζει από μια ετικέττα με αυτό που βρίσκεται σε μια μεταβλητή. Όπως αναφέρθηκε σε προηγούμενο κεφάλαιο, αυτή η ενέργεια, που δείχνει την πραγματική ικανότητα του αυτομεταβαλλόμενου κώδικα, δεν είναι πάντα ακίνδυνη. Υπάρχει η περίπτωση να τροποποιείται τμήμα κώδικα από το οποίο εξαρτάται η εντολή store, όπως η διαγραφή του βρόχου στον οποίο βρίσκεται. Σε αυτήν την περίπτωση η πρόβλεψη του νέου σημείου από το οποίο θα συνεχίσει η εκτέλεση είναι πολύ δύσκολη και πολύπλοκη συνάρτηση του κώδικα που ήδη υπάρχει. Επειδή στόχος της AMK είναι ο ασφαλής προγραμματισμός και η παραγωγή καλογραμμένου κώδικα, τέθηκε ο εξής περιορισμός: η store λειτουργεί όπως παραπάνω μόνο αν η ετικέττα που τροποποιεί δεν είναι πρόγονός της, δηλαδή δε μπορεί να βρεθεί πάνω στο μονοπάτι που τη συνδέει με τη ρίζα. Σε αντίθετη περίπτωση, η store εξακολουθεί να εκτελεί την εργασία της αλλά με μια επιπλέον ενέργεια, την επενεκτέλεση της ετικέττας αυτής. Αυτή η ενέργεια εκ πρώτης όψης μπορεί να μοιάζει αυθαίρετη αλλά η ετικέττα αυτή είναι το πιο μακρινό σημείο από την εντολή store που δεν είναι ασφαλές να αγνοηθούν οι συνέπειες της μεταβολής του από αυτή. Οτιδήποτε μέσα σε αυτή είναι πιθανό, όχι απλά να τροποιηθεί, αλλά να μην υπάρχει καν μετά την εκτέλεση της store, του εαυτού της συμπεριλαμβανομένου. Για αυτό το λόγο πρέπει να επανεκτελεστεί ο ύποπτος κώδικας πάλι. Με αυτήν τη σύμβαση ο προγραμματιστής της AMK είναι πάντα σίγουρος για το σημείο στο οποίο προκειται στη συνέχεια να μεταφερθεί ο έλεγχος του ρπογράμματος.

- Η εντολή store2 είναι παρόμοια με τη store αλλά διαφέρει ως προς το είδος της παρέμβασής της στο δένδρο του προγράμματος. Ενώ η store τροποποιεί τα περιεχόμενα μιας ετικέττας, η store2 διαγράφει εντελώς την ετικέττα από το πρόγραμμα και στη θέση της εισάγει το σώμα των εντολών που περιέχονται σε μια μεταβλητή. Επίσης αντικαθιστά κάθε εντολή vlabel που θα βρεί στο εσωτερικό της ετικέττας με μια εντολή label. Ο σκοπός της store2 σε συνδυασμό με τη vlabel είναι η υλοποίηση αναδρομικών τροποποιήσεων σε σώματα κώδικα στη μνήμη χωρίς την ανάγκη χρήσης φωλιασμένων ετικεττών, μεταβλητών ετικεττών, πολλαπλά ορισμένων ετικεττών ή άλλης πολύπλοκης λύσης. Για παράδειγμα ο παρακάτω κώδικας κατασκευάζει μια κενή δομή από τρείς φωλιασμένους βρόχους while:

```
label whilecmds {

  set a { 3 }
  set z { 9 }

  set x {

    [ Βρόχος while που περιέχει σημείο εισόδου για κάθε store2]
    while z {
      vlabel modifyme { }
    }

  }

  [ Τοποθέτηση των φωλιασμένων εντολών στη μνήμη ]
```

```
    for a {

      store2 (modifyme x)

    }

    [ Διαβασμα της δομής που δημιουργήθηκε ]
    fetch (constr modifyme)

    [ Εκτύπωσή της στην οθόνη ]
    write (constr)
    writeln

    [ Διαγραφή της ετικέτας γιατί να τερματίσει ομαλά το πρόγραμμα ]
    [ αλλιώς θα εκτελεστεί ένας ατέρμων βρόχος αφού στο σώμα των ]
    [ while η τιμή της μεταβλητής x δεν αλλάζει ]
    [ Γίνεται μια ιδιόμορφη χρήση της fetch για τη διαγραφή αυτή ]
    fetch (null modifyme)
    store (outer null)

    [ Βοηθητικό σημείο για διαγραφή των κατασκευασμένων εντολών ]
    label outer {

      [ Ετικέττα από την οποία αρχίζει αναδρομικά η κατασκευή ]
      label modifyme {

      }
    }

  }
```

Με τον παραπάνω μηχανισμό που μόλις περιγράφηκε, είναι φανερό ότι όλες οι δυνατότητες της γλώσσας για αυτομεταβαλλόμενο κώδικα ανάγονται σε τέσσερεις εντολές και τα μόνα σημεία στη μνήμη που μπορούν να μεταβληθούν είναι αυτά που αντιστοιχίζονται σε μια ετικέττα. Με αυτόν τον τρόπο είναι εύκολο να παρατηρηθεί η πορεία της εκτέλεσης ενός προγράμματος και συστηματοποιείται η θεώρηση των αυτομεταβαλλόμενων χαρακτηριστικών της γλώσσας ξεχωριστά από τα πιο τυπικά προστακτικά που επίσης διαθέτει.

Όμως, φαίνεται ότι τελικά ο μηχανισμός των ετικεττών είναι αρκετά ισχυρός για να υλοποιηθούν όλες οι χρήσιμες ιδιότητες του αυτομεταβαλλλόμενου κώδικα χωρίς βλάβη της γενικής περίπτωσης που αφορά ένα πρόγραμμα που έχει γενικευμένη πρόσβαση σε όλη τη μνήμη. Επίσης αποκαλύπτεται ότι οι δυνατότητες αυτομεταβαλλόμενου κώδικα μπορούν πολύ απλά να προστεθούν ως άλλη μια προέκταση μιας ήδη υπάρχουσας γλώσσας, λόγω της απουσίας αλληλεπιδράσεων των παραπάνω εντολών με τις υπόλοιπες.

3.3 Δυναμικές μεταβλητές

Κατά την κατασκευή νέων τμημάτων κώδικα στη μνήμη προκύπτει το ζήτημα του αποθηκευτικού χώρου που χρειάζονται, καθώς και η μέθοδος με την οποία μπορούν να αναφέρονται σε αυτόν. Είναι εμφανής η ανάγκη για κάποιο τρόπο δυναμικής εκχώρησης μνήμης που να είναι συνεπής με τη γλώσσα και να επιτρέπει τη δημιουργία νέων μεταβλητών, το πλήθος των οποίων δεν είναι γνωστό εκ των προτέρων.

Ο μηχανισμός αυτός στην AMK υλοποιείται με το μηχανισμό της ταινίας, ενός δυναμικού χώρου στη μνήμη που μπορεί να αυξάνεται ανάλογα με τις ανάγκες του προγράμματος. Η ταινία είναι χωρισμένη σε κελιά (cells), κάθε ένα από τα οποία είναι αριθμημένο με έναν ακέραιο δείκτη και χωρά ακριβώς έναν πραγματικό αριθμό ενώ έγινε επιλογή των πραγματικών αριθμών γιατί είναι πιο χρήσιμος από αυτόν των συμβολοσειρών.

Το σύστημα της AMK είναι υπεύθυνο για την καταγραφή του τρέχοντος μεγέθους της ταινίας και την αύξησή του ανάλογα με τη ζήτηση του προγράμματος. Οι εντολές με τις οποίες χρησιμοποιείται η ταινία είναι οι εξής δύο:

- Η εντολή readv που διαβάζει κάποια αριθμημένη θέση της ταινίας και τοποθετεί τα περιεχόμενά της σε μια μεταβλητή. Αν η θέση αυτή δεν υπάρχει, αν δηλαδή ο δείκτης προς αυτό το κελί είναι μεγαλύτερος του μεγέθους της ταινίας, αυτή ενημερώνεται κατάλληλα και αρχικοποιούνται τόσα κελιά όσα χρειάζονται για την κάλυψη αυτού του κενού.

- Η εντολή writev που μεταφέρει τα περιεχόμενα μιας μεταβλητής που περιέχει έναν πραγματικό αριθμό στην αντίστοιχη θέση της ταινίας που περιγράφεται από ένα δείκτη. Αν η θέση αυτή βρίσκεται εκτός των ορίων της ταινίας, τότε η τελευταία μεγαλώνει αντίστοιχα για να ανταποκριθεί σε αυτήν την απαίτηση.

Με το παραπάνω ζεύγος εντολών επιτυγχάνεται η ανάγνωση και η εγγραφή της ταινίας ενώ η δυναμική εκχώρηση μνήμης από αυτή γίνεται έμμεσα από τη γλώσσα AMK, χωρίς να μεσολαβεί ειδική αίτηση από την πλευρά του αυτομεταβαλλόμενου κώδικα.

Κεφάλαιο 4

Υλοποίηση

Ακολουθεί μια σύντομη περιγραφή της υλοποίησης του διερμηνέα της γλώσσας AMK. Θα αναφερθούν τα εργαλεία που χρησιμοποιήθηκαν, η διάρθρωσή του, ο λεκτικός και ο σημασιολογικός αναλυτής, η αποδοτικότητά του και οι εσωτερικές δομές δεδομένων που χρησιμοποιούται από αυτόν. Τέλος θα περιγραφεί η χρήση του.

4.1 Εργαλεία

Για την υλοποίηση του διερμηνέα χρησιμοποιήθηκαν τα μεταεργαλεία ocamllex και ocamlyacc, το πρώτο για την κατασκευή του λεκτικού αναλυτή και το δεύτερο για την κατασκευή του συντακτικού αναλυτή. Τα εργαλεία αυτά είναι μέρος του περιβάλλοντος της γλώσσας προγραμματισμού OCaml. Ο υπόλοιπος διερμηνέας γράφτηκε επίσης σε OCaml ώστε να κάνει απευθείας χρήση των δομών δεδομένων που προέκυψαν από αυτούς.

4.2 Διάρθρωση

Ο διερμηνέας αποτελείται από το λεκτικό αναλυτή που αναγνωρίζει τις λεκτικές μονάδες (tokens), το συντακτικό αναλυτή που αναγνωρίζει το συντακτικό δένδρο και το διερμηνέα με τα βοηθητικά του αρχεία που είναι υπεύθυνος για την εκτέλεση του προγράμματος.

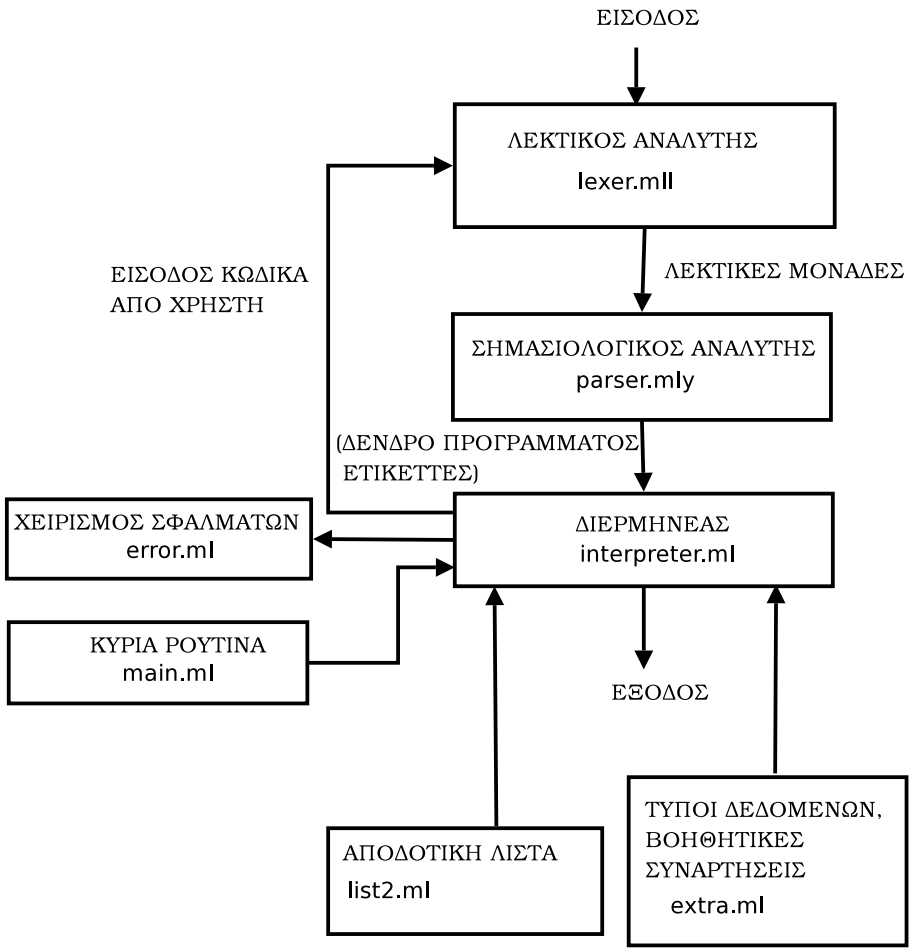
Ένας διάγραμμα της εσωτερικής αυτής διάρθρωσης, μαζί με τα ονόματα των αρχείων που την αποτελούν, φαίνεται στο σχήμα 4.1.

4.3 Λεκτικός αναλυτής

Ο λεκτικός αναλυτής κατασκευάστηκε από το εργαλείο ocamllex με βάση των αντίστοιχο αρχείο lexer.mll που περιλαμβάνει τον ορισμό των λεκτικών μονάδων. Δεν είναι πολύπλοκος αλλά έχει ένα ιδιαίτερο χρακτηριστικό: είναι υπεύθυνος για την αναγνώριση των σχολίων. Επειδή η γλώσσα AMK υποστηρίζει σχόλια στα προγράμματά της από κάποιο περιορισμένο

Πίνακας 4.1: Διάρθρωση διερμηνέα σε αρχεία.

Όνομα αρχείου	Τμήμα	Λειτουργία
Lexer.mll	Λεκτικός αναλυτής	Λεκτικός αναλυτής
Parser.mly	Συντακτικός αναλυτής	Συντακτικός αναλυτής
Interpreter.ml	Βασικό	Διερμηνέας
Main.ml	Βασικό	Κύρια ρουτίνα
List2.ml	Βοηθητικό	Τύπος διπλής λίστας, βοηθητικές συναρτήσεις
Extra.ml	Βοηθητικό	Βοηθητικές συναρτήσεις και τύποι
Error.ml	Βοηθητικό	Συναρτήσεις εξόδου στην οθόνη και εμφάνισης λαθών



Σχήμα 4.1: Αλληλεξαρτήσεις μεταξύ τμημάτων διερμηνέα.

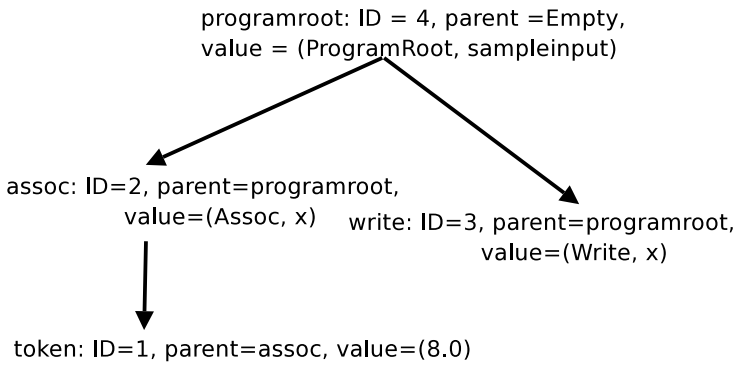
αλφάβητο, θεωρήθηκε σκόπιμο η ανάλυσή τους να γίνει από το λεκτικό αναλυτή, αφού άλλωστε δεν πρόκειται να τους αποδοθεί κάποια σημασία. Έτσι, οποιοδήποτε σχόλιο ανάγεται στη λεκτική μονάδα Trem. Τα σύμβολα που επιτρέπονται στα σχόλια είναι οι αριθμοί ('0'-'9'), τα γράμματα('A'-'Z' και 'a'-'z') και άλλα σύμβολα στίξης ('_', '.', ',', ' ', '=', '+', '-', '(', ')', '{' και '}').

4.4 Συντακτικός αναλυτής

Οι λεκτικές μονάδες που προκύπτουν από τη λεκτική ανάλυση στη συνέχεια δίνονται στο συντακτικό αναλυτή, ο οποίος εφαρμόζει τους συντακτικούς κανόνες της γλώσσας και αποφαίνεται για τη συντακτική ορθότητα του προγράμματος. Περιγράφεται από το αρχείο parser.mly που μέσω του εργαλείου ocamlyacc παράγει το αντίστοιχο πρόγραμμα σε OCaml.

Εαν το πρόγραμμα είναι συντακτικά ορθό, τότε ο συντακτικός αναλυτής δημιουργεί δύο δομές δεδομένων:

- Το δένδρο του προγράμματος που απεικονίζει τη συντακτική του δομή και περιέχει όλες τις απαιτούμενες πληροφορίες ώστε στη συνέχεια ο διερμηνέας να το εκτελέσει. Η αναπαράσταση του προγράμματος σε υψηλού επιπέδου δομή δεδομένων έχει το πλεονέκτημα ότι στη συνέχεια η δομή αυτή που είναι το ίδιο το πρόγραμμα μπορεί εύκολα να είναι προσβάσιμη στο χρήστη κατά την εκτέλεση του προγράμματος με απλές σχετικά εντολές. Το δένδρο αυτό έχει τον ακόλουθο τύπο όπως ορίζεται στο βοηθητικό αρχείο extra.ml με τις εξής αμοιβαία αναδρομικές δηλώσεις:



Σχήμα 4.2: Παράδειγμα συντακτικού δένδρου προγράμματος

τύπος περιεχόμενου κόμβου με δεδομένα:

```
type pnode = {
mutable name: string;
mutable id: int;
mutable value: lang_value;
mutable parent: program_node ref;
mutable subtree: (program_node ref List2.list2) ref
}
```

γενικός τύπος κόμβου:

```
and program_node =
  Empty
| Node of pnode
```

Για παράδειγμα, για το παρακάτω πρόγραμμα, το δένδρο που δημιουργείται εμφανίζεται στο σχήμα που ακολουθεί.

```
label sampleinput {

  set x { 8 }
  write (x)

}
```

- Ένας πίνακας κατακερματισμού (hashtable) που αποτελεί μία συντομογραφία του δένδρου για λόγους αποδοσης και εύκολης χρήσης του. Αποτελείται από ζεύγη τιμών (συμβολοσειρά, δείκτης σε δένδρο) που συμβολίζουν κάθε ετικέττα και το υποδένδρο που αντιστοιχεί στον κώδικα που αυτή περιέχει. Έτσι μια πρώτη ανάλυση των σημείων τροποποίησης του προγράμματος, όπως αυτά ορίζονται από τις εντολές label { ... } γίνεται από το συντακτικό αναλυτή. Δεν είναι όμως αρκετή γιατί νέες ετικέττες μπορούν να δημιουργηθούν με το μηχανισμό των εντολών vlabel ακολουθούμενων από τις αντίστοιχες store2. Ο έλεγχος των δηλώσεων των ετικεττών έχει και άλλο ένα σκοπό: να εντοπίσει αν κάποια ετικέττα δηλώνεται δυο φορές μέσα στο ίδιο πρόγραμμα, αν υπάρχουν δηλαδή δυο εντολές label με το ίδιο ονομα. Αν αυτό συμβαίνει, τότε ο συντακτικός αναλυτής

τερματίζει με μήνυμα λάθους. Η συμπειφορά αυτή μπορεί να αλλάξει με επιλογή μιας μεταβλητής του αρχείου του συντακτικού αναλυτή αλλά η καθορισμένη επιλογή είναι αυτή που αναφέρθηκε, ώστε ο προγραμματιστής να αποθαρρύνεται να δημιουργεί δυσνόητα προγράμματα.

4.5 Αποδοτικότητα

4.5.1 Χρήση αναφορών

Στα παραπάνω φαίνεται ότι σε κάθε κόμβο που έχει παιδιά, αυτά ορίζονται ως ένας δείκτης σε μια λίστα δεικτών σε κόμβους. Η γενικευμένη χρήση δεικτών στους τύπους δεδομένων που χρησιμοποιούνται από το διερμηνέα οφείλεται στη χρήση απευθείας ανάθεσης σε αυτές για βελτίωση της απόδοσης του προγράμματος. Αν και η OCaml επιτρέπει συναρτησιακό προγραμματισμό, η χρήση του για την τροποποίηση λιστών θα είχε ως συνέπεια την υλοποίηση ενός πολύ πιο αργού διερμηνέα, αφού μια λίστα δε μπορεί να τροποποιηθεί αλλά μόνο να κατασκευαστεί από την αρχή η νέα της έκδοση, διαδικασία χρονοβόρα και απαιτητική. Αντίθετα, με την εισαγωγή των δεικτών, γίνεται χρήση των προστακτικών χαρακτηριστικών της OCaml που επιτρέπουν την ανάθεση σε τμήματα μιας δομής δεδομένων και άρα στην πιο αποδοτική της χρήση.

4.5.2 Τύπος διπλής λίστας

Ένα άλλο σημείο στο οποίο δόθηκε προσοχή ήταν η αναπαράσταση των λιστών. Αντί του ενσωματωμένου τύπου λίστας της OCaml (List) χρησιμοποιήθηκε ο τύπος List2 που είναι ο εξής:

```
type 'a list2 = {norm: 'a list ; inv: 'a list}
```

Δηλαδή αποτελείται από μια εγγραφή (record) με δυο λίστες. Η λίστα που συμβολίζεται με αυτόν τον τρόπο συμβολίζεται από τη συνένωση της πρώτης λίστας (norm) με την αντίστροφη της δεύτερης (inv), δηλαδή οι δύο λίστες αποτελούν τα δύο μισά της τελικής λίστας, με τη δεύτερη να είναι ταξινομημένη αντίστροφα.

Ο τακτική αυτή ακολουθήθηκε ώστε να είναι πιο αποδοτική η προσθήκη ενός στοιχείου στο τέλος της λίστας. Αν η λίστα ήταν η List της OCaml, τότε κάθε εισαγωγή στο τέλος της θα απαιτούσε χρόνο O(n) όπου n το ήδη υπάρχον μήκος της. Σε αυτή όμως την περίπτωση (List2), ο χρόνος αυτός είναι σταθερός O(1), λόγω της εισαγωγής του τελευταίου στοιχείου στην κεφαλή της λίστας inv. Κάθε φορά που μια συνάρτηση για τον τύπο List2 διατρέχει μια τέτοια δομή(για παράδειγμα για τον υπολογισμό του μήκους της ή για την εύρεση κάποιου στοιχείου), τότε εκμεταλλεύεται την περίπτωση αυτή και παράλληλα μεταφέρει τα στοιχεία της αντίστροφης λίστας inv στη λίστα norm με τη σωστή σειρά. Η δομή αυτή δεδομένων δεν δείχνει κάποια διαφορά στην απόδοση για μικρά προγράμματα και μικρό όγκο δεδομένων αλλά την εμφανίζει όταν αυτά είναι μακροσκελή ή δημιουργούν μεγάλες δομές για αποθήκευση.

4.6 Διερμηνέας

Ο διερμηνέας αποτελεί και το κύριο μέρος του προγράμματος. Το αρχείο επικεφαλίδας του (interpreter.mli) είναι το ακόλουθο:

```
val print_label : string -> program_node ref -> unit

val print_vars : 'a -> unit
```

```
val check_parent_path : program_node -> int -> bool

val replace_id :
  program_node ref List2.list2 ref ->
  int ->
  program_node ref List2.list2 ref ->
  program_node ref List2.list2 ref

val subst_vlabel : program_node ref -> program_node ref

val subst_vlabels :
  program_node ref List2.list2 ref ->
  program_node ref List2.list2 ref

val run_prog : program_node ref -> unit

val interp :
  program_node ref * (string, program_node ref) Hashtbl.t
  -> unit
```

Παρακάτω εξηγούνται όλες οι παραπάνω μεταβλητές και συναρτήσεις.

val var_table : (string, program_node ref List2.list2 ref ref) Hashtbl.t ref

Πίνακας κατακερματισμού που περιέχει το όνομα κάθε μεταβλητής σε αντιστοιχία με τα περιεχόμενά της ή μια κενή λίστα αν αυτή είναι κενή.

val label_table : (string, program_node ref) Hashtbl.t ref

Πίνακας κατακερματισμού που περιέχει το όνομα κάθε ετικέττας σε αντιστοιχία με τα περιεχόμενά της ή Empty αν αυτή είναι κενή.

val tape : float array

Συμβολίζει τον αποθηκευτικό χώρο της ταινίας, ένα μονοδιάστατο πίνακα από πραγματικούς αριθμούς. Το μήκος του θεωρητικά είναι άπειρο, στην πράξη πεπερασμένο αλλά με κάποια μεγάλη τιμή ώστε να μην αποτελεί πρόβλημα για τα περισσότερα προγράμματα.

val tape_len : int ref

Αποθηκεύει κάθε φορά το μέγιστο αριθμό θέσεων της ταινίας που χρησιμοποιούνται από το πρόγραμμα.

val root_node : program_node ref

Δείκτης στον κόμβο που είναι η ρίζα του προγράμματος.

val eDEBUG : int ref

Εσωτερική μεταβλητή που ορίζεται κατάλληλα ανάλογα με τις παραμέτρους που δίνονται στη γραμμή εντολών στο διερμηνέα ώστε να εκτυπώνει επιπλέον μηνύματα για την πορεία της εκτέλεσης του προγράμματος. Επίσης επηρεάζει την έξοδο του διερμηνέα μετά την εκτέλεση του προγράμματος ώστε να εκτυπώνει διαγνωστικά μηνύματα και την τελική κατάστασή του.

val set_debug : int -> unit

Ορίζει το βαθμό στον οποίο ο διερμηνέας θα πληροφορεί το χρήστη για τις ενέργειές του κατά την εκτέλεση του προγράμματος, θέτοντας την κατάλληλη τιμή στη μεταβλητή eDEBUG που αναφέρθηκε παραπάνω.

val c : int ref

Βοηθητική μεταβλητή που έχει το ρόλο μετρητή. Κάθε κόμβος του δένδρου περιγράφεται από ένα ξεχωριστο αριθμό, το id, που είναι μοναδικό για αυτόν. Ο ρόλος της μεταβλητής c είναι να αυξάνεται κάθε φορά που δημιουργείται ένας νέος κόμβος, ωστε να του αποδώσει ένα καινούριο και μοναδικό id.

val counter : int ref -> int

Η συνάρτηση counter δέχεται ένα δείκτη σε ένα μετρητή και τον αυξάνει. Στο συγκεκριμένο πρόγραμμα ο μετρητής αυτός θα είναι πάντα η μεταβλητή c.

val init_counter : program_node -> unit

Λόγω του τρόπου κατασκευής του δένδρου του προγράμματος, όταν αυτό δίδεται από το συντακτικό αναλυτή δεν έχει ακόμα δοθεί κάποιο αναγνωριστικό id στον κόμβο που είναι στη ρίζα του. Αυτό αναλαμβάνει αυτή η συνάρτηση.

exception Label_interrupt

Εξαίρεση που ρίχνεται από το διερμηνέα όταν καποια εντολή store τροποποιήσει μια ετικέττα, της οποίας είναι απόγονος, ώστε να αρχίσει πάλι η εκτέλεση της ετικέττας αυτής από την αρχή.

val print_var : string -> program_node ref List2.list2 ref ref -> unit

Δέχεται το όνομα μιας μεταβλητής και τα περιεχόμενά της και τα εκτυπώνει με κατανοητό τρόπο στην οθόνη.

val print_label : string -> program_node ref -> unit

Δέχεται το όνομα μιας ετικέττας και τον κόμβο της στο πρόγραμμα και την εκτυπώνει στην οθόνη.

val print_vars : 'a -> unit

Μετά την εκτέλεση ενός προγράμματος εκτυπώνει τις μεταβλητές και τις ετικέττες που αυτό δημιούργησε και το χώρο στην ταινία που τυχόν χρησιμοποίησε.

val check_parent_path : program_node -> int -> bool

Δέχεται κάποιον κόμβο ως αρχικό κόμβο και αναζητά ένα συγκεκριμένο id (που είναι η δεύτερη παράμετρος) σε κάθε κόμβο στο μονοπάτι προς τη ρίζα. Είναι βοηθητική αναδρομική συνάρτηση που χρησιμοποιείται από την εντολή store ώστε να βρεθεί αν αυτή τροποποιεί κάποιο πρόγονό της.

val replace_id : program_node ref List2.list2 ref -> int -> program_node ref List2.list2 ref -> program_node ref List2.list2 ref

Δέχεται μια λίστα, έναν αριθμό που συμβολίζει το id κάποιου κόμβου και μια επιπλέον λίστα. Στη συνέχεια, διαγράφει στην πρώτη λίστα τον κόμβο με το δοθέν id και στη θέση του εισάγει το περιεχόμενο της δεύτερης λίστας και επιστρέφει τη λίστα που μόλις δημιουργήθηκε. Είναι βοηθητική συνάρτηση για την υλοποίηση της εντολής store2 που αντικαθιστά μια ετικέττα με μια λίστα κόμβων.

val subst_vlabel : program_node ref -> program_node ref

Η αναδρομική αυτή συνάρτηση αρχίζει από έναν κόμβο και στη συνέχεια αντικαθιστά στο υποδένδρο του κάθε εντολή vlabel { } με την αντίστοιχη κενή label { }. Χρησιμοποιείται από την εντολή store2 για τη δημιουργία δυναμικών ετικεττών.

val subst_vlabels : program_node ref List2.list2 ref ->program_node ref List2.list2 ref

Χρησιμοποιεί την προηγούμενη συνάρτηση sbst_vlabel για να αλλάξει κάθε κόμβο σε μια λίστα κόμβων.

val run_prog : program_node ref -> unit

Η run_prog είναι η βασική συνάρτηση του διερμηνέα. Εφαρμόζεται σε κάθε κόμβο του συντακτικού δένδρου και αναλαμβάνει να εκτελέσει την κάθε εντολή, δίνει δηλαδή το τελικό νόημα στο πρόγραμμα.

val interp : program_node ref * (string, program_node ref) Hashtbl.t -> unit

Η συνάρτηση interp είναι ο συνδετικός κρίκος μεταξύ του συντακτικού αναλυτή και του διερμηνέα. Δέχεται το ζεύγος (δένδρο προγράμματος, πίνακας κατακερματισμού ετικεττών) και αναλαμβάνει να το εκτελέσει καλώντας τη συνάρτηση run_prog ενώ στο τέλος εκτυπώνει και τα αποτελέσματα του προγράμματος ανάλογα με το επίπεδο της πληροφόρησης (verbosity) που έχει ορίσει ο χρήστης.

4.7 Βοηθητικές συναρτήσεις

Στα αρχεία extra.ml και error.ml ορίζονται κάποιοι βοηθητικοί τύποι, μεταβλητές και συναρτήσεις για το χειρισμό λαθών και για δηλώσεις που δε μπορούν να ανήκουν σε κάποιο άλλο από τα άλλα αρχεία. Ένα επιλεγμένο σύνολο από τα σημαντικότερα από αυτά είναι:

```
type lang_value =
  EmptyVal
| StringV of string
| FloatV of float
| CommandData of string * string * string * string

type pnode = {
mutable name : string;
mutable id : int;
mutable value : lang_value;
mutable parent : program_node ref;
```

```
mutable subtree : program_node ref List2.list2 ref;
}

and program_node =
  Empty
| Node of pnode

type parsed_prog = program_node ref * (string, program_node ref)
Hashtbl.t

type program_tree = program_node ref * int
```

Οι παραπάνω τύποι, κάποιοι από τους οποίους έχουν ήδη αναφερθεί περιγράφουν πλήρως το δένδρο του προγράμματος. Το μόνο που μπορεί να σημειωθεί εδώ είναι ότι ένας κόμβος είτε είναι εκφυλισμένος με τιμή Empty ή έχει κάποιο περιεχόμενο. Στη δεύτερη περίπτωση τον χαρακτηρίζει κάποιο όνομα που μπορεί να είναι κάτι μημονικό για να το κατανοεί ο άνθρωπος (name), ένας μοναδικός ακέραιος (id), μια τιμή που μπορεί να είναι πραγματικός, συμβολοσειρά ή εντολή με μέχρι τρείς παραμέτρους ή και το κενό (value), ένα δείκτη προς τον πατέρα του (parent) και ένα δείκτη σε μια λίστα που δείχνει στα παιδιά του.

val string_of_node : program_node ref -> string

Περιγραφή ενός κόμβου ως συμβολοσειρά. Βοηθητική συνάρτηση για την εκτύπωση στην οθόνη χρήσιμων πληροφοριών.

val val_of_node : program_node ref -> string

Εμφάνιση της τιμής ενός κόμβου. Επίσης βοηθητική συνάρτηση που χρησιμοποιείται για έξοδο στην οθόνη.

val show_ptree : program_node ref -> unit

Εμφανίζει το υποδένδρο με ρίζα τον κόμβο, στον οποίο δίνεται δείκτης. Κάθε υποδένδρο τυπώνεται στην οθόνη με τη μορφή φωλιασμένων εγγραφών, μιας για κάθε κόμβο. Εγγραφές ενός επίπεδου χαμηλότερα στο δένδρο βρίσκονται μέσα στα σύμβολα '<' και '>'.

val create_node : program_node ref -> program_node ref List2.list2 ref -> program_node ref

Κατασκευαστική συνάρτηση, αναλαμβάνει να δημιουργήσει έναν κόμβο από έναν κόμβο και μία λίστα κόμβων που του δίνεται. Ο κόμβος που προκύπτει είναι ο πρώτος με τη λίστα ως υποδένδρο του (πεδίο subtree) με τη διαφορά ότι οι κόβοι της λίστας έχουν ενημερωθεί ότι αυτός είναι ο πατέρας τους μέσω του πεδίου τους parent.

val create_root : program_node -> program_node ref List2.list2 ref -> 'a -> program_node ref * 'a

Όμοια με τη συνάρτηση create_node, αλλά χρησιμοποιείται μόνο για την κατασκευή του κόμβου που βρίσκεται στη ρίζα του δένδρου.

exception Terminate

Εξαίρεση που ρίχνεται όταν ο διερμηνέας πρόκειται να τερματίσει, για παράδειγμα λόγω κάποιου λάθους ή κάποιας εξαίρεσης για την οποία ο διερμηνέας δεν παρέχει χειριστή.

type verbose = Vquiet — Vnormal — Vverbose

Οι τρόποι με τους οποίους ο διερμηνέας μπορεί να επικοινωνεί με μηνύματα με το χρήστη κατά την εκτέλεση του προγράμματος.

val fatal : ('a, Format.formatter, unit, 'b) format4 -> 'a

Εμφανίζει ένα μήνυμα στην οθόνη και τερματίζει το διερμηνέα. Χρησιμοποιείται μόνο για σοβαρά λάθη (για παράδειγμα διαίρεση με το 0) που προκαλούν σφάλμα από το οποίο δεν μπορεί να επανέλθει το πρόγραμμα. Ακολουθεί το πρότυπο του αρθρώματος Format της OCaml.

val warning : ('a, Format.formatter, unit, unit) format4 -> 'a

Εμφανίζει ένα μήνυμα στην οθόνη για κάποιο σφάλμα που δεν είναι αρκετά σοβαρό ώστε να διακοπεί η λειτουργία του προγράμματος αλλά ο χρήστης θα έπρεπε να το έχει υπ'οψη του. Ακολουθεί το πρότυπο του αρθρώματος Format της OCaml.

val message : ('a, Format.formatter, unit, unit) format4 -> 'a

Εμφανίζει ένα απλό μήνυμα στην οθόνη, ακολουθούμενο από αλλαγή γραμής. Ακολουθεί το πρότυπο του αρθρώματος Format της OCaml.

4.8 Χρήση του διερμηνέα AMK

Ο διερμηνέας της γλώσσας AMK δέχεται το αρχείο που θα εκτελεστεί από τη γραμμή εντολών, μαζί με κάποιες βοηθητικές παραμέτρους.

Για παράδειγμα η παρακάτω εντολή εμφανίζει ποιές είναι αυτές και μια σύντομη περιγραφή τους.

```
[george@localhost amk]$ ./amk --help

Usage: amk [options] file...

Options:
  -p Pretty print
  -i Intepret (default)
  -v Verbose
  -q Quiet
  -V Print compiler version and exit
  -help Display this list of options
  --help Display this list of options
```

Αναλυτικά αυτές είναι:

- -p: Εμφανίζει το συντακτικό δέντρο του προγράμματος με μια μορφή κατανοητή στο χρήστη
- -i: Καλεί το διερμηνέα για το αρχείο που δόθηκε. Η παράμετρος αυτή είναι προεπιλεγμένη και αν δε δοθεί κάποια άλλη που να έρχεται σε σύγκρουση με αυτή, τότε εννοείται.
- -v: Κατά τη διάρκεια της εκτέλεσης του προγράμματος και μετά εκτυπώνει χρήσιμες πληροφορίες, όπως τα περιεχόμενα της μνήμης (μεταβλητές, ταινία) και οι ετικέττες που χρησιμοποιήθηκαν από το πρόγραμμα και η σειρά με την οποία εκτελέστηκε.
- -q: Δεν δείχνει μηνύματα στην έξοδο.

- -V: Εμφανίζει μόνο την έκδοση του διερμηνέα.
- -help ή –help: Εμφανίζει το παραπάνω σύντομο βοηθητικό μήνυμα.

Για παράδειγμα, έστω το παρακάτω αρχείο trivial.amk:

```
label trivial {

set x { 1 }
set y { -2 }
set z { 3.14159 }
+ a x y

set codee { write (y) writeln }
write (z)
writeln

label writer {
  write (x)
  writeln
}

label decc {

  fetch (code writer)
  store (decc codee)

}

write (a) writeln

}
```

Τότε κάποιοι τρόποι να εκτελεστεί είναι:

```
[george@localhost amk]$ ./amk trivial.amk
3.14159
1
-2
-1
-----done-------

[george@localhost amk]$ ./amk -v trivial.amk
AMK interpreter, version 0.3a
Processing file: trivial.amk
programroot is valid!
Running node [programroot]:
Running node [assoc]:
Running node [assoc]:
Running node [assoc]:
Running node [add]:
Running node [assoc]:
Running node [write]:
```

```
3.14159Running node [writeln]:

Running node [label]:
Running node [write]:
1Running node [writeln]:

Running node [label]:
Running node [fetch]:
Storing label writer in var code
Running node [store]:
Storing var codee in label decc
Running node [var-stored-label]:
Running node [write]:
-2Running node [writeln]:

Running node [write]:
-1Running node [writeln]:

-----done-------
Variables:
* Variable a
<element var-float[id=0, parent=ProgramRoot trivial  ]: value=-1.
>
* Variable x
<element token[id=1, parent=Assoc x  ]: value=1.
>
* Variable y
<element token[id=3, parent=Assoc y  ]: value=-2.
>
* Variable z
<element token[id=5, parent=Assoc z  ]: value=3.14159
>
* Variable codee
<element write[id=8, parent=Label decc  ]: value=Write y
>
<element writeln[id=9, parent=Label decc  ]: value=WriteLn
>
* Variable code
<element write[id=13, parent=Label writer  ]: value=Write x
>
<element writeln[id=14, parent=Label writer  ]: value=WriteLn
>
Labels:
* Label decc
<element var-stored-label[id=18, parent=ProgramRoot trivial  ]: value=Label decc

<element write[id=8, parent=Label decc  ]: value=Write y
>
<element writeln[id=9, parent=Label decc  ]: value=WriteLn
>
>
```

```
* Label writer
<element label[id=15, parent=ProgramRoot trivial  ]: value=Label writer
<element write[id=13, parent=Label writer  ]: value=Write x
>
<element writeln[id=14, parent=Label writer  ]: value=WriteLn
>
>
Tape:
0.000000
---Program:----
<element programroot[id=21, parent=*Empty*]: value=ProgramRoot trivial
<element assoc[id=2, parent=ProgramRoot trivial  ]: value=Assoc x
<element token[id=1, parent=Assoc x  ]: value=1.
>
>
<element assoc[id=4, parent=ProgramRoot trivial  ]: value=Assoc y
<element token[id=3, parent=Assoc y  ]: value=-2.
>
>
<element assoc[id=6, parent=ProgramRoot trivial  ]: value=Assoc z
<element token[id=5, parent=Assoc z  ]: value=3.14159
>
>
<element add[id=7, parent=ProgramRoot trivial  ]: value=Add a x y
>
<element assoc[id=10, parent=ProgramRoot trivial  ]: value=Assoc codee
<element write[id=8, parent=Label decc  ]: value=Write y
>
<element writeln[id=9, parent=Label decc  ]: value=WriteLn
>
>
<element write[id=11, parent=ProgramRoot trivial  ]: value=Write z
>
<element writeln[id=12, parent=ProgramRoot trivial  ]: value=WriteLn
>
<element label[id=15, parent=ProgramRoot trivial  ]: value=Label writer
<element write[id=13, parent=Label writer  ]: value=Write x
>
<element writeln[id=14, parent=Label writer  ]: value=WriteLn
>
>
<element label[id=18, parent=ProgramRoot trivial  ]: value=Label decc
<element fetch[id=16, parent=Label decc  ]: value=Fetch code writer
>
<element store[id=17, parent=Label decc  ]: value=Store decc codee
>
>
<element write[id=19, parent=ProgramRoot trivial  ]: value=Write a
>
<element writeln[id=20, parent=ProgramRoot trivial  ]: value=WriteLn
>
```

>

Κεφάλαιο 5

Παραδείγματα

5.1 Υλοποίηση εντολής eval( )

Σε αρκετές γλώσσες προγραμματισμού όπως οι Perl, Tcl, LISP, Python και κάποιες εκδόσεις της BASIC, παρέχεται στον προγραμματιστή μία εντολή, συνήθως με το όνομα eval, αν και αυτό εξαρτάται από τη συγκεκριμένη γλώσσα και υλοποίησή της που είναι υπεύθυνη για δυναμική εκτέλεση κώδικα. Συγκεκριμένα σε αυτή δίνεται ως παράμετρος κώδικας είτε με τη μορφή συμβολοσειράς είτε με τη μορφή κώδικα όπως παρίσταται στη γλώσσα και στη συνέχεια το σύστημα της γλώσσας αναλαμβάνει:

- να τον αναλύσει αν δεν είναι σε κάποια έτοιμη μορφή (για παράδειγμα στην περίπτωση που δίνεται ως απλή συμβολοσειρά)
- να τον εκτελέσει

Το πρώτο βήμα έχει να κάνει με την πιστοποίηση των δεδομένων που δίνονται ότι πράγματι αυτά μπορούν να μετατραπούν σε κώδικα της γλώσσας σύμφωνα με τους κανόνες της. Για παράδειγμα, το αντίστοιχο βήμα για AMK πρέπει να απορρίπτει δεδομένα της μορφής, δηλαδή να διατηρεί τις εντολές της γλώσσας ως αδιαίρετες μονάδες:

```
set x { set Y { {]}
```

Η παραπάνω απαίτηση είναι άλλωστε και ένας από τους βασικούς στόχους αυτής της προσπάθειας για δημιουργία ενός ασφαλούς συστήματος AMK.

Το δεύτερο βήμα στη συνέχεια αναλαμβάνει να εκτελέσει την αναπαράσταση του συντακτικά σωστού κώδικα που προέκυψε από το πρώτο βήμα.

Η παραπάνω συμπεριφορά μπορεί να προσομοιωθεί στην AMK από ένα πρόγραμμα που τοποθετεί σε μία μεταβλητή κώδικα και στη συνέχεια τον εκτελεί, βάζοντας τον σε μία ετικέττα που αμέσως ακολουθεί και άρα εκτελείται αμέσως μετά. Ως κώδικα θεωρούμε ότι δίνεται πρόγραμμα επίσης σε AMK από την είσοδο ή μέσω αναθέσεων της γλώσσας. Η δεύτερη προσέγγιση εγγυάται τη συντακτική ορθότητα του κώδικα, αφού αυτή εγγυάται από τις ίδιες τις εντολές του AMK. Αντίθετα, στην περίπτωση της εισόδου από το πληκτρολόγιο πρέπει να κληθούν πάλι ο λεκτικός και ο σημασιολογικός αναλύτής της γλώσσας ώστε να ελέγξουν την εγκυρότητα της εισόδου.

Ένα σημείο που πρέπει να προσεχθεί είναι το περιβάλλον του νέου προγράμματος. Επειδή ο νέος κώδικας τρέχει στο ίδιο περιβάλλον με τον ήδη εκτελούμενο, έχει πρόσβαση στις ίδιες μεταβλητές και ετικέττες. Αυτή η συμπεριφορά επιλέχθηκε ώστε ο κώδικας να μπορεί πραγματικά να επηρεάσει το νέο πρόγραμμα, αλλιώς ανάγεται στην περίπτωση των γλώσσών που μπορούν να καλούν άλλα προγράμματα από τον κώδικά τους, χωρίς όμως να μπορούν να επηρεαστούν από αυτά στο επίπεδο της ίδιας της γλώσσας. Τέτοιο παράδειγμα είναι η οικογένεια εντολών exec( ) της C που δημιουργεί μία νέα διεργασία αλλά αυτή μπορεί να μιλήσει με το πρόγραμμα που τη δημιούργησε μόνο με μεθόδους του λειτουργικού συστήματος (σωληνώσεις, αρχεία).

5.1.1 Self-debugger

Το παρακάτω τμήμα κώδικα μπορεί να παρεμβληθεί σε οποιοδήποτε σημείο ενός προγράμματος εντοπίζεται ένα πρόβλημα και δίνει τον έλεγχο στο χρήστη, ο οποίος και μπορεί να πληκτρολογήσει ένα πρόγραμμα που θα εκτελεστεί εμβόλιμα και με το οποίο μπορεί να εξετάζει τα περιεχόμενα κάποιων μεταβλητών, να τα τροποποιήσει ή να παρέμβει με δραστικότερους τρόπους στο πρόγραμμα που ήδη τρέχει, όπως να τροποποιήσει τα περιεχόμενα μιας ετικέττας που ακολουθεί.

```
label SelfDebugger {

[ κώδικας... ]
...

[ Διαβάζει από την είσοδο μία συμβολοσειρά και δημιουργεί ]
[ τον αντίστοιχο κώδικα από αυτή ]
readcode(cmd)

[ Τοποθετεί τον κώδικα στην ετικέττα DebugLabel ]
store (DebugLabel cmd)

[ Αμέσως μετά η ετικέττα εκτελείται ]
label DebugLabel { }

[ κώδικας... ]
...

}
```

Για παράδειγμα, ο παρακάτω κώδικας:

```
label SelfDebugger1 {

[ Δημιουργία 2 μεταβλητών και πράξη μεταξύ τους ]
set x { 23 }
set y { 27 }
+ x x y

[ Διαβάζει από την είσοδο μία συμβολοσειρά και δημιουργεί ]
[ τον αντίστοιχο κώδικα από αυτή ]
readcode(cmd)

[ Τοποθετεί τον κώδικα στην ετικέττα DebugLabel ]
store (DebugLabel cmd)

[ Αμέσως μετά η ετικέττα εκτελείται ]
label DebugLabel { }

write(x)
writeln

}
```

έχει το εξής αποτέλεσμα:

```
[george@localhost amk]$ ./amk 5.1.amk
input:
label tester { set x { 89 } write(y) writeln }
27
89
-----done-------
```

5.1.2 Διερμηνέας

Από τη στιγμή που ο λεκτικός και ο σημασιολογικός αναλυτής μπορούν να χρησιμοποιηθούν σε επίπεδο γλώσσας ώστε να εξετάζουν κώδικα, ο οποίος στη συνέχεια θα εκτελεστεί, είναι φανερό ότι ένας διερμηνέας AMK γραμμένος σε AMK που δέχεται προγράμματα από το πληκτρολόγιο είναι μία πάρα πολύ απλή περίπτωση:

```
label Interpreter {

  label main {

    readcode (program)
    store (main program)

  }

}
```

Η εντολή store τροποποιεί μία ετικέττα στην οποία και βρίσκεται, άρα η εκτέλεση θα αρχίσει πάλι από τα νέα περιεχόμενα της ετικέττας main. Η τοποθέτηση της εντολής store μέσα στην ετικέττα που τροποποιεί έχει ως αποτέλεσμα μια πολύ μικρή εξοικονόμηση μνήμης: οι δύο εντολές readcode και store, όταν χρησιμοποιηθούν είναι πια άχρηστες και συμφέρει από πλευράς οικονομίας μνήμης να αντικατασταθούν από το νέο πρόγραμμα.

Η έξοδος στην οθόνη τότε είναι:

```
[george@localhost amk]$ ./amk 5.2.amk
input:
label new_prog { set one { 1 } write (one) writeln }
1
-----done-------
```

5.2 Αναδρομή ουράς

Μια κλήση της συνάρτησης f από τον εαυτό της ονομάζεται αναδρομή ουράς (tail recursion) αν είναι το τελευταίο πράγμα που κάνει η f πριν επιστρέψει. Μια τέτοια συνάρτηση μπορεί να απλοποιηθεί κατά την υλοποίησή της σύμφωνα με το μετασχηματισμό βελτιστοποίησης κλήσεως ουράς (tail call optimization) [15] αν αντί η κλήση στο τέλος της αντικατασταθεί από ένα απλό άλμα στην αρχή της συνάρτησης. Αν γίνει χρήση της ιδιότητας της store να προκαλεί την επανεκτέλεση του εσώτερου σώματος κώδικα που επηρεάζεται από αυτήν αν την περιέχει, το παραπάνω άλμα μπορεί να προσομοιωθεί από μια εντολή store στο τέλος της συνάρτησης. Η εντολή αυτή θα έχει τη μορφή store (f f_bdy) όπου f το όνομα της ετικέττας στην οποία βρίσκεται η συνάρτηση f και f_body το περιεχόμενο της συνάρτησης f.

Για παράδειγμα η συνάρτηση παραγοντικό θα είναι η ακόλουθη:

```
label factorial {

  read (x)

  set fact { 1 }

  set one { 1 }

  label factlab {

    if x {

      * fact fact x
      - x x one

      fetch (k factlab)

      store (factlab k)

    }

  }

  writeln
  write(fact)
  writeln

}
```

Φαίνεται η μεταβλητή fact που προορίζεται να έχει το αποτέλεσμα της συνάρτησης και η μεταβλητή x που περιέχει το όρισμα που δίνεται κάθε φορά. Με αυτές τις δύο μεταβλητές επιτυγχάνεται η μεταφορά της πληροφορίας για ανάμεσα στις διαδοχικές εκτελέσεις της συνάρτησης.

Μια εκτέλεση της συνάρτησης του παραγοντικού είναι η ακόλουθη:

```
[george@localhost amk]$ ./amk 5.3.amk
input:
6
720
-----done-------
```

5.3 Φωλιασμένοι βρόχοι επανάληψης

Ένα ενδιαφέρον πρόβλημα που δείχνει την ευελιξία του αυτομεταβαλλόμενου κώδικα είναι το εξής:

> *"Να κατασκευαστεί ένα πρόγραμμα που να ζητά από το χρήστη έναν αριθμό N στο διάστημα [1...9] και στη συνέχεια να εκτυπωθούν χωρίς επαναλήψεις όλοι οι N-ψήφιοι αριθμοί που κάθε ψηφίο τους ανήκει στο διάστημα [1...N]."*

Το παραπάνω πρόβλημα μπορεί να λυθεί με τις συνηθισμένες γλώσσες διαδικαστικού προγραμματισμού αλλά η υλοποίησή του συνήθως είναι πολύπλοκη και όχι προφανής. Είναι φανερό

ότι μπορεί να υλοποιηθεί ως ένα σύνολο από N φωλιασμένους βροχους επανάληψης από το 1 ως το N στο εσωτερικό των οποίων βρίσκεται μια εντολή εκτύπωσης όλων των μεταβλητών των βρόχων αυτών. Η υλοποίηση αυτή είναι αρκετά κομψή και κατανοητή αλλά έχει το μειονέκτημα ότι ο αριθμός των βρόχων δεν είναι γνωστός από πριν και για αυτόν το λόγο δε μπορούν να κατασκευαστούν για τη γενική περίπτωση.

Στην περίπτωση όμως του αυτομεταβαλλόμενου κώδικα είναι δυνατή η κατασκευή των N βρόχων στη διάρκεια της εκτέλεσης του προγράμματος και συνέχεια η εκτέλεσή τους ώστε να προκύψει το ζητούμενο αποτέλεσμα.

Για αυτό το λόγο θα χρησιμοποιηθεί ο μηχανισμός των δυναμικών ετικετών για την κατασκευή τους. Επίσης θα χρησιμοποιηθεί ο χώρος αποθήκευσης της ταινίας για τη δημιουργία των δυναμικών μεταβλητών που χρειάζονται για τις θέσεις των ψηφίων. Κάθε τέτοια δυναμική μεταβλητή χρειάζεται και μία θέση στην ταινία.

Τελικά τα τμήματα κώδικα που θα χρησιμοποιηθούν είναι δύο:

- το τμήμα ffor_cmd που κάθε περιέχει στο εσωτερικό του τη δυναμική ετικέττα ff, στην οποία και θα τοποθετηθεί ο επόμενος βρόχος ή η εντολή της εκτύπωσης όλων των μεταβητών αν έχουν ήδη κατασκευαστεί όλοι οι εξωτερικοί βρόχοι και δηλώνεται ως εξής:

```
set ffor_cmd {

  [ Εύρεση του δείκτη της δυναμικής μεταβλητής στην ταινία ]

  + env env one

  [ Μηδενισμός της ]

  writev(env zero)

  - env env one

  for times {
    + env env one
    [ Αυξάνει το περιεχόμενο της θέσης της δυναμικής ]

    [ μεταβλητής στην ταινία κατά 1 ]}
    readv(env var)

    + var var one

    writev(env var)

    [ Σε αυτό το σημείο θα προστεθεί κώδικας ]
    vlabel ff { }

    - env env one

  }

}
```

- το τμήμα printer που εκτυπώνει όλες τις δυναμικές μεταβλητές από την ταινία και δηλώνεται ως εξής:

```
    set printer {
      label print_d {
        set index { 1 }
        [ Εκτυπώνει όλες τις δυναμικές μεταβλητές από την ταινία ]
        for times {
          readv (index var)
          write (var)
          + index index one
        }
      }
      writeln
    }
```

Το πλήρες πρόγραμμα τότε που θα δημιουργεί το ζητούμενο κώδικα και στη συνέχεια θα τον εκτελεί είναι το εξής:

```
label ffors {

[ Παράδειγμα (times) φωλιασμένων (for) με τρόπο που ]
[ να φαίνεται η γενική περίπτωση ]
set x { 0 }
set zero { 0 }
set one { 1 }
set two { 2 }

read(times)

[ Αρχικός μετρητής ‘‘περιβάλλοντος’’ που φροντίζει για τη σωστή ]
[ πρόσβαση στα περιεχόμενα της ταινίας ]
set env { 0 }

set ffor_cmd {

  [ Εύρεση του δείκτη της δυναμικής μεταβλητής στην ταινία ]
  + env env one

  [ Μηδενισμός της ]
  writev(env zero)

  - env env one
```

```
  for times {

    + env env one

    [ Αυξάνει το περιεχόμενο της θέσης της δυναμικής ]
    [ μεταβλητής στην ταινία κατά 1 ]
    readv(env var)

    + var var one

    writev(env var)

    [ Σε αυτό το σημείο θα προστεθεί κώδικας ]
    vlabel ff { }

    - env env one

  }

}

set printer {

label print_d {

  set index { 1 }

  [ Εκτυπώνει όλες τις δυναμικές μεταβλητές από την ταινία ]

  for times {

    readv (index var)

    write (var)

    + index index one

  }

}

writeln

}

[ Κατασκευή των βρόχων ]
for times {

store2 (ff ffor_cmd)

}
```

```
[ Τοποθέτηση της συνάρτησης εκτύπωσης ]
store2 (ff printer)

[ Σε αυτήν την ετικέττα θα τοποθετηθεί ο κώδικας ]

label gen1 {

  label ff { }

}

}
```

Για παράδειγμα, για N=3 το παραπάνω πρόγραμμα εκτυπώνει στη οθόνη:

```
[george@localhost amk]$ ./amk 5.4.amk
input:
3
111
112
113
121
122
123
131
132
133
211
212
213
221
222
223
231
232
233
311
312
313
321
322
323
331
332
333
-----done-------
```

5.4 Εντολή goto

Η εντολή goto δεν υπάρχει στη γλώσσα αλλά μπορεί εύκολα να προσομοιωθεί με την κατάλληλη χρήση της ιδιάζουσας συμπεριφοράς της store, όταν τροποποιεί ετικέττα προγονό της.

Έστω το παρακάτω πρόγραμμα σε ψευδοκώδικα:

```
a = 1
b = 2
A:
write(a)
goto C:
B:
write(b)
goto END
C:
writeln
goto B
END
```

όπου οι ετικέττες είναι συμβολοσειρές που λήγουν στον χαρακτήρα ":". Το αντίστοιχο πρόγραμμα σε AMK είναι:

```
label goto {

  fetch (labela A)
  fetch (labelb B)
  fetch (labelc C)

  set a { 1 }
  set b { 2 }
  set null { }

  [ Κατασκευή του τμηματος από goto εντολές ]
  [ A + C + B ]
  := gotos null
  concat gotos labela labelc
  concat gotos gotos labelb

  [ Κυρίως πρόγραμμα ]
  label main {

    store (main gotos)

  }

  [ Διαγραφή των τμημάτων - δε χρειάζονται πια ]
  store (stuff null)

  [ Τα τμήματα κώδικα που θα κληθούν ]
  label stuff {

    label A {
      write (a)
    }

    label B {
```

```
      write (b)
    }

    label C {
      writeln
    }

  }

}
```

5.5 Η συνάρτηση του Ackermann

Η συνάρτηση του Ackermann συναντάται στη θεωρία υπολογισιμότητας και είναι ένα απλό παράδειγμα ανδρομικής συνάρτησης που δεν ακολουθεί το σχήμα της πρωταρχικής αναδρομής. Είναι συνάρτηση δυο μεταβλητών και χαρακτηρίζεται από το μεγάλο ρυθμό αύξησής της, αφού ακόμα και για μικρές τιμές της εισόδου της (π.χ. 4,3) η έξοδός της απαιτεί πολλούς υπολογισμούς.

Ορίζεται ως εξής:

```
function ack(m, n)
     if m = 0
         return n+1
     else if n = 0
         return ack(m-1, 1)
     else
         return ack(m-1, ack(m, n-1))
```

Η γλώσσα AMK δεν έχει υποστήριξη για συναρτήσεις και ο παραπάνω ψευδοκώδικας δείχνει αναδρομική κλήση που, ενώ στη δεύτερη περίπτωση έχει εξεταστεί σε παραπάνω παράδειγμα (αναδρομή ουράς), στην τρίτη είναι πιο δύσκολη. Για αυτόν το λόγο θα χρησιμοποιηθεί μια τμηματικά επαναληπτική (iterative) παραλλαγή της:

```
function ack(m, n)
     while m!=0
         if n = 0
             n := 1
         else
             n := ack(m, n-1)
         m := m - 1
     return n+1
```

Σε αυτήν την περίπτωση ο κώδικας δείχνει πιο εύκολα υλοποιήσιμος σε AMK αλλά λείπει πάλι ο ορισμός των συναρτήσεων από τη γλώσσα. Η υλοποίησή του μέσω AMK θα δοθεί στη συνέχεια.

Επίσης ενδιαφέρουσα θα ήταν μια προσπάθεια να υλοποιηθεί η συνάρτηση Ackermann ως μια συνάρτηση που ανάλογα με την τιμή της πρώτης παραμέτρου m δημιουργεί ένα πρόγραμμα από φωλιασμένους βρόχους for που είναι εξειδικευμένο μόνο για αυτήν την τιμή.

5.5.1 Προσομοίωση συναρτήσεων στην AMK

Ένα σώμα κώδικα μιας ετικέττας μπορεί να θεωρηθεί ως μια συνάρτηση που διαβάζει από το περιβάλλον της κάποιες μεταβλητές (ορίσματα της συνάρτησης) και επιστρέφει κάποια τιμή

σε κάποια μεταβλητή (τιμή της συνάρτησης). Για να είναι δυνατό μια συνάρτηση να είναι γενική, πρέπει α)να μπορεί να κληθεί και β)να είναι ανεξάρτητη του περιβάλλοντος στο οποίο λειτουργεί.

Το πρώτο είναι εύκολο, αν θεωρηθεί ότι η συνάρτηση καλείται με την τοποθέτηση του κώδικά της (που ήδη βρίσκεται σε μια μεταβλητή μέσω κάποιας fetch ή set) σε μια ετικέττα. Όταν έρθει η στιγμή να εκτελεστεί η ετικέττα, θα εκτελεστεί (κληθεί) η συνάρτηση.

Το δεύτερο απαιτεί έναν ξεχωριστό αποθηκευτικό χώρο που είναι προσβάσιμος μόνο στη συνάρτηση και δεν τροποποιείται από το υπόλοιπο πρόγραμμα. Επομένως θα αποκλειστούν οι μεταβλητές με όνομα (named variables) και θα γίνει χρήση του δυναμικού χώρου της ταινίας. Κατά αυτήν τη σύμβαση, πριν την κλήση της συνάρτησης, τοποθετούνται στο τέλος της ταινίας τα ορίσματά της. Στη συνέχεια η συνάρτηση εκτελούμενη τα διαβάζει και υπολογίζει την τιμή της, την οποία τοποθετεί επίσης στην ταινία. Όταν τελειώσει η εκτέλεσή της, το υπόλοιπο πρόγραμμα θα κοιτάξει στην ταινία για την τιμή της. Αυτό προϋποθέτει ότι υπάρχει κάποιος μετρητής env που αναφέρεται κάθε φορά στο τελευταίο κελί της ταινίας, ώστε να μπορεί κάθε συνάρτηση να τοποθετήσει τα δεδομένα της χωρίς τον κίνδυνο να τροποποιήσει ξένο περιεχόμενο.

5.5.2 Ο τελικός κώδικας

Παρακάτω δίνεται ο κώδικας που υλοποιεί τη συνάρτηση του Ackermann σε AMK:

```
label ackermannexample {

      [ Αρχικοποιεί σταθερές ]
      set zero { 0 }
      set one { 1 }
      set two { 2 }
      set env { 1 }

      [ Διαβάζει τους αριθμούς από το πληκτρολόγιο ]
      read(m)
      read(n)

      [ Διαβάζει τη συνάρτηση του Ackermann ]
      fetch (ackcode ack)
      [ Την προετοιμάζει να κληθεί ]
      store2 (ack ackcode)

      label test {

      [ Τοποθετεί τα m,n στις θέσεις 1 και 2 ]
      + env env one writev (env m)
      + env env one writev (env n)
      - env env two

      label ack {

            [ Μεταβλητή m ]
            + env env one readv (env m) - env env one
            [ Μεταβλητή n ]
            + env env two readv (env n) - env env two
            [ Εκτυπώνει στην οθόνη τις παραμέτρους της ]
```

```
write (m) writesp write (n) writeln
[ Υλοποιεί τις αναδρομικές κλήσεις στο σώμα ]
store2 (ack ackcode)

while m {
  [ Διαβάζει το n ]
  + env env two readv (env n) - env env two
  [ λεγχος του n ]
  if n {

     [ Μετακινείται στο επόμενο ζεύγος ]
     + env env two

     [ Τοποθετεί τα (m,n-1) ]
     + env env one writev (env m) - env env one
     - tmpn n one
     + env env two writev (env tmpn) - env env two

     vlabel ack { }

     [Παίρνει την τιμή του n από τη 2η θέση ]
     + env env two
     readv (env n)
     - env env two
     [ επιστρέφει στο προηγούμενο περιβάλλον ]
     - env env two
     [ Τοποθετεί τη νέα τιμή του n σε αυτό ]
     + env env two
     writev (env n)
     - env env two

}
else {

     [ Θέτει το n στη θέση 2 ίσο με 1 ]
     + env env two
     readv(env n)
     set n { 1 }
     writev(env n)
     - env env two

}

[ m = m - 1]
+ env env one
readv(env m)
- m m one
writev(env m)
- env env one

}
```

```
            [ Μεταβλητή n ]
            + env env two
            readv (env n)

            [ return n στην ταινία ως n = n + 1 ]
            + n n one
            writev (env n)
            - env env two

      }

      }

      [ Εμφάνιση στην οθόνη ]
      + env env two
      readv (env n)
      write (n)
}
```

Για παράδειγμα για τις τιμές (2, 8) η εκτέλεση της συνάρτησης είναι η εξής (τα ζεύγη αριθμών είναι οι κλήσεις με ενδιάμεσες τιμές):

```
[george@localhost amk]$ ./amk ackermann.amk
input:
2
input:
8
2 8
2 7
2 6
2 5
2 4
2 3
2 2
2 1
2 0
1 0
1 2
1 1
1 0
1 4
1 3
1 2
1 1
1 0
1 6
1 5
1 4
1 3
1 2
1 1
```

```
1 0
1 8
1 7
1 6
1 5
1 4
1 3
1 2
1 1
1 0
1 10
1 9
1 8
1 7
1 6
1 5
1 4
1 3
1 2
1 1
1 0
1 12
1 11
1 10
1 9
1 8
1 7
1 6
1 5
1 4
1 3
1 2
1 1
1 0
1 14
1 13
1 12
1 11
1 10
1 9
1 8
1 7
1 6
1 5
1 4
1 3
1 2
1 1
1 0
1 16
1 15
```

```
1 14
1 13
1 12
1 11
1 10
1 9
1 8
1 7
1 6
1 5
1 4
1 3
1 2
1 1
1 0
19
-----done-------
```

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

Η παρούσα διπλωματική εργασία έδειξε ότι η έννοια του αυτομεταβαλλόμενου κώδικα αν και λιγότερο μελετημένη σε σχέση με άλλες στρατηγικές συγγραφής αλγορίθμων, δεν υπολείπεται αυτών σε εκφραστικότητα και υπολογιστική ικανότητα. Είναι πλήρης κατά Turing και υπάρχουν αλγόριθμοι που αν υλοποιηθούν σε αυτόν είναι πιο φυσικοί και εύκολοι στην κατανόηση.

Η μέχρι τώρα εμπειρία από το σχεδιασμό γλωσσών προγραμματισμού τον έχει αγνοήσει, λόγω της στενής του σχέσης μέχρι πρόσφατα με γλώσσες χαμηλού επιπέδου και περιθωριακής χρήσης του σε ιδιάζουσες εφαρμογές (ιοί, προστασία) που δεν έδειχναν όλες τις ικανότητές του.

Με την κατασκευή της γλώσσας AMK και του διερμηνέα της έγινε φανερό ότι η παραπάνω άποψη είναι συντηρητική και μάλλον για προληπτικούς λόγους αποθαρρύνεται η χρήση του αυτομεταβαλλόμενου κώδικα. Είναι δυνατή η προσθήκη δυνατοτήτων αυτομεταβαλλόμενου κώδικα σε μια υπάρχουσα γλώσσα, όπως το προστακτικό υποσύνολο της AMK, χωρίς να διαταράσσεται η λειτουργία της και με διατήρηση της ασφάλειας που αυτή συνεπάγεται, δηλαδή τήρηση κανόνων που εγγυώνται τη συνέπεια του κώδικα και την προβλέψιμη συμπεριφορά του.

6.2 Μελλοντική έρευνα

Μια κατεύθυνση στην οποία θα μπορούσε να οδηγήσει το παραπάνω είναι η προσπάθεια να προστεθούν τέτοια χαρακτηριστικά σε μια πιο προχωρημένη γλώσσα. Με αυτόν τον τρόπο θα φανεί η διαφορά ανάμεσα στα πραγματικά μειονεκτήματα του αυτομεταβαλλόμενου κώδικα ως πρακτικής και στα ψευδοπροβλήματα που οφείλονται σε λανθασμένες υλοποιήσεις, προγραμματισμό χαμηλού επιπέδου και αγκύλωση σε τρόπους συγγραφής που ανήκουν στο παρελθόν (π.χ. αυτομεταβαλλόμενοι ιοι).

Επίσης το παραπάνω μοντέλο αυτομεταβαλλόμενου κώδικα αξίζει να μελετηθεί θεωρητικά για να βρεθούν οι αντιστοιχίες του με τα υπόλοιπα υπολογιστικά μοντέλα και τα χαρακτηριστικά του που μπορούν να τον κάνουν πιο ελκυστικό έναντι αυτών.

Τέλος, αξίζει να αναφερθεί η περίπτωση του αυτομεταβαλλόμενου ή αυτορρυθμιζόμενου υλικού (self-configurable hardware) που αποτελεί νέα τάση στην κατασκευή υλικού, αποτελούμενο από συστοιχίες πυλών και άλλων ηλεκτρονικών που δυναμικά προσαρμόζονται ανάλογα με τις ανάγκες του εκάστοτε προβλήματος(υπολογιστική ισχύς, κατανάλωση ενέργειας). Ο αυτομεταβαλλόμενος κώδικας μοιάζει ιδανικό συμπλήρωμα ενός τέτοιου συστήματος και αξίζει να μελετηθεί η μεταξύ τους σχέση.

Βιβλιογραφία

[1] E.W.D. Dijkstra, "Computing Science: Achievements and Challenges", ACM Symposium on Applied Computing at San Antonio, TX, March 1999.

[2] Goldberg D. E., "GENETIC ALGORITHMS in Search, Optimization and Machine Learning", Addison Wesley Publishing Company, Inc., 1989.

[3] Thomas Grill, "dyn~ dynamic object management", http://www.parasitaere-kapazitaeten.net/ext/dyn/.

[4] Gaurav S. Kc Computer Science Dept. Columbia University, Angelos D. Keromytis Computer Science Dept. Columbia University, Vassilis Prevelakis Computer Science Dept. Drexel University, "Countering Code-Injection Attacks With Instruction-Set Randomization'.'

[5] Peter Lee, Mark Leone, "Optimizing ML with Run-Time Code Generation", School of Computer Science, Carnegie Mellon University.

[6] Erik Meijer and Peter Drayton, "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages", Microsoft Corporation.

[7] Michalewich Z., "Genetic Algorithms + Data Structures = Evolution Programs", Springer-Verlag, 2nd ed., 1992.

[8] Peter Nordin and Wolfgang Banzhaf "Evolving turing-complete programs for a register machine with self-modifying code", i. In L. Eshelman, editor, Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), pages 318 325, Pittsburgh, PA, USA, 15-19 1995, Morgan Kaufmann.

[9] Calton Pu, Henry Massalin and John Ioannidis, "The Synthesis Kernel", Department of Computer Scence, Columbia University, PhD.

[10] Nicolas Roard, "Camaelon Theme Engine", www.roard.com/camaelon.

[11] Roy S. Rubinstein, John N. Shutt, "Self-Modifying Finite Automata", Computer Science Technical Report Series, WPI-CS-TR-95-2, August 1995.

[12] Sriram Srinivasan, "Advanced Perl Programming", ISBN 1-56592-220-4, O'Reilly.

[13] Unsanity, "Application Enhancer", www.unsanity.net/haxies/ape.

[14] Ορέστης Αλεβίζος, "Μεταγλώττιση αυτομεταβαλλόμενου κώδικα υψηλού επιπέδου", διπλωματική εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχ. και Μηχ. Υπολογιστών, Οκτώβριος 2005.

[15] Νικόλαος Σ. Παπασπύρου, Εμμανουήλ Σ. Σκορδαλάκης, "Μεταγλωττιστές", εκδόσεις Συμμετρία.

[16] Institut National de Recherche en Informatique et en Automatique, http://caml.inria.fr.

[17] http://www.gnustep.org.

[18] http://www.haskell.org.