ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΜΕΤΑΓΛΩΤΤΙΣΗ ΑΥΤΟΜΕΤΑΒΑΛΛΟΜΕΝΟΥ ΚΩΔΙΚΑ ΥΨΗΛΟΥ ΕΠΙΠΕΔΟΥ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ορέστης Η. Αλεβίζος

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Λέκτορας Ε. Μ. Π.

Αθήνα, Οκτώβριος 2005

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΜΕΤΑΓΛΩΤΤΙΣΗ ΑΥΤΟΜΕΤΑΒΑΛΛΟΜΕΝΟΥ ΚΩΔΙΚΑ ΥΨΗΛΟΥ ΕΠΙΠΕΔΟΥ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ορέστης Η. Αλεβίζος

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Λέκτορας Ε. Μ. Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24η Οκτωβρίου 2005.

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

Αθήνα, Οκτώβριος 2005

Περίληψη

Το αντικείμενο της εργασίας αυτής ήταν η σχεδίαση και η υλοποίηση μιας γλώσσας προγραμματισμού υψηλού επιπέδου που ενσωματώνει χαρακτηριστικά αυτομεταβαλλόμενου κώδικα διατηρώντας το επίπεδο ασφάλειας τύπων μιας κλασσικής γλώσσας προγραμματισμού. Η υλοποίηση έγινε με τη μορφή μεταγλωττιστή για αυτή τη γλώσσα.

Συγκεκριμένα σχεδιάσθηκε η πειραματική γλώσσα SMEC η οποία βασίζεται σε υποσύνολο της ANSI C και ακολουθεί το συντακτικό της γλώσσας C αλλά παραλλήλως έχει δυνατότητες κατασκευής, ανάλυσης και εκτέλεσης κώδικα στο χρόνο εκτέλεσης. Ο παραπάνω κώδικας, που ονομάζουμε δυναμικό κώδικα, δημιουργείται με συντακτικούς όρους υψηλού επιπέδου και ελέγχεται σημασιολογικά κατά τη μεταγλώττιση του προγράμματος με τη βοήθεια ενός ασφαλούς συστήματος τύπων. Έτσι, παρέχεται το ίδιο επίπεδο ασφάλειας με τις κλασσικές γλώσσες υψηλού επιπέδου (συγκεκριμένα με τη γλώσσα C) χωρίς κόστος στο χρόνο εκτέλεσης.

Υλοποιήθηκε ένας μεταγλωττιστής για τη γλώσσα SMEC που υλοποιεί τα παραπάνω. Οι δυναμικές επεκτάσεις της γλώσσας υλοποιήθηκαν έτσι ώστε να έχουν όσο το δυνατόν μικρότερο κόστος σε χρόνο και πράγματι επετεύχθησαν μικρά κόστη. Ο μεταγλωττιστής κατά τα άλλα έγινε προσπάθεια να κατασκευαστεί όσο το δυνατόν απλούστερος.

Οι τεχνικές που χρησιμοποιήθηκαν για την εισαγωγή και υλοποίηση των δυναμικών επεκτάσεων στη γλώσσα SMEC είναι αρκετά γενικές ώστε να μπορούν να χρησιμοποιηθούν για τον εμπλουτισμό των σημερινών γλωσσών προγραμματισμού με παρόμοια στοιχεία.

Λέξεις Κλειδιά

Αυτομεταβαλλόμενος κώδικας υψηλού επιπέδου, μεταγλωττιστές, κατασκευή και ανάλυση δυναμικού κώδικα.

Abstract

The goal of this diploma dissertation was the definition and implementation of a high-level programming language that would embed characteristics of self-modifying code, while preserving the level of type safety of well known high-level programming languages. The implementation was made in the form of a compiler.

Indeed, a simple experimental language (SMEC) was defined, based on a subset of ANSI C, following its syntax. At the same time the aforementioned language has the ability to construct, analyze and execute code at runtime. This kind of code, which we refer to herein as "dynamic code", is created with high-level syntactic terms and is semantically checked at compile-time using a strong type system. Thus, the dynamic code produced has a safety level equivalent to that of conventional high-level languages (for this matter C) at no cost at runtime.

A compiler was developed for SMEC that implements the tasks mentioned above. The conventional part of the compiler was kept as simple as possible. Great effort was made so that the dynamic extensions would have the minimum runtime cost (indeed, fairly small costs were achieved).

The techniques employed for the embedding and implementation of SMEC's dynamic extensions are general enough so that they can be used to enhance modern high-level languages with similar capabilities.

Keywords

High-level self modifying code, compilers, construction and analysis of dynamic code.

Ευχαριστίες

Με την ευκαιρία αυτή θα ήθελα να ευχαριστήσω το Εθνικό Μετσόβιο Πολυτεχνείο, τους διδάσκοντες και τους συναδέλφους μου. Ιδιαιτέρως ευχαριστώ τον επιβλέποντα καθηγητή της διπλωματικής αυτής κ. Νίκο Παπασπύρου για την πρόθυμη και ουσιαστική βοήθειά του στην εκπόνηση αυτής της εργασίας.

Ορέστης Η. Αλεβίζος,

Αθήνα, 19η Σεπτεμβρίου 2005.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-6-05, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2005.

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

Πίνακας Περιεχομένων

Περίληψη 5
Abstract 7
Ευχαριστίες 9
Πίνακας Περιεχομένων 11
Ευρετήριο Πινάκων και Εικόνων 15
1. Εισαγωγή 17
1.1. Σκοπός της Έργασίας 17
1.2. Αυτομεταβαλλόμενος Κώδικας Υψηλού Επιπέδου 17
1.2.1. Τι είναι ο αυτομεταβαλλόμενος κώδικας 17
1.2.2. Αυτομεταβαλλόμενος Κώδικας Υψηλού Επιπέδου 18
1.2.3. Υπολογιστική Ισχύς Γλώσσας Αυτομεταβαλλόμενου Προγράμματος 19
1.3. Σύνοψη της Εργασίας 20
2. Η Γλώσσα SMEC 21
2.1. Γενικά 21
2.2. Στατικός – Δυναμικός Κώδικας 22
2.3. Τύποι – Ο τύπος δυναμικού κώδικα codeof 22
2.4. Λειτουργίες Δυναμικού Κώδικα 23
2.4.1. Γενικά 23
2.4.2. Κατασκευή 23
2.4.3. Ανάλυση – Κατακερματισμός 23
2.4.4. Εκτέλεση 24
2.4.5. Πολυπλοκότητα Υλοποίησης Λειτουργιών 24
2.5. Συντακτικό Της Γλώσσας 24
2.6. Ανάλυση Γλωσσικών Δομών Κατασκευής Δυναμικού Κώδικα 26
2.6.1. Γενικά 26
2.6.2. Το Δυναμικό Σύμβολο # 26
2.6.3. Δεδομένα Δυναμικού Κώδικα 26
2.6.4. Δυναμικοί Τελεστές Αντίστοιχοι Στατικών 28
2.6.5. Δυναμικοί Τελεστές Ανάθεσης 29
2.6.6. Το σύμβολο #; - Ο τελεστής #; 29
2.6.7. Προτεραιότητα Τελεστών 30
2.6.8. Στατικοί Τελεστές Σε Αντικείμενα Τύπου CODEOF 30
2.6.9. #IF - #IF … #ELSE 30
2.6.10. #FOR 30
2.6.11. Δυναμική Κλήση Συνάρτησης 31
2.7. Εντολές Ανάλυσης - Εκτέλεσης Δυναμικού Κώδικα 32
2.7.1. Γενικά 32
2.7.2. match 32
2.7.3. typematch 33
2.7.4. exec 33
2.8. Διαχείριση Μνήμης 34
3. Ο Μεταγλωττιστής 35
3.1. Γενικά 35
3.2. Απόδοση 36
3.3. Διάρθρωση 36
3.3.1. Τμήματα 36

3.3.2. Αρχεία - Κλάσεις ... 37
3.4. Δυναμικός Κώδικας ... 39
3.4.1. Γενικά ... 39
3.4.2. Δυναμικά Δεδομένα ... 39
3.4.3. Παράσταση Δυναμικού Κώδικα - Αντικείμενο Δυναμικού Κώδικα (Dynamic Code Object) ... 40
3.4.4. Γενικές Αρχές Κατασκευής Δυναμικού Κώδικα ... 40
3.4.5. Λεπτομέρειες Κατασκευής Δυναμικού Κώδικα – Πρότυπα ... 42
3.4.6. Εκτέλεση Δυναμικού Κώδικα ... 44
3.4.7. Ανάλυση – Αποδόμηση Δυναμικού Κώδικα ... 45
3.4.8. Διαχείριση των ΑΔΚ - Καταστροφή ... 48
3.5. Σημασιολογική Ανάλυση ... 48
3.5.1. Γενικά ... 48
3.5.2. Σημασιολογική Ανάλυση Στατικού Κώδικα ... 48
3.5.3. Σημασιολογική Ανάλυση Δυναμικού Κώδικα ... 49
3.5.4. Σύστημα Τύπων – Εσωτερικοί δείκτες ... 50
3.5.5. Δομές match - typematch ... 51
3.5.6. DL-Values ... 52
3.5.7. Πίνακας Συμβόλων ... 52
3.6. Παραγωγή Ενδιάμεσου Κώδικα ... 53
3.6.1. Γενικά ... 53
3.6.2. Ενδιάμεση γλώσσα ... 53
3.6.3. Αναπαράσταση Τετράδων - Ενοτήτων ... 55
3.6.4. Σημασιολογικές Ρουτίνες Παραγωγής Ενδιάμεσου Κώδικα ... 56
3.6.5. Συναρτήσεις Βιβλιοθήκης – Υποστήριξη Ανάστροφου Περάσματος Παραμέτρων ... 56
3.6.5.1. makecode ... 57
3.6.5.2. addref ... 57
3.6.5.3. gettype ... 57
3.6.5.4. getdeps ... 58
3.6.5.5. destroy ... 58
3.6.5.6. new - dispose ... 58
3.6.6. Παραγωγή Ενδιάμεσου Κώδικα Για Τις Δυναμικές Δομές Υψηλού Επιπέδου - Γενικά 58
3.6.7. Παραγωγή Ενδιάμεσου Κώδικα ΠΔΚ - Γενικά ... 59
3.6.8. Πλήρες Σχέδιο Παραγωγής Ενδιάμεσου Κώδικα ΠΔΚ ... 61
3.6.8.1. Σύμβολο #; (εντολή skip) ... 61
3.6.8.2. Τελεστής #; (δύο ορισμάτων) ... 61
3.6.8.3. Δυναμική Σταθερά ... 62
3.6.8.4. Δυναμική Μεταβλητή ... 62
3.6.8.5. Δομή IF ... 63
3.6.8.6. Δομή IF-ELSE ... 63
3.6.8.7. Δομή FOR ... 64
3.6.9. Κώδικας Κατασκευής ΑΔΚ ... 64
3.6.10. Εκτέλεση Δυναμικού Κώδικα ... 66
3.6.11. Ανάλυση – Δομή match ... 66
3.6.12. Ανάλυση – Δομή typematch ... 69
3.6.13. Καταστροφή ... 70
3.7. Παραγωγή Τελικού Κώδικα ... 70
3.7.1. Γενικά ... 70
3.7.2. Μηχανή – Στόχος ... 70

3.7.3. Δυναμικές Μεταβλητές ... 71
3.7.4. Οργάνωση Μνήμης - Εγγραφήματα Δραστηριοποίησης – Μοντέλο Κλήσης.. 71
3.7.5. Διαφορές Συναρτήσεων – ΑΔΚ ... 73
3.7.6. Βοηθητικές Συναρτήσεις Γεννήτορα Τελικού Κώδικα ... 74
3.7.7. Παράσταση Εντολών, CFinalCode, SFinalCmd. SFinalOp ... 74
3.7.8. Κατασκευή ΠΔΚ ... 75
3.7.9. Η κλάση CPattern - Σχισμές ... 78
3.7.10. Παραγωγή Τελικού Κώδικα για Δυναμικές Τετράδες ... 79
3.7.11. Βιβλιοθήκη Χρόνου Εκτέλεσης ... 80
3.8. Τρόπος Κλήσης του Μεταγλωττιστή ... 81
4. Παραδείγματα ... 83
4.1. Γενικά ... 83
4.2. Εκτύπωση Αριθμών Ν Ψηφίων ... 84
4.2.1. Περιγραφή ... 84
4.2.2. Κώδικας ... 84
4.2.3. Ενδιάμεσος Κώδικας Προγράμματος ... 84
4.2.4. Ενδιάμεσος Κώδικας ΠΔΚ ... 86
4.2.5. Τελικός Κώδικας Προγράμματος ... 87
4.2.6. Παράδειγμα Εκτέλεσης ... 101
4.3. Αναδρομική Ανάλυση Κώδικα ... 102
4.3.1. Περιγραφή ... 102
4.3.2. Κώδικας ... 102
4.3.3. Ενδιάμεσος Κώδικας Προγράμματος ... 104
4.3.4. Ενδιάμεσος Κώδικας ΠΔΚ ... 107
4.3.5. Τελικός Κώδικας Προγράμματος ... 107
4.3.6. Παράδειγμα Εκτέλεσης ... 107
5. Συμπεράσματα ... 109
5.1. Συνεισφορά ... 109
5.2. Μελλοντικές Επεκτάσεις ... 110
Βιβλιογραφία ... 111

Ευρετήριο Πινάκων και Εικόνων

Πίνακας 3-1 39
Πίνακας 3-2 40
Πίνακας 3-3 43
Πίνακας 3-4 43
Πίνακας 3-5 50
Πίνακας 3-6 55
Πίνακας 3-7 56
Πίνακας 3-8 69

Π 3-1 62
Π 3-2 63
Π 3-3 63
Π 3-4 64
Π 3-5 64

1. Εισαγωγή

1.1. Σκοπός της Έργασίας

Σκοπός της παρούσας εργασίας είναι η σχεδίαση και υλοποίηση μιας γλώσσας προγραμματισμού υψηλού επιπέδου με τη δυνατότητα να κατασκευάζει εκτελέσιμο κώδικα στο χρόνο εκτέλεσης με τη βοήθεια γλωσσικών δομών υψηλού επιπέδου, να εκτελεί τον κώδικα αυτόν και να τον αναλύει στα συστατικά του. Η γλώσσα αυτή θα πρέπει να διαθέτει σύστημα τύπων που να μπορεί να βεβαιώσει ότι κάθε πρόγραμμα που θα παραχθεί από αυτή δεν θα προκαλέσει σφάλμα λόγω ασυμφωνίας τύπων. Επίσης η γλώσσα θα είναι σύμφωνη με τη φιλοσοφία του δομημένου προγραμματισμού.

1.2. Αυτομεταβαλλόμενος Κώδικας Υψηλού Επιπέδου

1.2.1. Τι είναι ο αυτομεταβαλλόμενος κώδικας

Ως αυτομεταβαλλόμενο κώδικα θεωρούμε:

Τον κώδικα ο οποίος αν εκτελεστεί θα εκτελέσει αλλαγές στον

εαυτό του ή θα δημιουργήσει νέα τμήματα κώδικα τα οποία και θα εκτελέσει.

Για να έχουμε μια ιδέα ας θεωρήσουμε τον παρακάτω κώδικα ψευδο-assembly:

```
data1:  mov   ax,3
data2:  mov   cx,2
        add   ax,cx
        ret
...
        mov   word ptr [offset data1+1],7
        mov   word ptr [offset data2+1],10
        call  data1
```

Έχουμε εδώ μια διαδικασία η οποία εκτελεί την πρόσθεση του ax, με τον cx αφού έχει τοποθετήσει στον ax το 3 και στον cx το 2. Ο παραπάνω κώδικας αλλάζει τη "διαδικασία" στο data 1 βάζοντας το 7 στη θέση του 3 και το 10 στη θέση του 2 και κατόπι καλεί τη διαδικασία.

Η προϋπόθεση που πρέπει να πληρεί μια μηχανή (πραγματική ή ιδεατή) είναι ο κώδικας και τα δεδομένα να αποθηκεύονται σε κοινή μνήμη στην οποία να επιτρέπονται εγγραφές. Από αυτή τη σκοπιά ο αυτομεταβαλλόμενος κώδικας προβλέπεται από το αφηρημένο μοντέλο της μηχανής Turing όσο και από το (πραγματικό) μοντέλο von Neuman. Παρ' όλα αυτά η έννοια θεωρήθηκε ούτως η άλλως επισφαλής σε σημείο που ο Dijkstra (EWD1284) αναφέρει ότι η έννοια του αυτομεταβαλλόμενου κώδικα αποδείχθηκε πως ήταν ένα αστείο υπεύθυνο για μια δεκαετία σύγχυσης.

1.2.2. Αυτομεταβαλλόμενος Κώδικας Υψηλού Επιπέδου

Σε επίπεδο γλώσσας μηχανής ή συμβολικής γλώσσας είναι πολύ εύκολο να δημιουργήσουμε προγράμματα αυτομεταβαλλόμενου κώδικα όπως π.χ. οι ιοί[1], αφού κώδικας και δεδομένα αποθηκεύονται στη μνήμη.

Η αρχική ιδέα της παρούσας εργασίας είναι να δούμε κατά πόσο μπορούμε να εκτελέσουμε παρόμοιες λειτουργίες στα πλαίσια μιας γλώσσας υψηλού επιπέδου. Όλες οι γνωστές γλώσσες υψηλού επιπέδου διαχωρίζουν αυστηρά τον κώδικα από τα δεδομένα και δεν επιτρέπεται η αλλαγή του υπάρχοντος κώδικα ή η δημιουργία καινούριου.

Η μη υιοθέτηση του αυτομεταβαλλόμενου κώδικα σε γλώσσες υψηλού επιπέδου έχει την αιτία της στα προβλήματα που ανακύπτουν με την προσπάθεια εναρμόνισης

[1] Οι ιοί στην πραγματικότητα αλλάζουν κώδικα άλλων προγραμμάτων και αντιγράφονται εκεί. Παρόλα αυτά πολλοί ιοί "μεταλλάσονται" δηλαδή αλλάζουν τον κώδικά τους για να "ξεγελάσουν" τα προγράμματα εντοπισμού ιών που ψάχνουν για αυτόν ακριβώς τον κώδικα στα εκτελέσιμα αρχεία.

του με την ασφάλεια τύπων και το δομημένο κώδικα. Ο στόχος αυτής της εργασίας είναι η κατασκευή μιας γλώσσας η οποία θα περιλαμβάνει λειτουργίες αυτομεταβαλλόμενου κώδικα και ταυτόχρονα θα παρέχει όλα τα άλλα χαρακτηριστικά και προπάντων την ασφάλεια των κλασσικών γλωσσών προγραμματισμού. Δηλαδή θα πρέπει να βρούμε έναν τρόπο ώστε να εξασφαλίσουμε από το στάδιο της μεταγλώττισης ότι ο κώδικας που θα δημιουργηθεί η θα αλλαχθεί κατά το χρόνο εκτέλεσης να μας παρέχει τις ίδιες εγγυήσεις ασφάλειας που έχουμε και από τον κώδικα που παράγεται από ένα κλασσικό μεταγλωττιστή.

Η παραπάνω γλώσσα θα πρέπει να παρέχει δυνατότητα κατασκευής κώδικα σε χρόνο εκτέλεσης (τουλάχιστο). Για να συνεχίσουμε να έχουμε συνέπεια τύπων θα πρέπει το υποσύνολο της γλώσσας που θα δημιουργήσουμε για να παράγουμε αυτόν τον κώδικα να παρέχει ασφάλεια τύπων. Έτσι:

> **Ως γλώσσα αυτομεταβαλλόμενου κώδικα υψηλού επιπέδου θεωρούμε τη γλώσσα εκείνη η οποία θα μας επιτρέψει να δημιουργούμε στο χρόνο εκτέλεσης κώδικα ο οποίος θα υλοποιεί δομές υψηλού επιπέδου της γλώσσας οι οποίες θα ορίζονται στο χρόνο εκτέλεσης και ταυτόχρονα ο κώδικας αυτός θα είναι ασφαλής από άποψη τύπων (type safe).**

Για να γίνει κατανοητός ο παραπάνω ορισμός θέλουμε μια γλώσσα η οποία θα μπορεί να φέρει εις πέρας ασφαλώς εντολές σαν τις παρακάτω:

"Κατασκεύασε *n* φωλιασμένα for-loops το ένα ως σώμα του άλλου έτσι ώστε το καθένα εκτελεί το σώμα του 5 φορές και το τελευταίο να τυπώνει τον αριθμό 1, όπου *n* αριθμός δοσμένος από το χρήστη"

"Δεδομένου κώδικα c_1 να βρεθεί αν αυτός υλοποιεί δομή if και αν ναι να αλλαχθεί η συνθήκη του if με τη συνθήκη c_2."

Τα παραπάνω βέβαια παραδείγματα είναι απλοϊκά και δεν έχουν καμία χρησιμότητα αλλά δίνουν μια εικόνα των δυνατοτήτων μιας τέτοιας γλώσσας.

Το πώς κατασκευάσαμε αυτή τη γλώσσα αναλύεται στο κεφάλαιο 2.

1.2.3. Υπολογιστική Ισχύς Γλώσσας Αυτομεταβαλλόμενου Προγράμματος

Εδώ θα εξετάσουμε αν και κατά πόσο η υπολογιστική ισχύς μιας γλώσσας αυτομεταβαλλόμενου κώδικα είναι μεγαλύτερη από αυτή μιας κλασσικής γλώσσας προγραμματισμου.

Μπορούμε να πούμε ότι μια γλώσσα αυτομεταβαλλόμενου κώδικας έχει την ίδια ισχύ με τις κλασσικές γλώσσες καθώς έχει δειχθεί ότι το υπολογιστικό μοντέλο του

Von Neuman (που υποστηρίζει αυτομεταβαλλόμενο) είναι ισοδύναμο με τις κλασσικές γλώσσες προγραμματισμού (Pascal, C, …) με την έννοια ότι για κάθε πρόβλημα P, αν το P λύνεται σε πολυωνυμικό χρόνο από το ένα θα λύνεται επίσης σε πολυωνυμικό χρόνο από το άλλο. Δηλαδή σίγουρα ο αυτομεταβαλλόμενος κώδικας δεν θα αποτελέσει λύση για τα προβλήματα που δεν λύνονται σε πολυωνυμικό χρόνο.

Παρ' όλα αυτά δεν μπορέσαμε να βρούμε θετική απόδειξη ότι με χρήση αυτομέταβαλόμενου κώδικα μπορούμε να κατασκευάσουμε προγράμματα σημαντικά αποδοτικότερα για γνωστα θεωρητικά προβλήματα. Εκεί που σίγουρα ο αυτομεταβαλλόμενος κώδικας υπερτερεί είναι στο ότι μπορεί κανείς να κατασκευάσει έυκολα λύσεις για προβλήματα για τα οποία θα χρειαζόταν να καταβάλει πολύ μεγαλύτερη προσπάθεια με μια κλασσική γλώσσα προγραμματισμου.

1.3. Σύνοψη της Εργασίας

Παρακάτω θα παρουσιουσιάσουμε τόσο τη γλώσσα όσο και την υλοποιήσή της σε μορφή μεταγλωττιστή.

Στο κεφάλαιο 2 θα περιγραφεί η γλώσσα συντακτικά και σημασιολογικά.

Στο κεφάλαιο 3 θα περιγραφεί ο μεταγλωττιστής που κατασκευάστηκε για τη γλώσσα. Η περιγραφή αυτή θα γίνει και ως προς τον τρόπο παράστασης των δυναμικών στοιχείων, ο οποίος είναι γενικός και θα μπορούσε να εφαρμοστεί σε κάθε μεταγλωττιστή που θέλει να περιλάβει ανάλογες δυνατότητες, καθώς και ο συγκεκριμένος μεταγλωττιστής.

Στο κεφάλαιο 4 θα δοθούν παραδείγματα προγραμμάτων της γλώσσας καθώς και ο ενδιάμεσος και τελικός κώδικας των συγκεκριμένων προγραμμάτων.

Στο κεφάλαιο 5 θα παρουσιασθεί η συνεισφορά της συγκεκριμένης εργασίας στην έρευνα καθώς και η μελλοντική εργασία που μπορεί να γίνει στον τομέα.

Σημειώνουμε ότι θα δοθεί βάρος στις νέες επεκτάσεις και όχι στα στοιχεία που έχει η γλώσσα και είναι γνωστά από τις κλασσικές γλώσσες τόσο σε επίπεδο περιγραφής της γλώσσας όσο και σε επίπεδο υλοποίησης. Για πληροφορίες πάνω στις προστακτικές γλώσσες, τη σημασιολογία τους και την υλοποίησή τους θα παραπέμπαμε τον αναγνώστη στη βιβλιογραφία.

2. Η Γλώσσα SMEC

2.1. Γενικά

Όπως προαναφέρθηκε ο στόχος της παρούσας εργασίας είναι να δημιουργηθεί μια γλώσσα αυτομεταβαλλόμενου κώδικα με δομές υψηλού επιπέδου η οποία να δίνει στον προγραμματιστή επίπεδο ασφάλειας αντίστοιχο με μιας συμβατικής γλώσσας.

Με αυτόν το στόχο έπρεπε να επιλεγεί μια πολύ μικρή γλώσσα εύκολη σε υλοποίηση στην οποία θα προστίθεντο δυναμικές επεκτάσεις. Η στατική γλώσσα από την οποία ξεκινήσαμε είναι ένα μικρό υποσύνολο της C.

Η γλώσσα SMEC (Self Modifying Elementary C) δημιουργήθηκε εμπλουτίζοντας αυτή τη γλώσσα με δυναμικές επεκτάσεις. Το συντακτικό της γλώσσας ταυτίζεται με αυτό της γλώσσας C όσον αφορά το κλασικό τμήμα και έτσι θα είναι εύκολο να το κατανοήσει κάποιος με μικρή εξοικείωση με τη γλώσσα C. Οι λεκτικές μονάδες και η σημασία των κλασικών δομών είναι ίδιες με τη γλώσσα C και στα επόμενα δεν θα

επεκταθούμε σε αυτές καθώς κάτι τέτοιο θα αποτελούσε άσκοπη επανάληψη.

2.2. Στατικός – Δυναμικός Κώδικας

Στη γλώσσα SMEC υπάρχουν δύο είδη κώδικα:

- Ο κλασσικός κώδικας που δημιουργείται χρησιμοποιώντας της κλασικές δομές υψηλού επιπέδου της γλώσσας C που έχει η γλώσσα και μεταγλωττίζεται κατά τα γνωστά στο χρόνο μεταγλώττισης. Σε αυτόν τον κώδικα δεν επιτρέπονται αλλαγές ή ανάλυση. Ο κώδικας αυτός ονομάζεται "στατικός".
- Ο κώδικας που δημιουργείται με χρήση των επεκτάσεων της γλώσσας, δημιουργείται δυναμικά και η κατασκευή του γίνεται κατά το χρόνο εκτέλεσης. Ο κώδικας αυτός ονομάζεται "δυναμικός", μπορεί να χρησιμοποιηθεί σαν δεδομένο και με χρήση των παραπάνω επεκτάσεων επιτρέπει να εκτελεστούν οι εξής λειτουργίες:
    - Σύνθεση - Κατασκευή
    - Εκτέλεση
    - Ανάλυση – Κατακερματισμός

Περισσότερα για της λειτουργίες αυτές και πως επιτελούνται από τις δομές της γλώσσας θα δούμε στα επόμενα.

Σε όλο το κείμενο οι όροι του "στατικού" και του "δυναμικού" έχουν αυτή την έννοια (π.χ. στατικός κώδικας, δυναμικές δομές υψηλού επιπέδου).

2.3. Τύποι – Ο τύπος δυναμικού κώδικα codeof

Κατ' αναλογία με τα παραπάνω οι τύποι της γλώσσας χωρίζονται σε στατικούς και δυναμικούς.

Οι στατικοί τύποι που υποστηρίζει η γλώσσα είναι οι εξής:

- Ο ακέραιος (int)
- Ο λογικός (bool)
- Ο τύπος χαρακτήρα (char)

Οι δυναμικοί τύποι ορίζονται ως:

```
codeof <static>
```

όπου static ένας στατικός τύπος ή ο κενός τύπος (void). Ένας τέτοιος δυναμικός τύπος παριστάνει κώδικα ο οποίος αν εκτελεστεί θα μας επιστρέψει μια τιμή τύπου static. Θα

μπορούσαμε να πούμε ότι ο τύπος αυτός παριστάνει μια συνάρτηση χωρίς παραμέτρους που επιστρέφει static και όπως θα δούμε έτσι ακριβώς υλοποιείται. Σημειώνουμε ότι ο τύπος void θεωρείται γενικός τύπος δηλαδή οποιοσδήποτε τύπος codeof μπορεί να θεωρηθεί τύπος codeof void. Αυτό έχει τη σημασία ότι μπορούμε να χρησιμοποιήσουμε οποιαδήποτε συνάρτηση χωρίς να ενδιαφερόμαστε για το αποτέλεσμα.

2.4. Λειτουργίες Δυναμικού Κώδικα

2.4.1. Γενικά

Ας θεωρήσουμε ότι έχουμε κώδικα υψηλού επιπέδου ως τύπο δεδομένων ποιες είναι οι λειτουργίες που θέλουμε να υλοποιήσουμε με αυτόν τον τύπο. Η γλώσσα SMEC υποστηρίζει τις λειτουργίες που αναφέρθηκαν παραπάνω. Εδώ η σημασία αυτών των λειτουργιών θα αποσαφηνισθεί.

2.4.2. Κατασκευή

Πρέπει να υπάρχουν δομές οι οποίες θα κατασκευάζουν αντικείμενα δυναμικού κώδικα με όρους υψηλού επιπέδου στο χρόνο εκτέλεσης, δηλαδή η δομή των αντικειμένων θα καθορίζεται από δεδομένα άγνωστα στο χρόνο μεταγλώττισης. Παρ' όλα αυτά πρέπει να είμαστε σίγουροι ότι ο κώδικας που θα κατασκευαστεί θα είναι τόσο ασφαλής όσο και αν τον είχαμε γράψει ως κανονικό στατικό. Ένα παράδειγμα λειτουργίας κατασκευής διατυπωμένης σε φυσική γλώσσα είναι:

"Κατασκεύασε *n* φωλιασμένα for-loops το ένα ως σώμα του άλλου έτσι ώστε το καθένα εκτελεί το σώμα του 5 φορές και το τελευταίο να τυπώνει τον αριθμό 1, όπου *n* αριθμός δοσμένος από το χρήστη"

2.4.3. Ανάλυση – Κατακερματισμός

Η λειτουργία της ανάλυση μας είναι η ανάλυση δυναμικού κώδικα υψηλού επιπέδου στα επιμέρους στοιχεία που τον αποτελούν. Η ανάλυση αυτή ακολουθεί τη δομή του συντακτικού δέντρου. πχ. μια δυναμική εντολή if θα αναλυόταν στη συνθήκη και το σώμα της. Χρησιμοποιώντας την αναδρομή μπορούμε να αναλύουμε τον κώδικα μέχρι τα βασικά στοιχεία από τα οποία αυτός αποτελείται. Αυτό μας δίνει και τη δυνατότητα να χειριζόμαστε τον κώδικα. Πχ. με συνδυασμό της ανάλυσης και σύνθεσης θα μπορούσαμε να επιτύχουμε την ακόλουθη λειτουργία:

"Δεδομένου κώδικα if c κατασκευάστε κώδικα for ο οποίος θα έχει τη συνθήκη του if και κενές εντολές επανάληψης και αρχικοποίησης"

2.4.4. Εκτέλεση

Ο δυναμικός κώδικας θα πρέπει φυσικά να μπορεί να εκτελείται κατά βούληση και όσες φορές χρειαστεί. Ο κώδικας εκτελείται και επιστρέφει το αποτέλεσμα το οποίο είναι τύπου T αν ο κώδικας είναι τύπου codeof T.

2.4.5. Πολυπλοκότητα Υλοποίησης Λειτουργιών

Είναι φανερό ότι κατά την υλοποίηση της γλώσσας πρέπει να δοθεί προσοχή ώστε η πολυπλοκότητα υλοποίησης των παραπάνω λειτουργιών να κρατηθεί σε χαμηλά επίπεδα. Ειδικά η λειτουργία της εκτέλεσης πρέπει να γίνεται σε σταθερό χρόνο. Εδώ διευκρινίζουμε ότι λέγοντας "χρόνο λειτουργίας εκτέλεσης" εννοούμε τον χρόνο εκτός του αμιγούς χρόνου εκτέλεσης του κώδικα, δηλαδή του χρόνου που απαιτεί ο ίδιος ο κώδικας να εκτελεστεί.

2.5. Συντακτικό Της Γλώσσας

Παρακάτω παρουσιάζουμε το πλήρες συντακτικό της γλώσσας σε μορφή EBNF όχι απολύτως αυστηρή.

Συνίσταται στον αναγνώστη να προχωρήσει στην ανάγνωση των παρακάτω κεφαλαίων και να χρησιμοποιεί το συντακτικό σαν αναφορά.

```
<program> ::= (<declaration>)+

<declaration> ::= <variable-declaration>
                  | <function-declaration>
                  | <function-definition>;

<basic_type> ::= "int" | "bool" | "char"

<variable-declaration> ::= <type> <ID>

<function-declaration> ::= <result type> <ID> "(" [<param-list>] ")"
";"

<result-type> ::= <type> | "void"

<param-list> ::= <parameter> ("," <parameter>) *

<parameter> ::= <type> <ID>

<function-definition> ::= <result type> <ID> "(" [<param-list>] ")"
            "{" <variable-declaration>* <statement>* "}"

<type> ::=  “codeof” basic_type | “codeof” “void”;


<statement> ::=   ";"
                  | <expression> ";"
```

```
                  | "{" <statement> "}"
                  | "if" "(" <expression> ")" <statement>
                  | "for" "(" <expression> ";" <expression> ";"
                          <expression> ")" <statement>"
                  | "match" "(" <expr> ")" "{" ("case" <pattern> ":"
                      <statement>)+ "}"
                  | "typematch" "(" <expr> ")" "{" ("case"
                      <basic_type>
                  | ":" <id> ":" <statement>)+ "}"
                  | "return" <expression>

<id-list> ::= void | <id> | <id> "," <id-list>

<pattern> ::= "#;"
            | <id> "#;"
            | <id> "#;" <id>
            | "#if" "(" <id> ")" <id>
            | basic_type_symb "#"
            | <id> <unary-op>
            | <unary-ass-post>
            | const
            | "#for" "(" <id> ";" <id> ";" <id> ")" <id>
            | <id>
            | <id> <binary_op> <id>
            | <basic_type_symb> "#" <id>
            | "#" <id> "(" <id-list> ")"
            | "otherwise"

<basic_type_symb> ::= "I" | "B"

<expression> ::=
                  <ID>
                  | "(" <expression> ")"
                  | <const>
                  | <ID> "(" [<expr_list>] ")"
                  | (type) expression
                  | "#;"
                  | "#" <ID>
                  | "#" <const>
                  | "#if" "(" <expression> ")" <expression>
                  | "#for" "(" <expression> ";" <expression> ";"
                      <expression> ")" <expression>
                  | basic_type_symb "#" expression
                  | "#"<ID> "(" [<expr_list>] ")"
                  | <expression> <binary-op> <expression>
                  | <expression> <dbinary-op> <expression>
                  | <unary-op> <expression>
                  | <expression> <unary-ass-post>
                  | <expression> <dunary-ass-post>
                  | <expression> <dunary-op>
                  | "exec" <expression>


<dbinary-op> ::= "#" <binary-op> | "#" ";"

<dunary-op> ::= "#" <unary-op>
```

```
<dunary-ass-post> ::= "#" <unary-ass-pre >

<unary-op> ::= ! | + | - | ++ | --

<unary-ass-pre> ::= ++ | --

<binary-op> ::=   "+" | "-" | "*" | "\" | "=" | "==" | "!=" | ">" |
                  "<" | "&&" | "||"
<expr_list> ::= <expression> (","<expression>)*
```

Σημειώνουμε ότι με το τερματικό σύμβολο const δηλώνεται μια σταθερά οποιουδήποτε σταθερού τύπου.

2.6. Ανάλυση Γλωσσικών Δομών Κατασκευής Δυναμικού Κώδικα

2.6.1. Γενικά

Σε αυτή την ενότητα θα αναλυθεί η σημασία όλων των επεκτάσεων της γλώσσας που αφορούν την κατασκευή δυναμικού κώδικα και θα δειχθεί πως επιτυγχάνεται το επιθυμητό επίπεδο ασφάλειας στη γλώσσα.

Το γενικό πλαίσιο είναι ότι για κάθε στατική έκφραση ή εντολή της γλώσσας θα πρέπει να υπάρχει η αντίστοιχη δυναμική η οποία θα κατασκευάζει κώδικα με παρόμοια λειτουργικότητα με τη στατική δομή.

2.6.2. Το Δυναμικό Σύμβολο

Καθώς υπάρχει μια σχεδόν 1-1 αντιστοιχία μεταξύ των στατικών δομών της γλώσσας και των αντίστοιχων δυναμικών οι πιο πολλές δυναμικές δομές ακολουθούν τη σύνταξη των στατικών με την προσθήκη του συμβόλου #. Με λόγια θα μπορούσαμε να πούμε ότι το σύμβολο έχει την εξής σημασία:

> **# [συντακτικός όρος]: Έχει την ίδια σύνταξη και δέχεται ίσο αριθμό ορισμάτων με το συντακτικό όρο που το καθένα είναι τύπου codeof T όπου T ο τύπος του αντίστοιχου ορίσματος του στατικού όρου. Είναι τύπου codeof R όπου R ο τύπος του συντακτικού όρου. Επιστρέφει κώδικα ο οποίος (αν εκτελεστεί) εκτελεί τον συντακτικό όρο πάνω στα δυναμικά του ορίσματα εκτελώντας τα αν χρειαστεί**

Ο παραπάνω ορισμός είναι αρκετός για να περιγράψει όλες τις δομές

2.6.3. Δεδομένα Δυναμικού Κώδικα

Ο δυναμικός όπως και ο στατικός κώδικας επενεργεί σε κάποια δεδομένα. Αυτά

μπορεί να είναι:

1. Στατικές Μεταβλητές
2. Σταθερές
3. Δυναμικές Μεταβλητές

Όλα αυτά τα δεδομένα έχουν τύπο codeof T όπου T ο τύπος των δεδομένων.

Οι στατικές μεταβλητές και οι σταθερές αποτελούν σταθερές στον δυναμικό κώδικα γιατί παίρνουν την τιμή που έχουν κατά τη δημιουργία του. Από εκεί και πέρα όσες φορές και να εκτελεστεί ο κώδικας θα έχουν αυτή την έννοια ακόμη και αν η στατική μεταβλητή στο μεταξύ αλλάξει τιμή. Δεν μπορούσε φυσικά να είναι διαφορετικά (όταν ο κώδικας εκτελεστεί

Τι είναι όμως οι δυναμικές μεταβλητές; Για να μπορέσει να έχει κάποια έννοια ο δυναμικός κώδικας θα έπρεπε να μπορεί να έχει μεταβλητές, δηλαδή δεδομένα για ανάγνωση ή τροποποίηση. Το πλήθος των δεδομένων αυτών δεν μπορεί να είναι γνωστό κατά τη διάρκεια της μεταγλώττισης, αλλά ο τύπος τους πρέπει να είναι για να γίνει σωστά και εγγυημένα ο έλεγχος τύπων. Μια λύση σε αυτό είναι οι πίνακες (σταθεροί ή μέσω δεικτών), δηλαδή να επιτρέπουμε στον δυναμικό κώδικα να προσπελαύνει δεδομένα στατικών πινάκων και να αποδεικτοδοτεί (dereference) στατικούς δείκτες. Αλλά η γλώσσα μας δεν επιτρέπει ούτε πίνακες ούτε δείκτες και ίσως και να μην ήταν πρακτικό να πρέπει να δημιουργεί πρώτα κάποιος πίνακες ή δείκτες για να δουλέψει με τον δυναμικό κώδικα. Έτσι θεωρήσαμε ότι έχουμε έναν πίνακα με μέγεθος σταθερό και οριζόμενο από τον προγραμματιστή για κάθε στατικό τύπο. Έτσι ο δυναμικός κώδικας μπορεί να αναφέρεται στα στοιχεία αυτών των πινάκων αυτά ακριβώς τα στοιχεία ονομάζουμε δυναμικές μεταβλητές.

Οι συντακτικοί κανόνες που αντιστοιχούν σε αυτά τα δεδομένα είναι οι εξής:

```
<expression> ::=     "#" <ID>  |  //1
                     "#" <const> | //2
                     basic_type_symb "#" expression //2
```

Θα εξηγήσουμε λίγο την ακριβή σημασία των κανόνων αυτών:

- Η 1η έκφραση έχει τύπο codeof δηλαδή δυναμικού κώδικα ο οποίος αν εκτελεστεί θα επιστρέφει την τιμή που είχε η μεταβλητή ID κατά την κατασκευή του δηλαδή κατά την εκτέλεση του `"#" <ID>`.
- Η 2η έκφραση έχει τύπο codeof δηλαδή δυναμικού κώδικα ο οποίος αν εκτελεστεί θα επιστρέφει την σταθερά const.
- Η 3η έκφραση έχει τύπο codeof δηλαδή δυναμικού κώδικα ο οποίος αν εκτελεστεί θα επιστρέφει τη θέση υπ. αριθμό expression (ο τύπος του

expression οπωσδήποτε ακέραιος) του πίνακα δυναμικών μεταβλητών του τύπου basic_type_symb.

Παραδείγματα τέτοιων εκφράσεων είναι τα:

```
#3 //Σταθερά

int i;
#i  //Στατική μεταβλητή

I#34 //Δυναμική Μεταβλητή
```

2.6.4. Δυναμικοί Τελεστές Αντίστοιχοι Στατικών

Κάθε στατικός τελεστής έχει τον αντίστοιχο δυναμικό ο οποίος διαφέρει λόγω της προσθήκης του δυναμικού συμβόλου. Οι δυναμικοί τελεστές είναι οι συντακτικά οι:

- dbinary-op: Δυναμικοί τελεστές δύο τελουμένων (binary operators)
- dunary-op: Δυναμικοί τελεστές ενός τελουμένου (unary operators)
- dunary-ass-post: Δυναμικοί τελεστές ανάθεσης ενός τελουμένου (unary assignment operators) τοποθετούμενοι μετά το όρισμα

Παραδείγματα:

Στατικός Τελεστής	Αντίστοιχος Δυναμικός Τελεστής
==	#==
=	#=
!=	#!=
+	#+
-	#-
%	#%

Πίνακας 2-1

Ο δυναμικός τελεστής έχει τις εξής ιδιότητες:

- Δέχεται σαν ορίσματα τους αντίστοιχους τύπους codeof των στατικών τύπων που δέχεται ο στατικός τελεστής.

  π.χ. ο τελεστής #+ δέχεται ορίσματα τύπου codeof int.

- Επιστρέφει τύπο codeof T όπου T ο τύπος που επιστρέφει ο στατικός τελεστής

  π.χ. ο τελεστής #== επιστρέφει τύπο codeof bool.

- Ο δυναμικός κώδικας στον οποίο αντιστοιχεί αν εκτελεστεί ισοδυναμεί με την εκτέλεση του αντίστοιχου στατικού τελεστή στα αποτελέσματα εκτέλεσης των ορισμάτων του.
- Η σχετική προτεραιότητα των δυναμικών τελεστών είναι ίδια με αυτή των

αντίστοιχων στατικών:

π.χ. ο τελεστής #+(δυαδικός) έχει μικρότερη προτεραιότητα από τον τελεστή #*.

Παράδειγμα 1:

```
codeof bool c;
c = I#21 #== #4.
```

Ο κώδικας c στο τέλος της εκτέλεσης του παραπάνω τμήματος είναι κώδικας ο οποίος επιστρέφει true αν το 21^{ο} στοιχείο του πίνακα δυναμικών μεταβλητών ακεραίων είναι ίσο με 4.

Παράδειγμα 2:

```
codeof int c;
c = I#21 #= #4.
```

Ο κώδικας c στο τέλος της εκτέλεσης του παραπάνω τμήματος είναι κώδικας ο οποίος αναθέτει στο 21^{ο} στοιχείο του πίνακα δυναμικών μεταβλητών ακεραίων την τιμή 4 και επιστρέφει την τιμή του 21^{ου} στοιχείου (δηλ. του 4).

2.6.5. Δυναμικοί Τελεστές Ανάθεσης

Για τους τελεστές ανάθεσης, ενός ή δύο τελουμένων ισχύουν όσα είπαμε προηγουμένως αλλά έχουν μία ιδαιτερότητα. Το αριστερό τους τελούμενο (ή το μοναδικό σε περίπτωση τελεστών ενός τελουμένου) πρέπει να μπορεί να ανατεθεί και συνεπώς σύμφωνα με όσα είπαμε στην ενότητα 2.6.3 πρέπει να είναι δυναμική μεταβλητή. Αυτό δεν ελέγχεται με τη γραμματική αλλά στο στάδιο της σημασιολογικής ανάλυσης.

2.6.6. Το σύμβολο #; - Ο τελεστής #;

Το σύμβολο #; έχει 2 σημασίες:

1. Ως έκφραση είναι τύπου codeof void αναπαριστά την κενή εντολή δηλαδή κώδικα που αν εκτελεστεί θα επιστρέψει αμέσως.
2. Ως τελεστής δύο τελουμένων παριστάνει την παράταξη δύο δυναμικών εντολών (που στη C γίνεται με το σύμβολο ;), δηλαδή τον κώδικα που αν εκτελεστεί θα εκτελέσει διαδοχικά τα δύο του τελούμενα τα οποία είναι και αυτά τύπου codeof void. Ο τελεστής #; δεν παράγεται από στατικό τελεστή. Ακόμη αναφέρουμε ότι ο τελεστής #; είναι ο χαμηλότερος σε προτεραιότητα δυναμικός τελεστής.

Στο εξής όταν αναφερόμαστε σε σύμβολο #; θα εννοούμε την 1^{η} λειτουργία ενώ όταν αναφερόμαστε σε τελεστή #; τη 2^{η}.

2.6.7. Προτεραιότητα Τελεστών

Ανακεφαλαιώνοντας παρουσιάζουμε συγκεντρωμένους τους κανόνες προτεραιότητας των τελεστών στον παρακάτω πίνακα. Οι τελεστές ψηλότερα στον πίνακα έχουν τη μεγαλύτερη προτεραιότητα

Δυναμικοί Τελεστές	**Αντίστοιχοι Στατικών Τελεστής #;**
Στατικοί Τελεστές	

Πίνακας 2-2

Η σχετική προτεραιότητα των δυναμικών τελεστών όπως είπαμε είναι ίδια με αυτή των αντίστοιχων στατικών.

2.6.8. Στατικοί Τελεστές Σε Αντικείμενα Τύπου CODEOF

Ο μοναδικός στατικός τελεστής που επιτρέπεται με τελούμενα αντικείμενα τύπου codeof είναι ο τελεστής ανάθεσης. πχ.

```
codeof int c;
codeof int d;
…
d = I#4 #+ #3;
c = d;
```

Προφανώς ο τελεστής ανάθεσης δεν μπορεί να έχει ως αριστερό μέλος έκφραση codeof παρά μόνο μεταβλητή.

Ακόμη επιτρέπεται η μετατροπή τύπων (typecasting) μεταξύ τύπων codeof.

2.6.9. #IF - #IF … #ELSE

Η έκφραση #if (#if … #else), συντάσσεται ως εξής:

```
#if "(" <codeof bool cond> ")" <codeof void stmt1> [ "#else"
  <codeof void stmt2>]
```

Η λειτουργία του, κατ αντιστοιχία με τον ορισμό του δυναμικού συμβόλου #, είναι η εξής:

Παράγει κώδικα ο οποίος όταν εκτελεστεί εκτελεί τη συνθήκη cond και αν η τιμή επιστροφής της είναι αληθής εκτελεί τον κώδικα stmt1 ενώ αν είναι ψευδής εκτελεί τον κώδικα stmt2. Προφανώς η έκφραση αυτή έχει τύπο codeof void.

2.6.10. #FOR

Η έκφραση #FOR, συντάσσεται ως εξής:

```
#for "(" <codeof void init>, <codeof bool cond>, <codeof void
  iteration>  ")" <codeof void body>
```

Η λειτουργία του, κατ αντιστοιχία με τον ορισμό του δυναμικού συμβόλου #, είναι η εξής:

Παράγει κώδικα ο οποίος όταν εκτελεστεί:

1. Εκτελεί τον κώδικα αρχικοποίησης init.
2. Εκτελεί τη συνθήκη cond και αν η τιμή επιστροφής της είναι αληθής εκτελεί τον κώδικα του σώματος body εάν είναι ψευδής επιστρέφει.
3. Εκτελεί τον κώδικα iteration και επιστρέφει στο βήμα 2

2.6.11. Δυναμική Κλήση Συνάρτησης

Η σύνταξη της δυναμικής κλήσης συνάρτησης είναι η εξής:

```
"#"<F> "(" <code of T1 arg1>,<code of T2 arg2>,…,<code of Tn
  argn>  ")"
```

Όπου F ένα αναγνωριστικό μιας συνάρτησης με επικεφαλίδα:

```
R F(T1,T2,…,Tn)
```

πχ.

```
void printInteger(int i);
...
void main()
{
  codeof void c;
  cideof int ci;
  ...
  c = #printInteger(ci);
}
```

Ο τύπος της έκφρασης είναι codeof R όπου R είναι το αποτέλεσμα της F.

Η σημασία είναι η ακόλουθη:

Η έκφραση παριστάνει κώδικα που αν εκτελεστεί εκτελεί τη συνάρτηση F με παραμέτρους τα αποτελέσματα εκτέλεσης των δυναμικών ορισμάτων arg1,arg2…,argn. Όπως γίνεται φανερό οι τύποι T1,T2..,Tn πρέπει να είναι συμφωνούν με τους τύπους των παραμέτρων της F, όπως και ο αριθμός των δυναμικών ορισμάτων πρέπει να είναι ο ίδιος με τον αριθμό των παραμέτρων της F.

2.7. Εντολές Ανάλυσης - Εκτέλεσης Δυναμικού Κώδικα

2.7.1. Γενικά

Παρακάτω θα δούμε τα τμήματα της γλώσσας που μας επιτρέπουν να υλοποιήσουμε την ανάλυση και την εκτέλεση δυναμικού κώδικα.

2.7.2. match

Η εντολή match είναι αυτή που ουσιαστικά εκτελεί την ανάλυση. Μέσα από αυτή εξετάζεται ένα αντικείμενο κώδικα και εκτελούνται η εντολές ανάλογα με την δομή υψηλού επιπέδου που το αντικείμενο αυτό αναπαριστά.

Αντιγράφουμε τη σύνταξη του match:

```
"match" "(" <codeof void c> ")" "{" ("case" <pattern> ":"
        <statement>)+ "}"
<pattern> ::= "#;"
            | <id> "#;"
            | <id> "#;" <id>
            | "#if" "(" <id> ")" <id>
            | basic_type_symb "#"
            | const
            | "#for" "(" <id> ";" <id> ";" <id> ")" <id>
            | <id>
            | <id> <binary-op> <id>
            | <id> <unary-op>
            | <unary-ass-post>
            | <basic_type_symb> "#" <id>
            | "#" <id> "(" <id-list> ")"
            | "otherwise"
```

Η δομή match λειτουργεί όπως η κλασσική δομή case μόνο που αντί για σταθερές έχει δομές κώδικα. Βλέπουμε ότι για κάθε συντακτική δομή υπάρχει μια δομή pattern. Για κάθε case ελέγχεται το είδος της δομής και αν είναι σύμφωνο με το είδος του pattern τότε:

1. Αναλύεται ο κώδικας και:
   i. Ορίζονται τα αναγνωριστικά ως νέες μεταβλητές με τύπους που προκύπτουν από τη δήλωση του pattern και με εμβέλεια μόνο το συγκεκριμένο case.
   ii. Ανευρίσκονται και τοποθετούνται στις μεταβλητές αυτές τα επιμέρους στοιχεία του κώδικα.

2. Εκτελούνται οι εντολές μέσα στο case
3. Ο έλεγχος μεταφέρεται στην επόμενη εντολή από το match.

Προφανώς στο otherwise δεν γίνεται έλεγχος και εκτελείται οπωσδήποτε αν κανένα άλλο δεν συμφωνεί.

Όσο αφορά το βήμα 1.ii ο προσεκτικός αναγνώστης θα παρατηρήσει ότι για κάποιες δομές είναι αδύνατο να προσδιορισθεί ο τύπος των μεταβλητών που θα ορισθούν πχ. το pattern

```
x1 #== x2
```

δεν μπορεί να καθορίσει τον τύπο των x1,x2 καθώς μπορεί να είναι οποιοσδήποτε codeof (εκτός από void). Για να μην δημιουργήσει αυτό κάποια αστάθεια στο σύστημα τύπων ορίζουμε ως τύπο αυτών των μεταβλητών τον code of void που είναι και ο πιο γενικός τύπος codeof (δηλ. οποιουδήποτε τύπος codeof μπορεί να χρησιμοποιηθεί ως codeof void χωρίς πρόβλημα).

2.7.3. typematch

Όπως είδαμε και στο match αλλά και αλλού ο τύπος codeof void χρησιμοποιείται σαν γενικός τύπος δηλ. στη θέση μιας έκφρασης τύπου codeof void μπορούμε να αναθέσουμε οποιαδήποτε έκφραση τύπου codeof. Για να μπορούμε να εντοπίσουμε τον πραγματικό τύπο των μεταβλητών τύπου codeof void, δηλαδή τι επιστρέφουν εκ της κατασκευής τους, χρησιμοποιούμε το typematch.

Το συντακτικό της typematch ακολουθεί:

```
"typematch" "(" <expr> ")" "{" ("case" <basic_type>
":" <id> ":" <statement>)+ "}"
```

Η έκφραση <expr> πρέπει να έχει τύπο codeof void.

Η typematch ανευρίσκει τον πραγματικό τύπο του κώδικα και εκτελεί την αντίστοιχη εντολή case αφού πρώτα έχει δηλώσει τη μεταβλητή <id> με τύπο codeof <basic_type> και της έχει αναθέσει την έκφραση <expr>. Σημειώνουμε ότι ο πραγματικός τύπος μπορεί να είναι codeof void αν όντως ο κώδικας δεν επιστρέφει τίποτα.

2.7.4. exec

Το συντακτικό της έκφρασης exec είναι το ακόλουθο:

```
exec <codeof T c>
```

Η παραπάνω έκφραση έχει τύπο T. Η έκφραση εκτελεί τον κώδικα c και επιστρέφει το αποτέλεσμά του που είναι τύπου T (με T φυσικά να μπορεί να είναι void). Η εκτέλεση δεν επιφέρει καμία αλλαγή στο αντικείμενο c.

2.8. Διαχείριση Μνήμης

Η γλώσσα όπως είναι δεν υποστηρίζει δείκτες ή πίνακες οπότε θα περίμενε κανείς ότι δεν χρειάζεται διαχείριση μνήμης. Παρ' όλα αυτά ο δυναμικός κώδικας δημιουργείται στο χρόνο εκτέλεσης χωρίς αυτό να γίνεται ορατό από το χρήστη οπότε ένας μεταγλωττιστής θα έπρεπε να φροντίζει για την απελευθέρωση του χώρου που δεν χρησιμοποιείται από δυναμικό κώδικα. Το πώς γίνεται αυτό το βλέπουμε στο επόμενο κεφάλαιο.

3. Ο Μεταγλωττιστής

3.1. Γενικά

Στα παρακάτω θα περιγραφεί ο μεταγλωττιστής για τη γλώσσα SMEC Θα αναλυθούν τα τμήματά του και θα περιγραφεί η λειτουργία τους καθώς και οι μέθοδοι που ακολουθήθηκαν για την υλοποίηση τους καθώς και οι δυναμικές επεκτάσεις. Καθώς το κύριο αντικείμενο μας είναι οι δυναμικές επεκτάσεις τα τμήματα του μεταγλωττιστή που δεν έχουν τέτοιες ή η αντιμετώπισή τους από πλευράς λειτουργίας και υλοποίησης δεν διαφέρει από των στατικών (π.χ. λεκτικός και συντακτικός αναλυτής) θα παρουσιαστούν εν συντομία. Επίσης θα αναλυθούν οι βασικές δομές και έννοιες παράστασης του δυναμικού κώδικα που διέπουν όλο το μεταγλωττιστή σε ξεχωριστή ενότητα.

Ο μεταγλωττιστής υλοποιήθηκε σε standard C++. Χρησιμοποιήθηκαν τα μεταεργαλεία flex και bison για την παραγωγή λεξιλογικού και σημασιολογικού αναλυτή αντίστοιχα.

Η υλοποίηση του μεταγλωττιστή σε προγραμματιστικό επίπεδο είναι τέτοια ώστε τα

διάφορα τμήματά του να μπορούν να επαναχρησιμοποιηθούν ή να αλλαχθούν εξ' ολοκλήρου (ώστε πχ. να παράγει τελικό κώδικα για διαφορετικό υπολογιστή) χωρίς οι μεταξύ τους διασυνδέσεις να χρειάζεται να μεταβληθούν.

Ο μεταγλωττιστής δεν υλοποιήθηκε από την αρχή αλλά βασίστηκε στον μεταγλωττιστή για το στατικό υποσύνολο της γλώσσας που είχε αναπτύξει ο συγγραφέας στο πλαίσιο του μαθήματος Μεταγλωττιστές του 8^{ου} εξαμήνου.

Στα παρακάτω θα παρουσιάσουμε τα τμήματα του μεταγλωττιστή και για το κάθε τμήμα θα αναφέρεται ξεχωριστά η υλοποίηση του κλασσικού μεταγλωττιστή και των δυναμικών επεκτάσεων. Θεωρούμε πως βασικά θέματα υλοποίησης μεταγλωττιστών προστακτικών γλωσσών είναι γνωστά στον αναγνώστη και θα επαναλαμβάνονται όπου κρίνεται αυτό απαραίτητο για λόγους επεξήγησης των επεκτάσεων.

3.2. Απόδοση

Στην ενότητα αυτή θα ασχοληθούμε με την απόδοση του μεταγλωττιστή αυτού όσον αφορά τις λειτουργίες που αφορούν τον δυναμικό κώδικα. Όπως θα δούμε οι πολυπλοκότητες είναι χαμηλές. Το πώς αυτές επιτυγχάνονται θα το δούμε στη συνέχεια. Όλες οι πολυπλοκότητες θα δοθούν ως προς το μέγεθος του κώδικα.

Λειτουργία	**Κόστος (n μέγεθος του κώδικα)**
Κατασκευή	O(n)
Εκτέλεση	O(1)
Ανάλυση	O(1)
Πλήρης Ανάλυση	O(n)

Ως πλήρη ανάλυση θεωρούμε την διεξοδική ανάλυση κώδικα μέχρι τα βασικά τμήματα, αυτή δεν είναι βασική λειτουργία αλλά μπορεί εύκολα να υλοποιηθεί αναδρομικά με τη δομή match. Μια τέτοια συνάρτηση δίνουμε στα παραδείγματα.

3.3. Διάρθρωση

3.3.1. Τμήματα

Ο μεταγλωττιστής αποτελείται από τα εξής τμήματα τα οποία εκτελούν ένα στάδιο της εργασίας της μεταγλώττισης:

- **Λεκτικός αναλυτής**

Ο λεκτικός αναλυτής παίρνει ως είσοδο το κείμενο και εντοπίζει τις λεκτικές μονάδες της γλώσσας (βλ. Κεφ. 2) και εμφανίζει μοιραίο σφάλμα (fatal error) αν δεν βρεθεί μια τέτοια μονάδα.

Η υλοποίηση του λεκτικού αναλυτή είναι σχετικά απλή και έγινε με το

μεταεργαλείο flex.

Η λειτουργία και η υλοποίηση του λεκτικού αναλυτή είναι εξαιρετικά απλές, το μόνο που πρέπει να επισημανθεί είναι ότι με τη μετατροπή η yylex δεν επιστρέφει μόνο τον κωδικό της επόμενης λεκτικής μονάδας αλλά και κάποιες άλλες πληροφορίες που φαίνονται στο struct `SLexUn`. Ο λεκτικός αναλυτής εντοπίζει λανθασμένες παρενθέσεις, άγκιστρα ή αγκύλες. Τα σχόλια δεν είναι φωλιασμένα αλλά πρέπει να συντάσσονται όπως οι παρενθέσεις. Παρέχεται η δυνατότητα include στα οποία γίνεται έλεγχος για την ύπαρξή τους αλλά όχι για κυκλικά includes ή διπλά includes. Η σύνταξη των includes είναι ίδια όπως στη C. Επίσης στα ονόματα των includes τα / και \ θεωρούνται ισοδύναμα.

- **Συντακτικός αναλυτής**

Ο συντακτικός αναλυτής ελέγχει την συμφωνία του κώδικα με το συντακτικό της γλώσσας. Στο πλαίσιο του συντακτικού αναλυτή εκτελούνται τα άλλα τμήματα του προγράμματος, δηλαδή έχουμε μεταγλώττιση οδηγούμενη από τη σύνταξη.

Η υλοποίηση του συντακτικού ελέγχου είναι απλή και έγινε με τη βοήθεια του μεταεργαλείου Bison. Δεν προβλέπονται δυνατότητες ανάνηψης από συντακτικά σφάλματα. Κάθε συντακτικό σφάλμα είναι μοιραίο.

Σημειώνουμε ότι το else θεωρείται ότι ανήκει στο κοντινότερο if (καθώς το bison προτιμά το reduce στα shift-reduce conflicts.

- **Σημασιολογικός αναλυτής**

Ο σημασιολογικός αναλυτής εξετάζει την ορθότητα του πρόγράμματος και τη συμφωνία με το σύστημα τύπων. Αναλυτικά η υλοποίηση στην ενότητα 3.5.

- **Γεννήτορας ενδιάμεσου κώδικα**

Ο γεννήτορας ενδιάμεσου κώδικα παράγει τον ενδιάμεσο κώδικα για τον οποίο επιλέξαμε τη μορφή των τετράδων. Αναλυτικά η υλοποίηση στην ενότητα 3.6.

- **Γεννήτορας τελικού κώδικα**

Ο γεννήτορας ενδιάμεσου κώδικα παράγει τον ενδιάμεσο κώδικα για τον οποίο επιλέξαμε τη μορφή των τετράδων. Αναλυτικά η υλοποίηση στην ενότητα 3.7.

3.3.2. Αρχεία - Κλάσεις

Παρακάτω παρουσιάζεται η διάρθρωση του μεταγλωττιστή και αναφέρονται τα αρχεία που τον αποτελούν.

Αρχείο	Τμήμα	Λειτουργία
cplist.cpp	Βοηθητικό	Υλοποίηση Κλάσης Ουράς (CPList)
graph.cpp	Βοηθητικό	Υλοποίηση Γράφου
helpfull.cpp	Βοηθητικό	Βοηθητικά macros και συναρτήσεις
cplist.h	Βοηθητικό	Δήλωση Κλάσης Ουράς (CPList)
graph.h	Βοηθητικό	Δήλωση Κλάσεων Γράφων
helpfull.h	Βοηθητικό	Βοηθητικά macros και συναρτήσεις
newquads.cpp	Γεννήτορας Ενδιαμέσου Κώδικα	Υλοποίηση συναρτήσεων παραγωγής νέων τετράδων
quads.cpp	Γεννήτορας Ενδιαμέσου Κώδικα	Υλοποίηση κλάσης Γενν. Ενδ. Κώδικα (CQuads)
quads.h	Γεννήτορας Ενδιαμέσου Κώδικα	Δήλωση κλάσης Γενν. Ενδ. Κώδικα (CQuads)
cfinalcode.cpp	Γεννήτορας Τελικού Κώδικα	Υλοποίηση κλάσης γεννήτορα τελικού κώδικα (CFinalCode)
patterns.cpp	Γεννήτορας Τελικού Κώδικα	Υλοποίηση κλάσης προτύπων δυναμικού κώδικα (CPattern)
patterns.h	Γεννήτορας Τελικού Κώδικα	Δήλωση κλάσης προτύπων δυναμικού κώδικα (CPattern)
foptimize.cpp	Γεννήτορας Τελικού Κώδικα	Υλοποίηση συναρτήσεων βελτιστοποίσης
cfinalcode.h	Γεννήτορας Τελικού Κώδικα	Δήλωση κλάσης γεννήτορα τελικού κώδικα (CFinalCode)
edsgar_comp.cpp	Κύριο Πρόγραμμα	
lexyy.cpp	Λεκτικός Αναλυτής	Παραγόμενο του flex
lex.h	Λεκτικός Αναλυτής	Βοηθητικές Δηλώσεις
lexend.h	Λεκτικός Αναλυτής	Βοηθητικές Δηλώσεις
flexinst.txt	Λεκτικός Αναλυτής	Αρχείο οδηγιών για το flex
dyn_semantics.cpp	Σημασιολογικός Αναλυτής – Δυναμικό Τμήμα	Υλοποίηση σημασιολογικών ρουτινών για το δυναμικό τμήμα της γλώσσας
semantics.cpp	Σημασιολογικός Αναλυτής	Υλοποίηση σημασιολογικών ρουτινών
semantics.h	Σημασιολογικός Αναλυτής	Δήλωση σημασιολογικών ρουτινών και τύπων
syntax.h	Συντακτικός Αναλυτής / Σημασιολογικός Αναλυτής	Παραγόμενο του Bison
bison.txt	Συντακτικός Αναλυτής Σημασιολογικός Αναλυτής	Αρχείο οδηγιών για το bison / Υλοποίηση Σημ. ρουτίνων
symbol.h	Σημασιολογικός Αναλυτής	Δήλωση Πίνακα Συμβόλων
symbol.cpp	Σημασιολογικός Αναλυτής	Υλόποιηση Πίνακα Συμβόλων

syntax.cpp	Συντακτικός Αναλυτής / Σημασιολογικός Αναλυτής	Παραγόμενο του Bison

Πίνακας 3-1

3.4. Δυναμικός Κώδικας

3.4.1. Γενικά

Σε αυτή την ενότητα θα δούμε το χειρισμό του δυναμικού κώδικα και των δυναμικών δεδομένων από το μεταγλωττιστή και τις βασικές δομές αναπαράστασής του.

Με την παρούσα υλοποίηση οι λειτουργίες κατασκευής, ανάλυσης, σύνθεσης και αποσύνθεσης πραγματοποιούνται με πολύ μικρό κόστος σε χρόνο, το πολύ O(n) με n το μήκος του κώδικα για την κατασκευή.

Η υλοποίηση του δυναμικού κώδικα επιτυγχάνεται με συνδυασμό του Αντικειμένου Δυναμικού Κώδικα (ΑΔΚ), των πρωτοτύπων δυναμικού κώδικα (ΠΔΚ) και των συναρτήσεων βιβλιοθήκης που αναλαμβάνουν να εκτελούν πράξεις στο χρόνο εκτέλεσης.

Στο κεφάλαιο αυτό δεν παρουσιάζονται λεπτομέρειες υλοποίησης που περιορίζονται σε ένα τμήμα του μεταγλωττιστή για να μπορεί ο αναγνώστης να σχηματίσει πλήρη εικόνα σε σχετικά σύντομο διάστημα ενώ όποιος ενδιαφέρεται για τις ακριβείς λεπτομέρειες υλοποίησης παραπέμπεται στις επόμενες ενότητες.

Σημειώνουμε κάτι όσον αφορά τα μεγέθη πεδίων που εμφανίζονται παρακάτω. Θα χρησιμοποιήσουμε τους όρους "μήκος δείκτη" (ΜΔ) και "μήκος λέξης" (ΜΛ) για να εκφράσουμε τον μήκος σε bytes μιας λέξης και ενός δείκτη αντίστοιχα στη μηχανή στόχο της μεταγλώττισης. Αυτό γίνεται για να δώσουμε όσο το δυνατό περισσότερη γενικότητα στο κείμενο. Και τα δύο μήκη για το μηχάνημα στόχο για το οποίο υλοποιήθηκε ο παρών μεταγλωττιστής είναι 2.

3.4.2. Δυναμικά Δεδομένα

Από την περιγραφή της γλώσσας γνωρίζουμε ότι τα δυναμικά δεδομένα είναι 2 ειδών:

4. Στατικές Μεταβλητές – Σταθερές (πχ. `#i,#3`)
5. Δυναμικές Μεταβλητές (πχ. `I#4, B#5, C#400`)

Τα δυναμικά δεδομένα έχουν και αυτά τύπο codeof, δηλαδή είναι αντικείμενα κώδικα που μπορούν να χρησιμοποιηθούν ως τέτοια.

Οι στατικές μεταβλητές και οι σταθερές υλοποιούνται ως ένα αντικείμενο κώδικα

που επιστρέφει την ίδια τη σταθερά.

Οι δυναμικές μεταβλητές αποθηκεύονται σε μια σειρά από πίνακες (ένας ανά είδος τύπου δεδομένων) το μέγεθος των οποίων καθορίζεται με παράμετρο στο μεταγλωττιστή. Άρα το I#40 είναι κώδικας που επιστρέφει δείκτη στο 40° στοιχείο του πίνακα δυναμικών ακεραίων. Οι πίνακες αυτοί είναι καθολικοί και δεν εξαρτώνται από τη μονάδα του προγράμματος.

3.4.3. Παράσταση Δυναμικού Κώδικα - Αντικείμενο Δυναμικού Κώδικα (Dynamic Code Object)

Ο δυναμικός κώδικας οργανώνεται ως ένα σύνολο Αντικειμένων Δυναμικού Κώδικα (εφεξής θα αναφέρονται ΑΔΚ για λόγους συντομίας. Η δομή αυτή έχει μεταβλητό μήκος και δημιουργείται και αποθηκεύεται στο σωρό(heap). Περιλαμβάνει ένα μέρος κώδικα καθώς και μεταδεδομένα, δηλ. πληροφορίες που θα διευκολύνουν στην αναγνώριση, την αποσύνθεση και την καταστροφή της. Οι μεταβλητές τύπου codeof είναι στην ουσία δείκτες σε ΑΔΚ.

Ο κώδικας που περιέχουν ΑΔΚ είναι πανομοιότυπος με τον κώδικα μιας συνάρτησης της γλώσσας χωρίς παραμέτρους που θα επέστρεφε ότι επιστρέφει η ΑΔΚ (προφανώς περιλαμβάνεται και το κενό(void) στις δυνατότητες επιστροφής). Αυτό σημαίνει ότι κάθε ΑΔΚ έχει το δικό του εγγράφημα δραστηριοποίησης και υπάρχει η δυνατότητα ένα ΑΔΚ να καλεί ένα άλλο.

Ένα ΑΔΚ αναπαριστάται στη μνήμη ως εξής:

Πεδίο	Μέγεθος(σε bytes)
TYPE (Είδος)	2+ΜΔ
Αρ. Αναφορών (REF_COUNT)	ΜΛ
Αρ. Εξαρτήσεων(DEP_COUNT)	ΜΛ
Μήκος Κώδικα (CODE_LENGTH)	ΜΔ
Κώδικας	CODE_LENGTH
Πίνακας εξαρτήσεων	DEP_COUNT*ΜΔ

Πίνακας 3-2

Το είδος του κώδικα εκφράζει την δομή που εκείνος αναπαριστά. Ο αρ. αναφορών είναι ένα κλασσικός μετρητής αναφορών που χρησιμεύει στην καταστροφή της ΑΔΚ.

Οι εξαρτήσεις περιγράφονται αναλυτικά στα επόμενα.

3.4.4. Γενικές Αρχές Κατασκευής Δυναμικού Κώδικα

Η κατασκευή δυναμικού κώδικα γίνεται ακολουθώντας το (ιδεατό συντακτικό δέντρο), όπως ακριβώς γίνεται και η κατασκευή του στατικού κώδικα. Δηλαδή όπως ακριβώς η έκφραση

```
z=(x+4)+y
```

αποτιμάται ως

```
$1 = x+1
z = $1+y
```

Έτσι και το

```
z=(#x #+ #4) #+ #y
```

αποτιμάται ως:

```
$1 = #x #+ #1
z = $1 #+ #y
```

Δηλαδή κατασκευάζεται πρώτα κώδικας που όταν εκτελεστεί επιστρέφει το αποτέλεσμα της πράξης `x + 1` και έπειτα ο κώδικας που όταν εκτελεστεί επιστρέφει το αποτέλεσμα της πράξης `exec($1) + y`.

Αυτά που πρέπει να διευκρινιστούν είναι:

1. Ο τρόπος που γίνεται αυτή η κατασκευή του κώδικα καθώς και
2. σε ποιο στάδιο (μεταγλώττιση εκτέλεση θα γίνεται η κατασκευή) και αν γίνεται και στα 2, ποιες εργασίες θα γίνονται σε κάθε στάδιο.

Το 1ο από αυτά θα απαντηθεί στην παρούσα ενότητα ενώ το 2ο στην επόμενη.

Ορίζουμε κάποιες δομές δυναμικού κώδικα ως βασικές με την έννοια ότι ο κώδικας που παράγουν δεν καλεί άλλες δομές κώδικα. Οι βασικές αυτές δομές είναι τα δεδομένα του δυναμικού κώδικα, δηλαδή:

1. Στατικές Μεταβλητές – Σταθερές (πχ. `#i,#3`)
2. Δυναμικές Μεταβλητές (πχ. `I#4, B#5, C#400`)

Ο κώδικας που παράγουν όλες οι άλλες δομές είναι όμοιος με τον αντίστοιχο στατικό κώδικα με τη διαφορά ότι προηγείται ένα αρχικό στάδιο που εκτελούνται οι δυναμικές συνιστώσες του δέντρου.

Για παράδειγμα έστω ο ακόλουθος δυναμικός κώδικας:

```
codeof int c,c1,c2;
…
c = c1 #+ c2
```

Ο κώδικας c θα είναι ο εξής:

```
$1 = exec c1
$2 = exec c2
$3 = $1 + $2
ret $3
```

Δηλαδή εκτελούνται οι κώδικες c1,c2 και έπειτα τα αποτελέσματα (που είναι ακέραιοι) προστίθενται για να δώσουν το τελικό αποτέλεσμα του κώδικα c. Σημειώνουμε ότι ο κώδικας εδώ αναπαριστάται με μια όχι τυπική γλώσσα για να κατανοηθούν οι λειτουργίες του.

Θα θέλαμε να τονίσουμε ότι δεν πρέπει να μπερδεύουμε τον κώδικα που παρουσιάστηκε αμέσως παραπάνω με αυτόν στην αρχή της ενότητας καθώς ο 1ος είναι ο κώδικας που παράγεται (δηλ. ο κώδικας που περιέχεται στο ΑΔΚ) ενώ ο 2ος είναι ο κώδικας ο οποίος κατασκευάζει το ΑΔΚ.

3.4.5. Λεπτομέρειες Κατασκευής Δυναμικού Κώδικα – Πρότυπα

Αν ακολουθήσουμε τα παραπάνω παρατηρούμε ότι για όλες τις δομές κώδικα οι λειτουργίες που θα εκτελεστούν είναι γνωστές αλλά είναι άγνωστα τα δεδομένα στα οποία οι λειτουργίες αυτές θα εκτελεστούν (π.χ. στο παραπάνω παράδειγμα όλες οι λειτουργίες του κώδικα c είναι γνωστές αλλά άγνωστα παραμένουν τα c1,c2) . Φυσικά τα δεδομένα αυτά μπορεί να είναι και τα ίδια αντικείμενα κώδικα, αυτό όμως δεν μας απασχολεί.

Για να εκμεταλλευτούμε το παραπάνω συμπέρασμα αποφασίσαμε να δημιουργούμε τα Πρότυπα Δυναμικού Κώδικα (ΠΔΚ). Τα πρότυπα αυτά είναι κανονικός κώδικας σε γλώσσα μηχανής ο οποίος περιέχει μηδενικές διευθύνσεις εκεί όπου θα έπρεπε να είναι οι διευθύνσεις των δεδομένων εισόδου. Το παράδειγμα με το c θα μπορούσε να δημιουργήσει το ακόλουθο πρότυπο σε ψευδο-assembly[2]:

```
call 0
mov $1,ax
call 0
mov cx,$1
add ax,cx
```

Ενώ ο τελικός κώδικας όταν τελειώσει η κατασκευή (στο χρόνο εκτέλεσης) θα είναι:

```
call c1
mov $1,ax
call c2
mov cx,$1
add ax,cx
```

Βλέπουμε ότι η μόνη διαφορά είναι η αντικατάσταση των μηδενικών από τα c1,c2 αντίστοιχα.

---

[2] Υποθέσαμε ότι exec υλοποιείται με ένα call και ότι το αποτέλεσμα του δυναμικού κώδικα που εκτελείται

Η χρησιμότητα των προτύπων είναι ότι μπορούν να κατασκευαστούν εξ ολοκλήρου στο χρόνο μεταγλώττισης(εφόσον δεν περιέχουν αναφορές σε δεδομένα) και το μόνο που θα χρειαστεί να γίνει σε χρόνο εκτέλεσης είναι να τοποθετηθούν στα κατάλληλα σημεία τα δεδομένα. Οι θέσεις των μηδενικών στις οποίες τοποθετούνται τα δεδομένα τις ονομάζουμε σχισμές (slots). Τα πρότυπα κατασκευάζονται ως κώδικας assembly και τοποθετούνται στο αρχείο τελικού κώδικα. Φυσικά τα πρότυπα περιέχουν και μεταδεδομένα για την τοποθεσία των σχισμών, το μέγεθος του κώδικα κτλ. Η δομή ενός προτύπου είναι η εξής:

Πεδίο	**Μέγεθος(σε bytes)**
Μήκος Κώδικα (CODE_LENGTH)	ΜΔ
Αρ. Σχισμών (SLOT_COUNT)	ΜΛ
Κώδικας	CODE_LENGTH
Πίνακας Σχισμών	SLOT_COUNT *ΜΔ

Πίνακας 3-3

Η σημασία όλων των πεδίων εκτός του πίνακα σχισμών είναι προφανής. Ο πίνακας σχισμών είναι ένας πίνακας με τις σχετικές θέσεις των σχισμών (σε bytes) από την αρχή του κώδικα.

Τα πρότυπα κατασκευάζονται ακριβώς όπως ο στατικός κώδικας. Ελέγχεται δηλαδή η σημασιολογία με τη βοήθεια του συστήματος τύπων, κατασκευάζεται ενδιάμεσος κώδικας και έπειτα ο ενδιάμεσος μετατρέπεται σε τελικό. Για κάθε ΠΔΚ δημιουργείται μια ενότητα προγράμματος με δικό της ενδιάμεσο και τελικό κώδικα. Για τον ακριβή κώδικα που παράγει κάθε δομή καθώς και τις λεπτομέρειες υλοποίησης βλ. ενότητες 3.6, 3.7.

Στον παρακάτω πίνακα φαίνονται οι διαδικασίες που γίνονται κατά τη διάρκεια της κατασκευής του κώδικα και σε ποιο στάδιο γίνονται αυτές:

Στάδιο Μεταγλώττισης	Στάδιο Εκτέλεσης
Κατασκευάζεται το πρότυπο με βάση τον κώδικα	**Αντιγράφεται ο κώδικας του προτύπου σε μια νέα ΑΔΚ.** **Τοποθετούνται τα δεδομένα στις σχισμές στο αντίγραφο του κώδικα.** **Κατασκευάζεται ο πίνακας εξαρτήσεων (βλ. 3.4.7)**

Πίνακας 3-4

Βλέπουμε ότι ο κώδικας του προτύπου αντιγράφεται αρχικά και μετά γίνονται οι αντικαταστάσεις. Η αντιγραφή καθίσταται αναγκαία καθώς ένα πρότυπο μπορεί να χρησιμοποιηθεί πολλές φορές στο χρόνο εκτέλεσης για την παραγωγή πολλών αντικειμένων κώδικα. Ένα ενδεικτικό παράδειγμα είναι:

```
int i;
codeof int c;
c = I#0;
for (i=0;i<10;i++)
  c = c #+ #1;
```

Το τμήμα δυναμικού κώδικα μέσα στο for εμφανίζεται μόνο μία φορά και άρα θα δημιουργήσει μόνο ένα πρότυπο όμως στο χρόνο εκτέλεσης θα δημιουργηθούν πολλά αντικείμενα δυναμικού κώδικα τα οποία όμως δεν πρέπει να επικαλύπτονται στη μνήμη.

Η προσέγγιση κατασκευής που ακολουθήσαμε βελτιώνει πάρα πολύ την απόδοση αφού ουσιαστικά όλη η μεταγλώττιση του δυναμικού κώδικα γίνεται στο χρόνο μεταγλώττισης του προγράμματος. Αν παρατηρήσουμε τον πίνακα το χρονικό κόστος των διαδικασιών που γίνονται στο χρόνο εκτέλεσης είναι σχετικά μικρό (O(n) σε πολυπλοκότητα αν n είναι το μήκος του κώδικα). Αν σκεφθούμε ότι το μήκος του κώδικα είναι γενικά μικρό αυτό οδηγεί σε πολύ μικρή χρονική επιβάρυνση σε σχέση με έναν εξ ολοκλήρου στατικό κώδικα.

Παραπέρα το μοντέλο αυτό μας βοηθάει να υλοποιούμε γρήγορα και τις αλλές λειτουργίες (εκτέλεση, ανάλυση, αποδόμηση). Κατασκευάζοντας τον κώδικα ιεραρχικά επιτρέπει στην ανάλυση και αποδόμηση να γίνονται σε πολύ μικρό χρόνο (γραμμικό ως προς το μέγεθος των σχισμών του προτύπου) και την εκτέλεση σε σταθερό χρόνο O(1) όπως θα δούμε παρακάτω.

3.4.6. Εκτέλεση Δυναμικού Κώδικα

Η εκτέλεση του δυναμικού κώδικα είναι η απλούστερη διαδικασία καθώς το μοντέλο κατασκευής και αναπαράστασης είναι τέτοιο που ο κώδικας μετά την περάτωση της κατασκευής του στο χρόνο εκτέλεσης μπορεί να τρέξει με μια απλή εντολή call στην αρχή του μέρους κώδικα της ΑΔΚ ανεξαρτήτως των ενοτήτων του προγράμματος στις οποίες γίνεται η κατασκευή και η κλήση. Αυτό επιτυγχάνεται για τους παρακάτω λόγους:

1. Τα αντικείμενα κώδικα αποθηκεύονται στο σωρό άρα οι διευθύνσεις τους δεν εξαρτώνται από το σημείο κλήσης.
2. Το κάθε αντικείμενο κώδικα έχει δικό του εγγράφημα δραστηριοποίησης (και συνεπώς δικές του μεταβλητές) και υπακούει στο γενικό μοντέλο κλήσης συναρτήσεων με κάποιες διαφοροποιήσεις.
3. Τα δυναμικά δεδομένα με τα οποία αλληλεπιδρά ο κώδικας έχουν σταθερή καθολική διεύθυνση ανεξάρτητη του σημείου κλήσης.
4. Ο κώδικας που κατασκευάζεται δεν χρειάζεται καμία άλλη μετατροπή μετά

την κατασκευή του.

Ίσως αυτό να είναι και το μεγαλύτερο πλεονέκτημα της προσέγγισής μας καθώς η εκτέλεση του κώδικα μπορεί να επαναλαμβάνεται και είναι πολύ σημαντικό να έχει όσο το δυνατόν μικρότερο κόστος.

3.4.7. Ανάλυση – Αποδόμηση Δυναμικού Κώδικα

Όπως είδαμε στην περιγραφή της γλώσσας υποστηρίζεται ανάλυση του κώδικα που επιτυγχάνεται με τη δομές `match` και `typematch` οι οποίες αναγνωρίζουν τη δομή την οποία αναπαριστά ο κώδικας και τον τύπο του κώδικα σε περίπτωση που ο τελευταίος είναι άγνωστος. Σημειώνουμε και πάλι ότι λέγοντας αποδόμηση δεν εννοούμε καταστροφή αλλά εξόρυξη των συστατικών μερών ενός ΑΔΚ.

Κατά την εκτέλεση μιας εντολής typematch η εργασία που πρέπει να γίνει είναι η ανεύρεση του τύπου του δυναμικού κώδικα.

Κατά την εκτέλεση μιας εντολής match/typematch πρέπει να γίνουν τα εξής:

1. Η ανεύρεση του είδους της δομής υψηλού επιπέδου που ο κώδικας εκτελεί
2. Η ανεύρεση του τύπου της δομής
3. Η ανεύρεση των επιμέρους στοιχείων της δομής υψηλού επιπέδου και η δημιουργία ΑΔΚ που να εκτελούν τα στοιχεία αυτά.

Δύο στοιχεία της παραπάνω υλοποίησης μας βοηθούν στις εργασίες αυτές:

- Το πεδίο τύπου στα ΑΔΚ
- Οι εξαρτήσεις στα ΑΔΚ

Στο πεδίο τύπου μήκους 4 byte αποθηκεύονται κατά την κατασκευή του τα εξής:

- Ο τύπος του ΑΔΚ (codeof int, codeof char …) (μήκος 1 byte)
- Το είδος της δομής υψηλού επιπέδου που υλοποιεί ο κώδικας (π.χ. ανάθεση, κλήση συνάρτησης print, τελεστής ==)

Εδώ σημειώνουμε ότι καθώς υπάρχουν προς το παρών μόνο απλοί τύποι η αποθήκευση πληροφοριών τύπου είναι εύκολη (αποθηκεύονται σε ένα μόνο πεδίο μήκους μερικών bit). Αν είχαμε π.χ. δείκτες θα έπρεπε να χρησιμοποιήσουμε συνδεδεμένες δομές για να αποθηκεύσουμε τύπους πράγμα εφικτό και όχι ιδιαίτερα δύσκολα υλοποιήσιμο.

Το είδος της δομής αποθηκεύεται ως εξής:

- Ένα byte για το γενικό είδος της δομής που μπορεί να είναι ένα από τα

ακόλουθα:

- Κενός Κώδικας
- Δομή IF
- Δομή FOR
- Σταθερά
- Δυναμική Μεταβλητή
- Κλήση Συνάρτησης
- Δυαδικός Τελεστής
- Τελεστής ενός τελούμενου
- Δομή IF-ELSE

- Ένα ΜΔ για επιπλέον πληροφορίες που χρειάζονται για να προσδιοριστεί πλήρως το είδος της δομής υψηλού επιπέδου. Αυτές είναι ανάλογα με τη δομή:
  - Η διεύθυνση της καλούμενης συνάρτησης
  - Ένα αναγνωριστικό τελεστή

Ο λόγος που έγινε αυτός ο διαχωρισμός έχει να κάνει με την καλύτερη συγγραφή του μεταγλωττιστή και θα γίνει φανερό στην ενότητα 3.6.

Με τις παραπάνω πληροφορίες καταφέρνουμε εύκολα να εντοπίσουμε το είδος της δομής υψηλού επιπέδου που υλοποιεί ένα ΑΔΚ στο χρόνο εκτέλεσης σε σταθερό χρόνο (O(1)). Τώρα θα δούμε πως εντοπίζουμε τα επιμέρους τμήματα που συνδέουν μια δομή υψηλού επιπέδου και μπορούμε να τα επιστρέφουμε για να επαναχρησιμοποιηθούν επίσης με πολύ μικρό υπολογιστικό κόστος.

Ας δούμε αρχικά τι εννοούμε όταν λέμε επιμέρους τμήματα. Εννοούμε τα ΑΔΚ[3] που εκτελούνται στον κώδικα ενός ΑΔΚ και στα αποτελέσματά τους βασίζεται η εκτέλεση του ΑΔΚ. Τα ΑΔΚ αυτά ταυτίζονται με τα παιδιά της δομής υψηλού επιπέδου που υλοποιεί το ΑΔΚ στο συντακτικό δέντρο του προγράμματος. Ας θεωρήσουμε για παράδειγμα μια εντολή if:

```
#if (I#4 #== #0) #print(1) #;
```

Εδώ τα επιμέρους τμήματα μιας εντολής if είναι τα:

```
I#4 #== #0 //συνθήκη
#print(#1) #; //εντολή
```

[3] Όπως θα δούμε παρακάτω τα επιμέρους τμήματα δεν είναι πάντα ΑΔΚ, μπορεί να είναι και σταθερές.

Αν προχωρήσουμε τα επιμέρους τμήματα του `I#4 #== #0` είναι τα `I#4` και `#0` έχουν ως επιμέρους τμήματα τα 4 και 0 αντίστοιχα. ενώ η εντολή (`#print(#1)`) έχει ως επιμέρους τμήμα το `#1`.

Όπως έχουμε δει παραπάνω τα επιμέρους τμήματα αποθηκεύονται σε διαφορετικά ΑΔΚ και απλά εκτελούνται μέσω στο κώδικα του "ΑΔΚ-γονέα" τους. Ακόμη είδαμε ότι κάθε ΑΔΚ μπορεί να εκτελεστεί αυτοτελές. Άρα το μόνο που έχουμε να κάνουμε για να μπορούμε να αναλύουμε τον δυναμικό κώδικα στα επιμέρους του στοιχεία και να έχουμε πρόσβαση σε αυτά είναι να γνωρίζουμε τις διευθύνσεις τους. Από κει και πέρα μπορούμε να τα χρησιμοποιήσουμε αυτοτελώς. Ας θεωρήσουμε την παρακάτω εντολή match:

```
match (m)
{
        case x1 #== x2:
        {
  …
        }
}
```

Για να την εκτελέσουμε θα πρέπει (αφού βεβαιωθούμε ότι η m αντιστοιχεί στον τελεστή σύγκρισης ==) να βρούμε τις διευθύνσεις των επιμέρους στοιχείων του ΑΔΚ της m και να τα αναθέσουμε στις νέες μεταβλητές `x1,x2`.

Για να εκτελέσουμε αυτές τις διεργασίες χρησιμοποιούμε τον πίνακα εξαρτήσεων. Αυτός κατασκευάζεται στο χρόνο εκτέλεσης μαζί με τον κώδικα. Στα προηγούμενα έχει αναφερθεί ότι στο χρόνο εκτέλεσης τοποθετούνται τα απαραίτητα δεδομένα στις σχισμές του προτύπου μετά την αντιγραφή. Ακριβώς αυτά τα δεδομένα είναι οι διευθύνσεις των επιμέρους τμημάτων μιας δομής (ή οι σταθερές) δηλαδή αυτά ακριβώς που πρέπει να λάβουμε με την ανάλυση του κώδικα. Έτσι τα δεδομένα αυτά αποθηκεύονται σε έναν πίνακα μετά τον κώδικα στο ΑΔΚ(πίνακας εξαρτήσεων), από όπου κατά τη φάση της ανάλυσης ανακτώνται και ανατίθενται στις νέες μεταβλητές. Για τη σωστή ανάθεση εγγυάται ο έλεγχος συμφωνίας της δομής υψηλού επιπέδου αφού ο τύπος και σειρά των επιμέρους τμημάτων είναι ίδια για κάθε δομή υψηλού επιπέδου.

Η όλη διαδικασία ανάλυσης έχει πολυπλοκότητα O(n) ως προς τον αριθμό των επιμέρους τμημάτων αλλά μπορούμε να θεωρήσουμε (καθώς ο αριθμός αυτός είναι σταθερός για κάθε δομή υψηλού επιπέδου) ότι έχουμε πολυπλοκότητα O(1) ως προς το μήκος του κώδικα για ένα στάδιο ανάλυσης. Έτσι μια αναδρομική διαδικασία που θα ανέλυε κώδικα πλήρως θα είχε πολυπλοκότητα O(n) ως προς το μήκος του κώδικα υπό ανάλυση. Μια τέτοια συνάρτηση γραμμένη στη γλώσσα παρατίθεται στα παραδείγματα.

3.4.8. Διαχείριση των ΑΔΚ - Καταστροφή

Τα ΑΔΚ είναι αντικείμενα που δημιουργούνται στο σωρό δυναμικά διαφανώς προς τον προγραμματιστή. Μια μεταβλητή codeof είναι ένας δείκτης προς ένα ΑΔΚ. Έτσι η ανάθεση μιας μεταβλητής τύπου codeof δεν επιφέρει αντιγραφή του ΑΔΚ αλλά μόνο της διεύθυνσης του. Παρόλα αυτά αυτό δεν δημιουργεί κάποιο πρόβλημα, δηλαδή μπορεί ο χρήστης να προγραμματίζει σαν να γινόταν αντιγραφή του ΑΔΚ.

Το επόμενο ζήτημα είναι η καταστροφή των ΑΔΚ. Στο τέλος κάθε συνάρτησης καταστρέφονται αυτόματα όλες οι μεταβλητές (προσωρινές και μη) καθώς και οι παράμετροι τύπου codeof. Δεν καταστρέφονται οι μεταβλητές που δημιουργούνται στα match,typematch. Για να μπορεί να γίνεται σωστά η καταστροφή εφαρμόζουμε το μηχανισμό των μετρητών αναφοράς. Κάθε ΑΔΚ έχει ένα μετρητή αναφοράς ο οποίος ξεκινά με την τιμή 1. Κάθε φορά που καλείται η διαδικασία καταστροφής ο μετρητής μειώνεται κατά 1 και αν φθάσει 0 τότε η μνήμη που καταλαμβάνει το αντικείμενο ελευθερώνεται. Επίσης για να επιτευχθεί ισορροπία με τις ακόλουθες πράξεις έχουμε αύξηση του μετρητή κατά 1:

- ανάθεση
- πέρασμα ως παράμετρο

3.5. Σημασιολογική Ανάλυση

3.5.1. Γενικά

Η σημασιολογική ανάλυση γίνεται, όπως άλλωστε και όλη η μεταγλώττιση, σε συγχρονισμό με τη συντακτική ανάλυση του προγράμματος από το Bison και στο πλαίσιο των σημασιολογικών ρουτινών.

Παρακάτω θα παρουσιάσουμε τον σημασιολογικό αναλυτή καθώς και τον πίνακα συμβόλων τόσο για το στατικό κομμάτι του μεταγλωττιστή όσο και για το δυναμικό με έμφαση στο τελευταίο.

Στις ενότητες 2-3 θα παρουσιαστούν η ανάλυση στατικού και δυναμικού κώδικα ενώ στις επόμενές τους θα παρουσιαστούν ειδικά θέματα σημασιολογικών ελέγχων δυναμικού κώδικα.

3.5.2. Σημασιολογική Ανάλυση Στατικού Κώδικα

Σε αυτή την ενότητα δεν θα προχωρήσουμε σε αναλυτική περιγραφή της σημασιολογικής ανάλυσης και των μεθόδων που χρησιμοποιήσαμε αλλά θα δώσουμε

τα στοιχεία που χρειάζονται για να κατανοηθεί η σημασιολογική ανάλυση του δυναμικού κώδικα. Λεπτομέρειες μπορούν να αναζητηθούν στη βιβλιογραφία.

Αρμόδια για τη σημασιολογία στατικού κώδικα είναι το αρχείο semantics.cpp καθώς και κάποια σημεία του bison.txt.

Βασική δομή της σημασιολογίας είναι η έκφραση. Για την εσωτερική αναπαράσταση της έκφρασης χρησιμοποιούμε τον τύπο SVar. Η δήλωση του τύπου αυτού γίνεται ως ακολούθως:

```
struct SVar
{
  SymbolEntry * pPlace;
  bool IsLValue;
  int ref;
};
```

- Το pPlace είναι δείκτης στην καταχώρηση του πίνακα συμβόλων που αντιστοιχεί στην έκφραση
- Το IsLValue δηλώνει το αν η μεταβλητή είναι LValue ή όχι.
- Το ref δηλώνει το αν η μεταβλητή είναι αναφορά ή αποδεικτοδότηση στο pPlace. Βέβαια η γλώσσα δεν υποστηρίζει pointers αλλά εσωτερικά χρησιμοποιούνται για τις δυναμικές μεταβλητές.

3.5.3. Σημασιολογική Ανάλυση Δυναμικού Κώδικα

Επιχειρώντας να αναλύσουμε σημασιολογικά (δηλ. να εντοπίσουμε τυχόν σημασιολογικά σφάλματα) τα τμήματα δυναμικού κώδικα ερχόμαστε αντιμέτωποι με νέα ζητήματα τα οποία πρέπει και να λύσουμε και μάλιστα με σχετικά αποδοτικό τρόπο.

Όπως αναφέρθηκε στο κεφάλαιο το σύστημα τύπων για τον δυναμικό κώδικα είναι συνεκτικό και υπάρχει μία 1-1 αντιστοιχία με τον στατικό κώδικα. Δηλαδή ο έλεγχος τύπων για μια δομή είναι ακριβώς ο ίδιος που πρέπει να γίνει και για τον στατικό κώδικα αν αντικατασταθούν οι τύποι δυναμικού κώδικα των τελουμένων από τους στατικούς τύπους αναφοράς των. πχ. ο έλεγχος τύπων για τον τελεστή #== είναι ο ίδιος με αυτόν για τον == με τη διαφορά ότι πρέπει τα δυναμικά ορίσματα του #== να αντικατασταθούν από στατικά. Επίσης δεν επιτρέπεται (βλ. κεφ 2) η χρησιμοποίηση δυναμικών τελεστών με στατικά ορίσματα.

Στο παραπάνω δεδομένο μπορούμε να προσθέσουμε αυτά που γνωρίζουμε από τα προηγουμένως ότι ο παραγόμενος κώδικας των δυναμικών δομών σε πολλές περιπτώσεις είναι ο ίδιος με των στατικών με προστιθέμενο στην αρχή ένα στάδιο εκτέλεσης των δυναμικών δεδομένων. Στις δομές αυτές ελέγχουμε αρχικά αν όλα τα

ορίσματα είναι δυναμικά, παράγουμε μετά τα στατικά αποτελέσματα της εκτέλεσης των δυναμικών ορισμάτων, και τέλος καλούμε τον σημασιολογικό ελεγκτή του στατικού κώδικα για τον περαιτέρω έλεγχο.

Οι δομές αυτές είναι:

- Τελεστές(εκτός του `#;`)
- Κλήση Συνάρτησης

Στις άλλες δομές ο έλεγχος τύπων γίνεται ρητά. Οι δομές αυτές είναι:

- if
- for
- Τελεστής `#;`
- Εκτέλεση Δυναμικού Κώδικα (exec)
- Καταστροφή Δυναμικού Κώδικα (destroy)
- Σταθερή Δυναμικού Κώδικα (π.χ. `#i,#4`)
- Δυναμική Μεταβλητή (π.χ. `I#4`)

Οι σημασιολογικές ρουτίνες δυναμικού κώδικα βρίσκονται όλες στο αρχείο dyn_semantics.cpp με εξαίρεση μερικές βοηθητικές λειτουργίες για το match. Παρακάτω παρουσιάζονται οι σημασιολογικές ρουτίνες που πραγματοποιούν ελέγχους και ποιες δομές ελέγχουν.

Πίνακας Συναρτήσεων Σημασιολογικής Ανάλυσης Δυναμικού Κώδικα	
Συνάρτηση	**Δομή που αντιστοιχεί**
`dm_null_cmd`	Τελεστής `#;` (Χωρίς όρισμα)
`dm_const_id`	Σταθερή Δυναμικού Κώδικα (π.χ. `#i,#4`)
`dm_new_dynamic_id`	Δυναμική Μεταβλητή (π.χ. `I#4`)
`dm_realize_op`	Τελεστές
`dm_call_function`	Κλήση Συνάρτησης
`dm_exec_code`	Εκτέλεση Δυναμικού Κώδικα
`dm_if`	IF
`dm_for`	FOR
`dm_destroy_code`	destroy
`dm_new_match_pattern`	Χρησιμοποιείται το match

Πίνακας 3-5

3.5.4. Σύστημα Τύπων – Εσωτερικοί δείκτες

Έχουμε δει στο κεφάλαιο περιγραφής της γλώσσας ότι η γλώσσα μας δεν υποστηρίζει δείκτες. Παρόλα αυτά εσωτερικά στο μεταγλωττιστεί υπάρχει τύπος δείκτη

τον οποίο χρησιμοποιούμε για να αποθηκεύσουμε δεδομένα που μας χρειάζονται όπως διευθύνσεις συναρτήσεων και προτύπων. Αυτή είναι μια προσθήκη για διευκόλυνση διαφανής στο χρήστη.

Στα επόμενα όπου χρειάζεται θα συμβολίζουμε τους δείκτες όπως στη C. πχ.:

```
int * v //Δείκτης σε ακέραιο
```

3.5.5. Δομές match - typematch

Η σημασιολογική ανάλυση των δομών match και typematch παρουσιάζουν αρκετό ενδιαφέρον και κάποια θέματα.

Η δομή typematch είναι η απλούστερη από τις δύο δομές. Παίρνει ένα αντικείμενο δυναμικού κώδικα και έπειτα ο προγραμματιστής μπορεί να καθορίσει ενέργειες ανάλογα με τον τύπο του αντικειμένου. Μπορεί να δεχθεί οποιοδήποτε δυναμικό τυπο και έτσι ο μόνος σημασιολογικός έλεγχος που γίνεται είναι εάν η έκφραση προς εξέταση είναι όντως δυναμική.

Η δομή match παρουσιάζει μια ιδιαιτερότητα καθώς έχει τη δυνατότητα να ορίζει ad hoc μεταβλητές που μάλιστα ο τύπος τους πρέπει να ορίζεται από τα συμφραζόμενα. Εδώ ακολουθείται η εξής διαδικασία: Ελέγχεται αν το αναγνωριστικό υπάρχει, αν υπάρχει έχουμε σφάλμα, αν όχι δηλώνεται μια νέα μεταβλητή και της ανατίθεται ο τύπος που προκύπτει από τη συντακτική της θέση.

Το θέμα είναι ότι σε μερικές περιπτώσεις δεν μπορούμε να συνάγουμε με ασφάλεια τον τύπο, της προς δημιουργία μεταβλητής. πχ. έχουμε τον εξής κώδικα:

```
match (c)
{
        case x1 #= x2:
        {
        }
        case x12 #== x22:
        {
        }
        case otherwise:
        {
                writeChar('-');
        }
}
```

Και στα δύο case δεν μπορούμε να ξέρουμε θετικά τον τύπο της κάθε παραμέτρου. Καθώς θα ήταν πολύ περιοριστικό για τον προγραμματιστή να δήλωνε και τον τύπο στη match θεωρούμε τις μεταβλητές αυτές ως codeof void.

3.5.6. DL-Values

Ένα από τα ζητήματα στην υλοποίηση των δυναμικών μεταβλητών αφορά τα l-values και τα r-values. Στον παρόν μεταγλωττιστή πρέπει να ξέρουμε όχι μόνο αν μια έκφραση είναι l-value με την έννοια ότι μπορεί να βρεθεί στα αριστερά των στατικών τελεστών ανάθεσης (`=`, `+=`, `++`) αλλά και αν μπορεί να βρεθεί στα αριστερά των δυναμικών τελεστών ανάθεσης (`#=`, `#+=`, `#++`). Θα ονομάσουμε τις εκφράσεις αυτές dl-values(dynamic l-values).

Το πρώτο που παρατηρούμε είναι ότι δεν υπάρχει σχέση μεταξύ της ιδιότητας του l-value και του dl-value, δηλαδή μπορεί μια μεταβλητή να είναι l-value χωρίς να είναι dl-value και αντιστρόφως, όπως μπορεί να είναι και τα 2 ή κανένα πχ.:

- το `I#3` είναι dl-value χωρίς να είναι l-value
- το `m = #3` είναι l-value χωρίς να είναι dl-value
- το `m = I#3` είναι και l-value και dl-value
- το `I#4 #== #0` δεν είναι ούτε l-value ούτε dl-value

Την τήρηση του LValue την αναλαμβάνει το πεδίο IsLValue της δομής SVar. Η τιμή του υπολογίζεται με την αποτίμηση της έκφρασης.

Αρχικά μπορούμε να σκεφθούμε ότι το ίδιο μπορεί να γίνει και με το dl-value. Όμως εδώ υπάρχει μια διαφορά το dl-value δεν αποτελεί χαρακτηριστικό της έκφρασης αλλά του ίδιου του αντικειμένου, που όμως είναι γνωστό κατά τη διάρκεια της μεταγλώττισης άρα δεν χρειάζεται να αποθηκευθεί στο αντικείμενο. Έτσι αποθηκεύεται στην καταχώρηση του πίνακα συμβόλων για το αντικείμενο.

3.5.7. Πίνακας Συμβόλων

Ο Πίνακας Συμβόλων έχει την αρμοδιότητα να κρατά πληροφορίες για τα σύμβολα (δηλ. τα ονόματα που εμφανίζονται στο πρόγραμμα) ώστε αυτά να συμπληρωθούν από το μεταγλωττιστή. Τα σύμβολα αυτά μπορεί να είναι μεταβλητές, συναρτήσεις, παράμετροι συναρτήσεων, σταθερές, τύποι δεδομένων κ.τ.λ. Με αυτόν τον ορισμό ο πίνακας συμβόλων έχει την ευθύνη διαχείρισης της εμβέλειας και της ορατότητας.

Ο πίνακας συμβόλων που χρησιμοποιήσαμε χρησιμοποιεί πίνακα κατακερματισμού (hash-table) για να αποθηκεύσει τις πληροφορίες συμβόλων και υποστηρίζει πολλαπλές εμβέλειες. Οι εμβέλειες δημιουργούνται και καταστρέφονται κατά τη μεταγλώττιση των ενοτήτων του προγράμματος.

Δεν θα παραθέσουμε λεπτομέρειες υλοποίησης πίνακα συμβόλων για τη στατική γλώσσα. Πληρέστερες πληροφορίες μπορούν να αναζητηθούν στη βιβλιογραφία.

Ο νέος τύπος CODEOF αναπαρίσταται στον πίνακα συμβόλων όπως ο τύπος δείκτη

δηλαδή έχοντας ένα τύπο αναφοράς. Η μόνη διαφορά βρίσκεται στο ότι από το συντακτικό δεν επιτρέπονται κυκλικές αναφορές άρα έχουμε μόνό βάθος αναφοράς.

Η άλλη δυναμική προσθήκη στον πίνακα συμβόλων είναι η προσθήκη του πεδίου IsDLValue το οποίο δηλώνει αν το αντικείμενο (μεταβλητή ή προσωρινή μεταβλητή) είναι dl-value, σύμφωνα με τα παραπάνω.

3.6. Παραγωγή Ενδιάμεσου Κώδικα

3.6.1. Γενικά

Η παραγωγή ενδιάμεσου κώδικα είναι το κομμάτι του μεταγλωττιστή που είναι επιφορτισμένο με το μεγαλύτερο όγκο εργασιών σχετικά με τις δυναμικές επεκτάσεις και γι΄αυτό θα αναλυθεί διεξοδικά στην ενότητα αυτή.

Ο ενδιάμεσος κώδικας παράγεται και αυτός σε συγχρονισμό με τη συντακτική ανάλυση του προγράμματος από το Bison και στο πλαίσιο των σημασιολογικών ρουτινών.

Η παραγωγή ενδιάμεσου κώδικα γίνεται από τα εξής:

- Το αρχείο bison.txt στο οποίο έχει γραφτεί μέρος των εντολών παραγωγής που δεν ήταν συμφέρον να υλοποιηθούν με ξεχωριστές συναρτήσεις.
- Τα αρχεία semantics.cpp, dyn_semantics.cpp που περιέχουν τις σημασιολογικές ρουτίνες και της εντολές παραγωγής ενδιάμεσου κώδικα για στατικό και δυναμικό κώδικα αντίστοιχα.
- Τις δομές `SQuad`, `SLabel`, `SQOperand` που αναπαριστούν την τετράδα, την ετικέττα και το όρισμα τετράδας αντίστοιχα.
- Την κλάση CQuads που αναλαμβάνει την αναπαράσταση μιας ενότητας προγράμματος ενδιάμεσου κώδικα καθώς και τη δημιουργία νεών τετράδων και την εκτέλεση βασικών λειτουργιών σ' αυτές.
- Την κλάση CPattern που αναπαριστά ένα ΠΔΚ και κληρονομεί την CQuads.

Στα παρακάτω θα αναλύσουμε τη λειτουργία των παραπάνω και θα περιγράψουμε αναλυτικά την παραγωγή του δυναμικού κώδικα. Και πάλι δεν θα σταθούμε στην υλοποίηση στατικών εντολών και θα παραθέσουμε μόνο όσα στοιχεία είναι απαραίτητα για την κατανόηση της λειτουργίας του γεννήτορα σε σχέση με τον δυναμικό κώδικα.

3.6.2. Ενδιάμεση γλώσσα

Σαν ενδιάμεση γλώσσα θα χρησιμοποιηθεί η γλώσσα των τετράδων. Σε αυτή ένα πρόγραμμα αποτελείται από ένα σύνολο τετράδων. Η κάθε τετράδα έχει τη μορφή

```
operator,op1,op2,op3
```

Όπου operator ένας τελεστής και op1,op2,op3 τελούμενα ανάλογα με τον τελεστή. Σημειώνουμε ότι ένα τελούμενο μπορεί να είναι και κενό ή ο ειδικός τελεστής αποτελέσματος συνάρτησης.

Παρακάτω παραθέτουμε έναν πίνακα των τελεστών με μια συνοπτική περιγραφή:

Τελεστής	**Σημασία**
UNIT,_,_,_	Έναρξη Ενότητας Προγράμματος
ENDU,_,_,_	Τέλος Ενότητας Προγράμματος
ARRAY,_,_,_	
IFB,x,_,z	Αν είναι αληθής η λογική μεταβλητή x εκτελείται άλμα στην τετράδα z.
JUMP,_,_,z	Εκτελείται άλμα στην τετράδα z.
CALL,x,_,z	Καλείται το υποπρόγραμμα z με συνολικό μέγεθος παραμέτρων x.
PAR,x,y,_	Τοποθετείται η παράμετρος x με τρόπο y (με y=BYVAL,BYREF)
RET,_,_,_	Επιστροφή από συνάρτηση
:=,x,_,z	Ανατίθεται η έκφραση x στο z.
=,x,y,z	Αν x=y τότε ο έλεγχος μεταβιβάζεται στην τετράδα z.
<>,x,y,z	Ομοίως.
>,x,y,z	Ομοίως.
<,x,y,z	Ομοίως.
>=,x,y,z	Ομοίως.
<=,x,y,z	Ομοίως.
+,x,y,z	z := x+y
-,x,y,z	z := x-y
*,x,y,z	z := xy
/,x,y,z	z := x/y
%,x,y,z	z := x modulo y
^,x,_,z	z := ¬x
GETFP,x,_,z	Η διεύθυνση της συνάρτησης x ανατίθεται στη μεταβλητή z.
EXEC,x,y,_	Εκτελείται η ο δυναμικός κώδικας x και αποθήκευση του αποτελέσματος στο y. Το y μπορεί να είναι κενό(_) αν δεν χρειαζόμαστε αποτέλεσμα.
PATTERN,_,_,x	Αρχή προτύπου δυναμικού κώδικα x
ENDP,_,_,_	Τέλος προτύπου δυναμικού κώδικα
DRET,x,_,_	Επιστροφή από δυναμική μονάδα κώδικα. Επιστρέφεται το x
DVAR,x,y,z	Η διεύθυνση της δυναμικής μεταβλητής τύπου x αρ. y αποθηκεύεται στο z.
GETP,x,_,z	Η διεύθυνση του προτύπου x αποθηκεύεται στο z.

Πίνακας 3-6

Βλέπουμε ότι οι ενδιάμεσες εντολές που προστέθηκαν είναι ελάχιστες και οι πιο πολλές είναι βοηθητικές.

3.6.3. Αναπαράσταση Τετράδων - Ενοτήτων

Η αναπαράσταση των τετράδων γίνεται με τη βοήθεια των δομών `SQuad`, `SQOperand` των οποίον οι δηλώσεις παρατίθονται σχολιασμένες στη συνέχεια.

```
struct SQOperand
{
  EType eType;       //Τύπος Ορίσματος
  ERef eRef;               //Αναφορά / Αποδεικτοδότηση
  void * param;            //Ανάλογα με τον τύπο
};

bool operator==(SQOperand &op1,SQOperand &op2);

struct SQuad
{
  int num;                 //Αριθμός Τετράδας
  WORD Oper;               //Τελεστής
  SQOperand * ops[3];      //Ορίσματα
  ~SQuad();                //Κατασκευαστής
};
```

Παρ' όλο που το όρισμα μιας τετράδας μπορεί να αναπαριστά πολλά διαφορετικά πράγματα (π.χ. μεταβλητές, διευθύνσεις συναρτήσεων, έχουμε μια ενιαία δομή για όλα. Η ουσιαστική πληροφορία του ορίσματος περνά μέσα από το στοιχείο `param` του οποίου η σημασία αλλάζει ανάλογα με τον τύπο του ορίσματος. Η δήλωση του `SQuad` είναι επεξηγηματική.

Η κλάση `CQuads` παρέχει:

- Στατικές Μεθόδους για τη εύκολη δημιουργία δομών `SQuad` (δηλώνονται στο quads.h και υλοποιούνται στο newquads.cpp).
- Στατικές Μεθόδους για τη εύκολη δημιουργία νέων δομών `SQOperand`.
- Μεθόδους για τις λειτουργίες πάνω στην ενότητα ή στις τετράδες όπως:
  - Εκτύπωση Ενότητας σε Αρχείο
  - Διαγραφή Τετράδας

- Δημιουργία
- Αναδρομική Αντικατάσταση (BackPatch)

3.6.4. Σημασιολογικές Ρουτίνες Παραγωγής Ενδιάμεσου Κώδικα

Στον παρακάτω πίνακα παρατίθενται οι ρουτίνες Παραγωγής Ενδιάμεσου Κώδικα που ορίζονται στο αρχείο dyn_semactics.cpp και αφορούν τις δυναμικές δομές:

Πίνακας Συναρτήσεων Σημασιολογικής Ανάλυσης Δυναμικού Κώδικα	
Συνάρτηση	**Δομή που αντιστοιχεί**
`dm_null_cmd`	Τελεστής `#;` (Χωρίς όρισμα)
`dm_const_id`	Σταθερή Δυναμικού Κώδικα (π.χ. `#i,#4`)
`dm_new_dynamic_id`	Δυναμική Μεταβλητή (π.χ. `I#4`)
`dm_realize_op`	Τελεστές
`dm_call_function`	Κλήση Συνάρτησης
`dm_exec_code`	Εκτέλεση Δυναμικού Κώδικα
`dm_if`	IF
`dm_for`	FOR
`dm_destroy_code`	destroy
`dm_new_match_pattern`	Χρησιμοποιείται στο match
`dm_match_case`	case (match ή typematch)
`dm_match`	match ή typematch
`NewPattern`	Δημιουργία νέου προτύπου (βοηθητική)
`EndPattern`	Εκτέλεση τυποποιημένων διαδικασιών σε πρότυπο μετά την ολοκλήρωση παραγωγής του ενδ. κώδικά του(βοηθητική).

Πίνακας 3-7

Παρατηρούμε, από τη σύγκριση των πινάκων Πίνακας 3-5, Πίνακας 3-7 ότι οι ρουτίνες παραγωγής ενδιάμεσου κώδικα εκτελούν και σημασιολογικό έλεγχο.

3.6.5. Συναρτήσεις Βιβλιοθήκης – Υποστήριξη Ανάστροφου Περάσματος Παραμέτρων

Για την υλοποίηση των λειτουργιών δυναμικού κώδικα που τελούνται στο χρόνο εκτέλεσης έχουμε κάποιες συναρτήσεις βιβλιοθήκης που καλούνται από τα προγράμματα διαφανώς προς το χρήστη. Οι συναρτήσεις αυτές χρησιμοποιούν το μοντέλο κλήσης συναρτήσεων της γλώσσας μας μόνο που το πέρασμα παραμέτρων σε αυτές γίνεται ανάποδα, δηλαδή από το τέλος προς την αρχή (αυτό γίνεται για να γίνει εύκολη η ανάγνωση μεταβλητού αριθμού παραμέτρων όπως απαιτείται από ορισμένες συναρτήσεις βιβλιοθήκης.

Στα παρακάτω περιγράφονται οι συναρτήσεις βιβλιοθήκης που χρησιμοποιούνται εσωτερικά ως προς το έργο που επιτελούν, τις παραμέτρους που παίρνουν και το τι επιστρέφουν. Η εσωτερική λειτουργία τους θα αναλυθεί στο κεφάλαιο για τον τελικό

κώδικα.

3.6.5.1. makecode

Η συνάρτηση makecode εκτελεί ουσιαστικά την κατασκευή του ΑΔΚ από το πρότυπο και τα επιμέρους τμήματα. Συγκεκριμένα:

1. Αντιγράφεται ο κώδικας του προτύπου σε μια νέα ΑΔΚ.
2. Τοποθετούνται τα επιμέρους τμήματα στις σχισμές στο αντίγραφο του κώδικα.
3. Κατασκευάζεται ο πίνακας εξαρτήσεων (βλ. 3.4.7)

Η δήλωση της makecode είναι η εξής:

```
codeof void * _makecode(void * pattern,DWORD type,void *
  slot1,..,void * slotn);
```

Οι παράμετροι αναλύονται ως εξής:

- `pattern`: Διεύθυνση προτύπου
- `type`: Είδος δομής που θα κατασκευαστεί (βλ. 3.4.7). Σημειώνουμε ότι επειδή η παράμετρος έχει μήκος 4 bytes περνιέται σαν δύο ακεραίους των 2 bytes.
- `slot1,slot2,…, slotn`: Τα δεδομένα που θα τοποθετηθούν στις σχισμές με τη σειρά δήλωσης των παραμέτρων να αντιστοιχεί στη σειρά εμφάνισης των σχισμών στον πίνακα σχισμών του προτύπου.

3.6.5.2. addref

Η συνάρτηση addref αυξάνει το μετρητή αναφορών ενός ΑΔΚ κατά 1. Η δήλωσή της είναι η εξής:

```
void _addref(codeof void c)
```

3.6.5.3. gettype

Η συνάρτηση gettype δίνει το είδος της δομής υψηλού επιπέδου που αναπαριστά μια δομή κώδικα. Η δήλωσή της είναι η εξής:

```
void _gettype(codeof void c,int * loword,int * hiword)
```

Οι παράμετροι αναλύονται ως εξής:

- c: ΑΔΚ
- loword: Δείκτης στον ακέραιο που θα παραλάβει τη λιγότερο σημαντική λέξη του είδους δομής

– hiword: Δείκτης στον ακέραιο που θα παραλάβει τη περισσότερο σημαντική λέξη του είδους δομής

3.6.5.4. getdeps

Η συνάρτηση getdeps αναθέτει τις εξαρτήσεις ενός ΑΔΚ σε μεταβλητές. Η δήλωση της συνάρτησης είναι η εξής:

```
void  _getdeps(codeof void c,word * dep1,word * dep2,...,word
  * depn)
```

Οι εξαρτήσεις ανατίθενται με τη σειρά που εμφανίζονται στον πίνακα εξαρτήσεων του ΑΔΚ στις θέσεις που δείχνουν οι δείκτες `dep1,dep2,…,depn`.

3.6.5.5. destroy

Η συνάρτηση destroy εκτελεί την "καταστροφή" ενός ΑΔΚ. Η δήλωσή της είναι η ακόλουθη:

```
void * _destroy(codeof void c)
```

Μειώνεται ο μετρητής αναφορών και εκτελείται η καταστροφή του αντικειμένου αν αυτός φθάσει το 0.

3.6.5.6. new - dispose

Είναι συναρτήσεις δέσμευσης – αποδέσμευσης χώρου στο σωρό. Δεν καλούνται άμεσα από τα προγράμματα αλλά από άλλες συναρτήσεις βιβλιοθήκης.

Οι δηλώσεις τους κάνουν προφανή τη χρήση τους και ακολουθούν:

```
void _dispose (void * pointer)
void * _new (int size)
```

3.6.6. Παραγωγή Ενδιάμεσου Κώδικα Για Τις Δυναμικές Δομές Υψηλού Επιπέδου - Γενικά

Όπως αναφέρθηκε στην ενότητα 3.4 όταν ο μεταγλωττιστής συναντάει μια δυναμική δομή υψηλού επιπέδου, παράγει 2 πράγματα:

1. Τον ενδιάμεσο κώδικα του ΠΔΚ για τη δομή
2. Τον κώδικα που θα εκτελεστεί και θα κατασκευάσει το ΑΔΚ με βάση το πρότυπο και ακολουθώντας τα βήματα που περιγράφηκαν στην ενότητα 3.4.

Κάθε ένα από αυτά τα βήματα καθώς και τα βοηθητικά που χρησιμοποιούμε θα αναλυθεί στις επόμενες ενότητες.

3.6.7. Παραγωγή Ενδιάμεσου Κώδικα ΠΔΚ - Γενικά

Όπως είδαμε τα πρότυπα αποτελούν, από τη σκοπιά του ενδιάμεσου κώδικα, ενότητες τετράδων. Η κατασκευή τους ακολουθεί, ανάλογα με τη δομή διαφορετικούς δρόμους.

Η ιδιαιτερότητα των προτύπων είναι ότι μερικές από τις τετράδες τους έχουν κενές θέσεις ορισμάτων που κανονικά έπρεπε να είναι γεμάτες. Αυτό γίνεται επειδή τα δεδομένα εκτέλεσης (όπως έχει αναφερθεί) δεν είναι γνωστά κατά την παραγωγή του ενδιάμεσου κώδικα των ΠΔΚ (στη μεταγλώττιση) και συμπληρώνονται μετά στον τελικό κώδικα. Ο γεννήτορας τελικού κώδικα φροντίζει να ερμηνεύει σωστά αυτές τις κενές τετράδες.

Η αναπαράσταση των προτύπων γίνεται με την κλάση CPattern. Η κλάση αυτή κληρονομεί την CQuads για να αναπαριστά τον ενδιάμεσο του προτύπου. Εκτός από αυτά η κλάση έχει δομές για να υποστηρίζουν τα επιπλέον χαρακτηριστικά του προτύπου(π.χ. σχισμές) και να κατασκευάζουν τελικό κώδικα.

Η διαχείριση των σχισμών γίνεται από την CPatterns ως εξής: Κατά την αρχικοποίηση της κλάσης ορίζεται ο αριθμός των σχισμών του προτύπου ο οποίος είναι γνωστός κατά το στάδιο της μεταγλώττισης και δίνεται από τις σημασιολογικές ρουτίνες που ασχολούνται με την παραγωγή κώδικα προτύπων. Έπειτα η σειρά των σχισμών καθορίζεται με βάση τη σειρά εμφάνισης των κενών στον ενδιάμεσο κώδικα. Η κατασκευή τελικού κώδικα προτύπων αναλύεται στην ενότητα 3.7.

Για τη μεγάλη πλειοψηφία των δομών η δομή του ενδιάμεσου κώδικα του προτύπου είναι:

```
EXEC,_,_,$1
EXEC,_,_,$2
…
EXEC,_,_,$n
[Αντίστοιχος Στατικός Κώδικας]
DRET,$(n+1),_,_
```

Δηλαδή πρώτα αποκομίζουμε τα στατικά δεδομένα εκτελώντας τις δυναμικές συνιστώσες, μετά εκτελούμε τον κώδικα που εκτελεί την ουσιαστική επεξεργασία και επιστρέφουμε το αποτέλεσμα. Στις δομές εκείνες που ακολουθούμε αυτό τον τρόπο παραγωγής ο "Αντίστοιχος Στατικός Κώδικας" παράγεται καλώντας την αντίστοιχη σημασιολογική ρουτίνα για στατικό κώδικα, όπως ακριβώς επιτυγχάνεται η σημασιολογική ανάλυση.

Παρατηρούμε ότι οι αρχικές εντολές EXEC δεν έχουν $1^{ο}$ όρισμα και αυτό γίνεται ακριβώς επειδή το όρισμα αυτό δεν είναι γνωστό κατά το χρόνο μεταγλώττισης.

Οι δομές που ακολουθούν τον παραπάνω τρόπο είναι οι εξής:

– Τελεστές(εκτός του `#;`)

– Κλήση Συνάρτησης

Για παράδειγμα ο ενδιάμεσος κώδικας του προτύπου

```
(I#3)#==#5
```

είναι ο (απόσπασμα προγράμματος):

```
157: PATTERN, pattern 2, _, _
158: EXEC, _, $9, _
159: EXEC, _, $10, _
160: =, [$9], $10, 163
161: :=, false, _, $11
162: JUMP, _, _, 164
163: :=, true, _, $11
164: DRET, $11, _, _
165: ENDP, pattern 2, _, _
```

Παρατηρούμε ότι στις γραμμές 158-159 τα δυναμικά δεδομένα (που στο πρότυπο είναι κενά) εκτελούνται και παράγονται οι κανονικές στατικές ακέραιες μεταβλητές $9,$10. Κατόπι ακολουθεί στις γραμμές 160-163 ο στατικός κώδικας που θα παραγόταν από το εξής:

```
$11 = $9==$10
```

Τέλος στη γραμμή 164 επιστρέφεται η $11 (τύπου boolean).

Τα πρότυπα των άλλων δομών παράγονται απευθείας χωρίς χρήση των ρουτίνων στατικού κώδικα. Οι δομές αυτές είναι:

– if και if-else

– for

– Τελεστής `#;`

– Σταθερή Δυναμικού Κώδικα (π.χ. `#i,#4`)

– Δυναμική Μεταβλητή (π.χ. `I#4`)

Παρατηρούμε ότι οι δύο διαφορετικές προσεγγίσεις σημασιολογίας και παραγωγής ενδιάμεσου κώδικα των ΠΔΚ αφορούν τις ίδιες δομές και είναι ουσιαστικά οι ίδιες προσεγγίσεις. Δηλαδή οι δυναμικές δομές υψηλού επιπέδου ουσιαστικά χωρίζονται σε αυτές που ο σημασιολογικός έλεγχος και

Δίνουμε για παράδειγμα τον ενδιάμεσο κώδικα που παράγει μια δομή if-else:

```
245: EXEC, _, $32, _
246: IFB, $32, _, 248
```

```
247: JUMP, _, _, 249
248: EXEC, _, _, _
249: EXEC, _, _, _
```

Η δομή παίρνει 3 ορίσματα(που στο στάδιο που βρισκόμαστε δεν είναι γνωστές οι τιμές τους):

1. Τη συνθήκη τύπου codeof bool.
2. Την εντολή που εκτελείται όταν ισχύει η συνθήκη (τύπου codeof void)
3. Την εντολή που εκτελείται όταν δεν ισχύει η συνθήκη (τύπου codeof void)

Εδώ αρχικά εκτελείται η συνθήκη (μη γνωστό όρισμα) και μετά ανάλογα με την τιμή της (true, false) ο έλεγχος μεταβιβάζεται στις εντολές 248, 249 που εκτελούν την παράμετρο 2 και 3 αντίστοιχα.

Στα επόμενα παρουσιάζεται το πλήρες σχέδιο παραγωγής για όλες τις δυναμικές δομές υψηλού επιπέδου.

3.6.8. Πλήρες Σχέδιο Παραγωγής Ενδιάμεσου Κώδικα ΠΔΚ

Παρακάτω θα δώσουμε το πλήρες σχέδιο παραγωγής για κάθε δομή. Για κάθε δομή θα αναφέρονται οι παράμετροι-σχισμές του ΠΔΚ, οι οποίες και θα προσδιορίζονται στον κώδικα με το σύμβολο $Si όπου i ο αριθμός σχισμής, και η σημασία τους. Δομές στις οποίες υπάρχει ομοιότητα θα εξεταστούν μαζί.

Δεν θα εξεταστούν η κλήση συνάρτησης και οι τελεστές καθώς το ακριβές τους σχέδιο έχει παρατεθεί προηγουμένως.

3.6.8.1. Σύμβολο #; (εντολή skip)

Τύπος: codeof void

Σχισμές:

Δεν έχει σχισμές

Κώδικας:

```
124: PATTERN, pattern 5, _, _
125: ENDP, pattern 5, _, _
```

3.6.8.2. Τελεστής #; (δύο ορισμάτων)

Τύπος: codeof void

Παραδείγματα:

```
(I#0 #= #1) #; (I#2 #= #0)

codeof void c,c1,c2;
c = c1 #; c2 ;
```

Σχισμές:

Σχισμή	Τύπος	Σημασία
$S1	codeof void	Κώδικας 1ης Εντολής
$S2	codeof void	Κώδικας 2ης Εντολής

Κώδικας:

```
124: PATTERN, pattern 5, _, _
125: EXEC, $S1, _, _
126: EXEC, $S2, _, _
125: ENDP, pattern 5, _, _
```

3.6.8.3. Δυναμική Σταθερά

Τύπος: codeof T (με T οποιονδήποτε βασικό τύπο)

Παραδείγματα:

```
#i
#4
```

Σχισμές:

Σχισμή	Τύπος	Σημασία
$S1	Τα	Σταθερά

Π 3-1

Κώδικας:

```
72: PATTERN, pattern 1, _, _
73: DRET, $S1, _, _
74: ENDP, pattern 1, _, _
```

3.6.8.4. Δυναμική Μεταβλητή

Τύπος: codeof T (με T οποιονδήποτε βασικό τύπο)

Παραδείγματα:

```
I#i
B#4
C#(4+2*i)
```

Σχισμές:

Σχισμή	Τύπος	Σημασία
$S1	int	Αρ. Δυναμικής Μεταβλητής

Π 3-2

Κώδικας:

```
61: PATTERN, pattern 0, _, _
62: DVAR, $S1, T, $4
63: DRET, $4, _, _
```

3.6.8.5. Δομή IF

Τύπος: codeof void

Παραδείγματα:

```
#if (I#0 #== #0) #writeInteger(I#4)
```

Σχισμές:

Σχισμή	Τύπος	Σημασία
$S1	codeof bool	Συνθήκη
$S2	codeof void	Εντολή αν η συνθήκη είναι αληθής

Π 3-3

Κώδικας:

```
132: PATTERN, pattern 6, _, _
133: EXEC, $S1 , $23, _
134: IFB, $23, _, 136
135: JUMP, _, _, 139
136: EXEC, $S2, _, _
139: ENDP, pattern 6, _, _
```

3.6.8.6. Δομή IF-ELSE

Τύπος: codeof void

Παραδείγματα:

```
#if (I#0 #== #0) #writeInteger(I#4) else #;
```

Σχισμές:

Σχισμή	Τύπος	Σημασία
$S1	codeof bool	Συνθήκη
$S2	codeof void	Εντολή αν η συνθήκη είναι αληθής
$S3	codeof void	Εντολή αν η συνθήκη είναι ψευδής

Π 3-4

Κώδικας:

```
132: PATTERN, pattern 6, _, _
133: EXEC, $S1 , $23, _
134: IFB, $23, _, 136
135: JUMP, _, _, 138
136: EXEC, $S2, _, _
137: JUMP, _, _, 139
138: EXEC, $S3, _, _
139: ENDP, pattern 6, _, _
```

3.6.8.7. Δομή FOR

Τύπος: codeof void

Παραδείγματα:

```
#for (I#0 #= #0; I#0 #< #10 ; I#0 #++) #writeInteger(I#0)
```

Σχισμές:

Σχισμή	Τύπος	Σημασία
$S1	codeof void	Αρχικοποίηση
$S2	codeof bool	Συνθήκη
$S3	codeof void	Επανάληψη
$S4	codeof void	Σώμα Εντολής

Π 3-5

Κώδικας:

```
257: PATTERN, pattern 16, _, _
258: EXEC, $S1, _, _
259: EXEC, $S2, $54, _
260: IFB, $54, _, 262
261: JUMP, _, _, 265
262: EXEC, $S4, _, _
263: EXEC, $S3, _, _
264: JUMP, _, _, 259
265: ENDP, pattern 16, _, _
```

3.6.9. Κώδικας Κατασκευής ΑΔΚ

Το 2^{ο} παραγόμενο μιας δυναμικής δομής υψηλού επιπέδου είναι ο κώδικας που κάνει την κατασκευή του ΑΔΚ όπως παρουσιάστηκε στην ενότητα 3.4. Ο κώδικας

αυτός είναι στο ίδιο μοτίβο ανεξαρτήτως της δομής. Ουσιαστικά επειδή την κατασκευή αναλαμβάνει η συνάρτηση βιβλιοθήκης makecode το μόνο που πρέπει να γίνει είναι να συγκεντρωθούν οι παράμετροι της makecode και να κληθεί η τελευταία.

Η δήλωση της makecode είναι η εξής:

```
codeof void * _makecode(void * pattern,DWORD type,void *
  slot1,..,void * slotn);
```

Η makecode δέχεται σαν παραμέτρους την διεύθυνση του ΠΔΚ, τον τύπο του κώδικα και ένα μεταβλητό αριθμό των εξαρτήσεων δηλαδή των διευθύνσεων των ΑΔΚ ή των σταθερών που θα τοποθετηθούν στις σχισμές και. Άρα ο ενδιάμεσος κώδικας θα πρέπει να κάνει τα εξής:

1. Ανεύρεση της διεύθυνσης του ΠΔΚ που αντιστοιχεί στη δομή και πέρασμα αυτής ως παραμέτρου (με την ενδιάμεση εντολή GETP).
2. Πέρασμα του τύπου ως παραμέτρου. Στην περίπτωση της δομής κλήσης συνάρτησης ο τύπος χρειάζεται να υπολογιστεί γιατί περιλαμβάνει τη διεύθυνση της καλούμενης συνάρτησης και δεν είναι γνωστός στο χρόνο μεταγλώττισης.
3. Πέρασμα των εξαρτήσεων
4. Πέρασμα της διεύθυνσης (προσωρινής) μεταβλητής στης οποία θα εκχωρηθεί το ΑΔΚ ως διεύθυνσης αποτελέσματος.

Για να κατανοηθεί η διαδικασία παραθέτουμε τον ενδιάμεσο κώδικα που παράγεται από το ακόλουθο τμήμα προγράμματος:

```
m = #writeInteger(I#4) ;
```

Ο ενδιάμεσος κώδικας ακολουθεί

```
140: GETP, pattern 0, _, $5
141: PAR, 4, V, _
142: PAR, 1, V, _
143: PAR, 260, V, _
144: PAR, $5, V, _
145: PAR, $6, RES, _
146: CALL, 8, _, _makecode

147: GETFP, writeInteger, _, $7
154: GETP, pattern 1, _, $9
155: PAR, $6, V, _
156: PAR, $7, V, _
157: PAR, 5, V, _
```

```
158: PAR, $9, V, _
159: PAR, $10, RES, _
160: CALL, 8, _, _makecode
161: :=, $10, _, m
```

Στις γραμμές 140-147 έχουμε τον κώδικα για την κατασκευή της I#4, στην οποία δεν θα επεκταθούμε, σημειώνουμε μόνο ότι το ΑΔΚ αποθηκεύεται στο $6.

Στις γραμμές 147-160 έχουμε τον κώδικα για την κατασκευή της συνολικής δομής και είναι:

- Στη γραμμή 147 μεταφέρουμε στην $7 τη διεύθυνση της `writeInteger`. (βήμα που ακολουθείται μόνο στη συγκεκριμένη δομή.
- Στις γραμμές 154,158 εκτελείται το 1ο βήμα και η διεύθυνση του προτύπου (pattern 1) ανατίθεται στην $9 και το πέρασμα της τελευταίας ως παραμέτρου
- Στις γραμμές 157-156 εκτελείται το βήμα 2 και περνούμε τον τύπο. Σημειώνουμε ότι ο τύπος περνιέται ως 2 παράμετροι καθώς έχει μέγεθος 4 bytes και στη γλώσσα μας δεν υπάρχει τέτοιος τύπος, οπότε θεωρούμε 2 ακεραίους των 2 bytes. Το 2ο τμήμα του τύπου είναι το $7, δηλαδή η διεύθυνση της `writeInteger` (βλ. ενότητα 3.4.7).
- Στη γραμμή 156 εκτελείται το βήμα 3 δηλαδή το πέρασμα των εξαρτήσεων. Εδώ η μόνη είναι η $6 δηλαδή το ΑΔΚ της παραμέτρου της συνάρτησης.

Σημειώνουμε ότι το πέρασμα των παραμέτρων γίνεται ανάποδα για λόγους που εξηγήθηκαν στην ενότητα 3.6.4.

3.6.10. Εκτέλεση Δυναμικού Κώδικα

Η εκτέλεση όπως ανεφέρθηκε στην ενότητα 3.4 είναι η πιο απλή λειτουργία ενδιάμεσου κώδικα και υλοποιείται απλά με μια εντολή ενδιάμεσου κώδικα:

```
EXEC,x,y,_
```

όπου x το δυναμικός κώδικας και y η μεταβλητή στην οποία θα αποθηκευθεί το αποτέλεσμα (πρέπει να είναι l-value).

3.6.11. Ανάλυση – Δομή match

Υλοποιούμε τη δομή match σε επίπεδο ενδιάμεσου κώδικα σαν μια σειρά από συνθήκες. Κάθε δομή if αντιστοιχεί σε κάθε case και αναλαμβάνει τον έλεγχο τύπου κώδικα (δηλ. είδος δομής υψηλού επιπέδου). Αν ικανοποιείται η συνθήκη τότε εκτελούνται οι εντολές που δίνονται στη case. Πριν όμως την εκτέλεση των εντολών παρεμβάλλεται η ανάθεση τιμών στις μεταβλητές που δηλώνονται στην case .

Για την εκτέλεση των παραπάνω λειτουργιών χρησιμοποιούμε 2 συναρτήσεις βιβλιοθήκης την gettype και την getdeps. Ο τρόπος χρήσης και οι εργασίες που επιτελούν έχουν αναλυθεί στα προηγούμενα. Η διαδικασία είναι η ακόλουθη:

1. Εκτελείται η gettype και το αποτέλεσμά της τοποθετείται σε προσωρινή μεταβλητή.
2. Για κάθε case ελέγχεται η τιμή που επιστράφηκε από την gettype και αν είναι ίση με αυτή του case (προφανώς επειδή το είδος λαμβάνεται ως 2 ακέραιοι θα χρειαστούν 2 συνθήκες) τότε
    i. Αναθέτονται τιμές στις μεταβλητές που ορίστηκαν στο case
    ii. Αυξάνεται ο μετρητής αναφορών κατά 1 στις νέες μεταβλητές
    iii. Εκτελούνται οι εντολές μέσα στο case
    iv. Ο έλεγχος μεταφέρεται στην επόμενη εντολή του match.

Σημειώνουμε ότι στο βήμα 2 ενδέχεται η ακριβής μορφή του case να μην είναι γνωστή κατά τη διάρκεια της μεταγλώττισης (κλήση συνάρτησης) τότε υπολογίζεται ακριβώς πριν τον έλεγχο της συνθήκης.

Θα δούμε τώρα ένα παράδειγμα ενδιάμεσου κώδικα για μια δομή match και θα αναλύσουμε τη διαδικασία κατασκευής. Έχουμε τη δομή:

```
match (c)
{
   case int const:         //Περίπτωση 1
   {
         writeChar('C');
   }
   case I # var_num:       //Περίπτωση 2
   {
         writeChar('V');
   }
   case otherwise:         //Περίπτωση 3
   {
         writeChar('-');
   }
}
```

Ο παραπάνω κώδικας ελέγχει αν ο κώδικας c αντιστοιχεί σε ακέραια σταθερά, ακέραια δυναμική μεταβλητή ή άλλο και τυπώνει αντίστοιχα C,V,-.

Ο ενδιάμεσος κώδικας που παράγεται ακολουθεί:

```
 2: PAR, {$2}, V, _
 3: PAR, {$1}, V, _
 4: PAR, c, V, _
 5: CALL, 6, _, _gettype
 6: =, 3, $1, 8
 7: JUMP, _, _, 16
 8: =, 0, $2, 10
 9: JUMP, _, _, 16
10: PAR, {const}, V, _
11: PAR, c, V, _
12: CALL, 4, _, _getdeps
13: PAR, 'C', V, _
14: CALL, 1, _, writeChar
15: JUMP, _, _, 29
16: =, 4, $1, 18
17: JUMP, _, _, 26
18: =, 1, $2, 20
19: JUMP, _, _, 26
20: PAR, {var_num}, V, _
21: PAR, c, V, _
22: CALL, 4, _, _getdeps
23: PAR, 'V', V, _
24: CALL, 1, _, writeChar
25: JUMP, _, _, 29
26: PAR, '-', V, _
27: CALL, 1, _, writeChar
28: JUMP, _, _, 29
29: …
```

Στον παρακάτω πίνακα φαίνεται η αντιστοίχιση ομάδων τετράδων με δομές. Αναλύεται εσωτερικά το 1° case. Σημειώνουμε ότι το case otherwise δεν έχει συνθήκη.

Τετράδες	**Λειτουργία**
2-5	Κλήση gettype. Αποθήκευση είδους στα $1,$2
6-15	Case 1
6-9	Έλεγχος Συμφωνίας Τύπου (βήμα 2)
10-12	Πέρασμα Παραμέτρων και κλήση getdeps για συμπλήρωσή μεταβλητής var_num που δηλώθηκε με το case (2.i)
13-14	Κώδικας case (2.ii)
15	Άλμα στο τέλος της δομής (2.iii)
16-25	Case 2
26-28	Case 3

Πίνακας 3-8

3.6.12. Ανάλυση – Δομή typematch

Η υλοποίηση της δομής typematch είναι όμοια με τη δομή match αλλά η ανάθεση των μεταβλητών γίνεται απευθείας (βλ. κεφ. 2). Επίσης κατά τον έλεγχο λαμβάνεται υπ' όψιν μόνο το 1° byte που επιστρέφεται από την gettype.

Παρακάτω δίνουμε μία τέτοια δομή και τον ενδιάμεσο κώδικα που αυτή παράγει:

```
int t;
typematch (c)
{
      case void : c_void : {t=0;}
      case int : c_int : {t=1;}
      case bool : c_bool : {t=2;}
      case char : c_char : {t=3;}
};
```

Ο ενδιάμεσος κώδικας που παράγεται είναι:

```
 2: PAR, 0, V, _
 3: PAR, {$1}, V, _
 4: PAR, c, V, _
 5: CALL, 6, _, _gettype
 6: =, 0, $1, 8
 7: JUMP, _, _, 11
 8: :=, c, _, c_void
 9: :=, 0, _, t
10: JUMP, _, _, 26
11: =, 256, $1, 13
12: JUMP, _, _, 16
13: :=, c, _, c_int
14: :=, 1, _, t
15: JUMP, _, _, 26
16: =, 512, $1, 18
17: JUMP, _, _, 21
18: :=, c, _, c_bool
19: :=, 2, _, t
20: JUMP, _, _, 26
21: =, 1024, $1, 23
22: JUMP, _, _, 26
23: :=, c, _, c_char
24: :=, 3, _, t
```

```
25: JUMP, _, _, 26
26: …
```

Παρατηρούμε ακριβώς την ίδια διάρθρωση με τη δομή match αν εξαιρέσουμε ότι η κλήση της getdeps έχει αντικατασταθεί από την εντολή ανάθεσης (πχ. 8) καθώς και το ότι υπάρχει μόνο μία συνθήκη καθώς ελέγχεται μόνο το πρώτο byte του είδους (που δείχνει τον τύπο του κώδικα).

3.6.13. Καταστροφή

Η καταστροφή γίνεται καλώντας στο τέλος κάθε συνάρτησης τη συνάρτηση βιβλιοθήκης καταστροφής σε όλες τις τοπικές μεταβλητές και παραμέτρους.

3.7. Παραγωγή Τελικού Κώδικα

3.7.1. Γενικά

Εδώ θα αναλύσουμε την παραγωγή τελικού κώδικα καθώς και άλλα θέματα σε σχέση με την υλοποίηση της γλώσσας στο χαμηλό επίπεδο.

Και πάλι θα ασχοληθούμε κυρίως με τα στοιχεία του γεννήτορα τελικού κώδικα που έχουν να κάνουν με το δυναμικό κώδικα και η αναφορά στα άλλα θα είναι επιγραμματική.

Η παραγωγή του τελικου κώδικα για τις κλασσικές εντολές ενδιάμεσου είναι πολύ απλή και όχι ιδιαίτερα αποδοτική, αυτό έγινε γιατί ήταν μέσα στους σκοπούς της εργασίας η παραγωγή αποδοτικού κώδικα αλλά η αποδοτική υλοποίηση των δυναμικών λειτουργιών

Την παραγωγή τελικού κώδικα αναλαμβάνουν αποκλειστικά οι κλάσεις CFinalCode και CPattern. Η τελευταία κληρονομεί μάλιστα τη CFinalCode. Η πρώτη αναλαμβάνει τη μεταγλώττιση στατικού κώδικα και η $2^{η}$ τη μεταγλώττιση κώδικα προτύπων. Η λειτουργία της CPattern θα αναλυθεί διεξοδικά στα επόμενα.

3.7.2. Μηχανή – Στόχος

Ο γεννήτορας τελικού κώδικα εξάγει συμβολικές εντολές (assembly instructions) για επεξεργαστή 8086 με Λ/Σ MS-DOS και ουσιαστικά για μοντέλο μνήμης ενός τμήματος (segment) των 64KB.

Οι λόγοι που χρησιμοποιήθηκε αυτό το απαρχαιωμένο μοντέλο ήταν οι εξής:

- Η γλώσσα είναι ούτως ή άλλως περιορισμένων δυνατοτήτων που το μόνο που χρειαζόταν είναι ένα μηχάνημα που μπορούσε να σηκώσει τις εφαρμογές παραδειγμάτων

- Επειδή ο μεταγλωττιστής είναι αυστηρά χωρισμένως, με μια αλλαγή στην υλοποίηση του γεννήτορα τελικού κώδικα μπορούμε να παράγουμε κώδικα για οποιοδήποτε μοντέλο ακολουθώντας τις οδηγίες της ενότητας αυτής παραλλαγμένες καθώς και της ενότητας 3.4.
- Ο αρχικός μεταγλωττιστής που μετατράπηκε για να υποστηρίζει δυναμικό κώδικα ήταν για αυτή την πλατφόρμα

Στα επόμενα θα θεωρήσουμε γνωστές τις δυνατότητες και της συμβολικές εντολές της πλατφόρμας. Εάν ο αναγνώστης δεν είναι εξοικειωμένος με αυτές τον συνιστούμε να ανατρέξει στη βιβλιογραφία.

3.7.3. Δυναμικές Μεταβλητές

Όπως έχουμε αναφέρει οι δυναμικές μεταβλητές αποτελούν στοιχεία πίνακα με συγκεκριμένη χωρητικότητα. Υπάρχουν τόσοι πίνακες όσοι και βασικοί στατικοί τύποι δεδομένων (ένας για χαρακτήρες, ένας για λογικούς και ένας για δείκτες). Οι πίνακες αυτοί καταλαμβάνονται στην αρχή της μνήμης με τις ακόλουθες εντολές που τυπώνονται πριν τον κώδικα.

```
__DVARS_1    db     300 DUP(?)
__DVARS_2    db     300 DUP(?)
__DVARS_4    db     300 DUP(?)
```

Εδώ έχουμε δώσει μέγεθος 300 για τους πίνακες. Οι αριθμοί στα ονόματα αντιστοιχούν σε αναγνωριστικά βασικών τύπων (1=ακέραιοι,2=χαρακτήρες, 4=λογικοί).

3.7.4. Οργάνωση Μνήμης - Εγγραφήματα Δραστηριοποίησης – Μοντέλο Κλήσης

Η μνήμη μας οργανώνεται σε τρία τμήματα: τον κώδικα και τα σταθερά δεδομένα, τη στοίβα(stack) και το σωρό(heap). Όλες οι τοπικές μεταβλητές αποθηκεύονται στη στοίβα. Τα ΑΔΚ αποθηκεύονται στο σωρό. Επισημαίνουμε ότι οι μεταβλητές τύπων codeof περιέχουν τη διεύθυνση ενός ΑΔΚ.

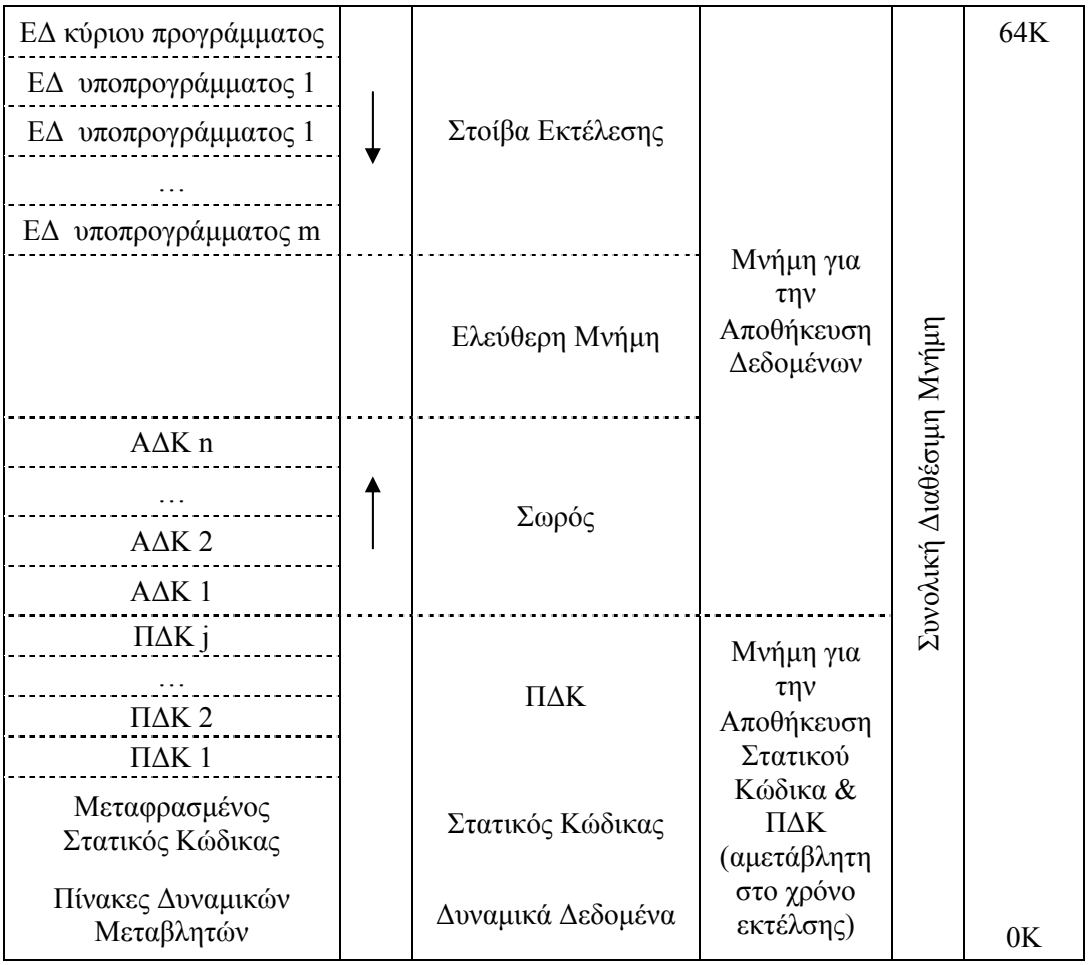
Κάθε ενότητα του προγράμματος (συνάρτηση ή ΑΔΚ) όταν εκτελείται δημιουργεί ένα εγγράφημα δραστηριοποίησης(ΕΔ, activation record, AR) στη στοίβα το οποίο καταστρέφει όταν τελειώσει η εκτέλεσή της. Τα ΕΔ συναρτήσεων και ΑΔΚ είναι ίδια. Περισσότερα για τα ΕΔ μπορούν να αναζητηθούν στη βιβλιογραφία. Η ακριβής μορφή ενός ΕΔ στο μεταγλωττιστή μας είναι η ακόλουθη(προφανώς για τις συναρτήσεις βιβλιοθήκης που δέχονται τις παραμέτρους ανάποδα η σειρά τους στο ΕΔ θα είναι ανεστραμμένη):

	Παράμετρος 1
…	Παράμετρος 2
	…
bp+8	Παράμετρος n
bp+6	Διεύθυνση Αποτελέσματος
bp+4	Προηγούμενο Display
bp+2	Διεύθυνση Επιστροφής
bp	Προηγούμενο bp
bp-2	Τοπική Μεταβλητή 1
	Τοπική Μεταβλητή 2
	…
	Τοπική Μεταβλητή n
…	Προσωρινή Μεταβλητή 1
	Προσωρινή Μεταβλητή 2
	…
	Προσωρινή Μεταβλητή n

Σχήμα 3-1

Σημειώνουμε ότι για την διαχείριση των μη τοπικών μεταβλητών χρησιμοποιήσαμε τον μηχανισμό του πίνακα δεικτών. Περισσότερες πληροφορίες για το μηχανισμό αυτόν μπορούν να αναζητηθούν στη βιβλιογραφία.

Σχηματικά η μνήμη μας μπορεί να παρασταθεί ως εξής:



Σχήμα 3-2

3.7.5. Διαφορές Συναρτήσεων – ΑΔΚ

Όπως έχουμε αναφέρει ο κώδικας των ΑΔΚ γράφεται όπως ένας κώδικας συνάρτησης. Όμως έχει, είτε εξ' ορισμού είτε για λόγους απόδοσης τις εξής διαφορές:

- Δεν έχει παραμέτρους
- Έχει μόνο προσωρινές μεταβλητές
- Δεν αναφέρεται σε μεταβλητές ανώτερων επιπέδων μόνο στους σταθερούς πίνακες των δυναμικών μεταβλητών
- Δεν επιστρέφει το αποτέλεσμα στη διεύθυνση επιστροφής αλλά στον καταχωρητή ax. Αυτό γίνεται για λόγους απόδοσης εκμεταλλευόμενοι το γεγονός ότι όλα τα αποτελέσματα θα χωρούν στον ax αφού δεν έχουμε τύπο στη γλώσσα μας μεγαλύτερο των 2 bytes.

Οι παραπάνω ιδιαιτερότητες θα μας επέτρεπαν να χρησιμοποιήσουμε ένα διαφορετικό ΕΔ για τα ΑΔΚ και να γλιτώσουμε σε μνήμη. Όμως έτσι η συγγραφή

του γεννήτορα τελικού κώδικα θα ήταν πιο χρονοβόρα καθώς θα έπρεπε να αλλάζει τη συμπεριφορά του αναλόγως με το αν μεταγλωττίζει στατικό κώδικα ή δυναμικό (κώδικα προτύπου).

3.7.6. Βοηθητικές Συναρτήσεις Γεννήτορα Τελικού Κώδικα

Για την παραγωγή τελικού κώδικα χρησιμοποιούμε τις παρακάτω βοηθητικές συναρτήσεις που μας παρέχει η κλάση CFinalCode:

- getAR(a): Παράγει τον τελικό κώδικα για τη αντιγραφή της διεύθυνση βάσης του ΕΔ που περιέχει το δεδομένο a στον si.
- loadAddr(R,a): Παράγει τον τελικό κώδικα για τη αντιγραφή της διεύθυνσης του δεδομένου a στον καταχωρητή R
- load(R,a): Παράγει τον τελικό κώδικα για τη αντιγραφή του δεδομένου a στον καταχωρητή R.
- store(R,a): Παράγει τον τελικό κώδικα για τη αντιγραφή του καταχωρητή R στο a.
- name(x): Δίνει την ετικέτα μιας μονάδας προγράμματος x. Η ετικέτα αποτελείται από κάτω παύλα, το όνομα συνάρτησης και το βάθος της.
- endof(x): Δίνει την ετικέτα του τέλους της μονάδας προγράμματος x. Οποιαδήποτε επιστροφή από τη μονάδα πρέπει να κάνει άλμα εκεί για να εκτελεστεί σωστά η καταστροφή του ΕΔ και η επιστροφή στην καλούσα μονάδα.
- label(x): Δίνει την ετικέτα τετράδας ή καθολικού δεδομένου η οποία έχει μορφή: @x όπου x ο αριθμός τετράδας.

3.7.7. Παράσταση Εντολών, CFinalCode, SFinalCmd. SFinalOp

Η μεταγλώττιση του κώδικα προγράμματος γίνεται από την κλάση CFinalCode χρησιμοποιώντας τις βοηθητικές μεθόδους που περιγράφηκαν παραπάνω. Ο τελικός κώδικας παριστάνεται σα μια διπλά συνδεδεμένη λίστα εντολών τελικού κώδικα. Οι εντολές πάλι αναπαριστούνται με τη βοήθεια των δομών SFinalCmd, SFinalOp. Η δήλωση τους παρατίθεται παρακάτω:

```
struct SFinalOp
{
  EFinalOpType type;       //Τύπος ορίσματος
  int size;                //Μέγεθος Ορίσματος
  int off;                 //Offset
  int seg;                 //Τμήμα (segment)
```

```
        union
        {
        char * flabel;          //Ετικέτα
        int reg;                //Καταχωρητής
        char * cnt_str;         //Σταθερή Συμβολοσειρά
        char cnt_c;             //Σταθερός Χαρακτήρας
        short cnt_i;            //Σταθερός Ακέραιος
        };
        ~SFinalOp();
      };
      struct SFinalCmd
      {
        char * label;
        ECmd cmd;
        SFinalOp * ops[2];
        SFinalCmd();
        ~SFinalCmd();
      };
```

Η δομή SFinalCmd αναπαριστά μια εντολή και αποτελείται από την ετικέτα (label), την εντολή (cmd) και τα ορίσματα της εντολής.

Για την αναπαράσταση καταχωρητών και εντολών έχουν κατασκευαστεί κατάλληλες αριθμήσης(enumerations).

Η δομή SFinalOp αναπαριστά ένα όρισμα εντολής τελικού κώδικα. Λόγω ότι τα ορίσματα μπορεί να είναι πολλών ειδών έχει εισαχθεί το union για να μπορούν να περιγραφούν όλα. Δεν έχουν βέβαια όλα τα πεδία νόημα για όλους τους τύπους ορισμάτων. Οι τύποι ορισμάτων δηλώνονται με την αρίθμηση EFinalOpType:

```
enum EFinalOpType
{
      FOP_VOID = 0,           //Κενό
      I_CNT,                  //Ακέραια Σταθερά
      CHAR_CNT,               //Σταθερός Χαρακτήρας
      STR_CNT,                //Συμβολοσειρά
      FLABEL,                 //Ετικέτα
      REGISTER,               //Καταχωρητής
      MEM_REF                 //Έμμεση Αναφορά στη μνήμη
};
```

3.7.8. Κατασκευή ΠΔΚ

Η κατασκευή ενός ΠΔΚ ακολουθεί σε ότι αφορά την παραγωγή τελικού κώδικα τις

ίδιες διαδικασίες με μια συνάρτηση. Παρ' όλα αυτά επειδή το ΠΔΚ πρέπει να φέρει μεταδεδομένα για να μπορεί να "αναγνωστεί" από τη makecode και κατασκευάζεται στο χρόνο εκτέλεσης είναι αναγκαίες ορισμένες τροποποιήσεις και τεχνάσματα τα οποία θα εξετάσουμε σε αυτή την ενότητα.

Αρχικά θα πρέπει να αναλύσουμε λίγο τη λειτουργία των σχισμών σε χαμηλό επίπεδο. Οι σχισμές είναι τμήματα τις γλώσσας μηχανής μιας εντολής που πρέπει να αντικατασταθούν στο χρόνο εκτέλεσης. Κατά τη συγγραφή του τελικού κώδικα μπαίνει η ίδια εντολή με ένα κενό όρισμα ως σχισμή. Πρέπει όμως να δοθεί προσοχή ώστε το κενό όρισμα να έχει μέγεθος ίσο με το δεδομένο που θα εισαχθεί ώστε να μην διαγραφεί η επόμενη εντολή. Για να λύσουμε αυτά τα προβλήματα αυτά κρατάμε 2 πληροφορίες για κάθε σχισμή:

- τη θέση της εντολής στην οποία βρίσκεται
- τη σχετική απόσταση σε bytes της θέσης της σχισμής από την εντολή

Ακόμη φροντίζουμε οι εντολές που τοποθετούνται να είναι τέτοιες ώστε να έχουν αρκετό χώρο για τα δεδομένα. Τελικά μπορέσαμε και χρησιμοποιήσαμε μία μόνο εντολή, την

```
mov     ax, offset __p_(id)
```

η οποία αφήνει αρκετό χώρο (2 bytes) για κάθε τύπο δεδομένων που θα θέλαμε να τοποθετήσουμε (κανείς τύπος στη γλώσσα μας δεν ξεπερνά τα 2 bytes).

Αντιγράφουμε εδώ τον πίνακα με τη δομή ενός ΠΔΚ για να δούμε πως υλοποιεί τη δομή ο κώδικας που παρατίθεται στη συνέχεια:

Πεδίο	**Μέγεθος(σε bytes)**
Μήκος Κώδικα (CODE_LENGTH)	ΜΔ
Αρ. Σχισμών (SLOT_COUNT)	ΜΛ
Κώδικας	CODE_LENGTH
Πίνακας Σχισμών	SLOT_COUNT *ΜΔ

Πίνακας 3-9

Σημειώνουμε ότι ο πίνακας σχισμών πρέπει να περιέχει τη σχετική απόσταση σε bytes κάθε σχισμής από την αρχή του κώδικα.

Ως κώδικας assembly ένα pattern ακολουθεί τους εξής κανόνες κατασκευής:

i. Ξεκινά με την ετικέτα __p_(id) όπου id ένα αναγνωριστικό (δηλ. ένας ακέραιος μοναδικός για κάθε ΠΔΚ) και το μήκος κώδικα. πχ.

```
__p_6           dw              offset __p_6_ce - offset __p_6_cs
```

ii. Ακολουθεί ο αρ. σχισμών (χωρίς ετικέτα). πχ:

```
dw              3
```

iii. Παρατίθεται ο κώδικας. Η αρχή του κώδικα έχει την ετικέτα

__p_(id)_cs.

iv. Κάθε εντολή που περιέχει σχισμή έχει ετικέτα __p_(id)_c_(m) όπου m όπου m ο αύξων αριθμός της σχισμής αρχίζοντας από το 0.

v. Αμέσως μετά το τέλος του κώδικα τοποθετείται η ετικέτα __p_6_ce που σημειώνει και την αρχή του πίνακα σχισμών.

vi. Ακολουθούν οι εγγραφές του πίνακα σχισμών κάθε εγγραφή είναι μια μετα-εντολή dw. πχ.:

```
dw              offset __p_6_c_0 - offset __p_6_cs + 1
```

Βλέπουμε ότι για να υπολογιστεί η απόσταση χρησιμοποιούνται οι προηγούμενες ετικέτες.

Ξεκινώντας θα δούμε τον τελικό κώδικα ενός ΠΔΚ. Το ΠΔΚ αυτό αντιστοιχεί σε μια δομή IF-ELSE και παίρνει όπως βλέπουμε 3 παραμέτρους. Παρατηρούμε ότι ακολουθούνται οι προηγούμενοι κανόνες. Σημειώνουμε ότι το 1 που βλέπουμε στις καταχωρήσεις του πίνακα σχισμών είναι η απόσταση από την αρχή της εντολής.

```
__p_6           dw              offset __p_6_ce - offset __p_6_cs
                dw              3
__p_6_cs:
@132:           push            bp
                mov             bp, sp
                mov             word ptr display[2], bp
                sub             sp, 1
@133:           sub             sp, 2
                push            word ptr display[2]
__p_6_c_0:      mov             ax,  offset __p_6
                call            ax
                mov             byte ptr [bp + (-1)], al
                pop             word ptr display[2]
@134:           mov             al, byte ptr [bp + (-1)]
                or              al, al
                jnz             @136
@135:           jmp             @138
@136:           sub             sp, 2
                push            word ptr display[2]
__p_6_c_1:      mov             ax,  offset __p_6
                call            ax
                pop             word ptr display[2]
@137:           jmp             @139
@138:           sub             sp, 2
                push            word ptr display[2]
__p_6_c_2:      mov             ax,  offset __p_6
                call            ax
                pop             word ptr display[2]
@139:
__p_6_rq:       mov             sp, bp
                pop             bp
                ret
```

```
__p_6_ce:
                dw              offset __p_6_c_0 - offset __p_6_cs + 1
                dw              offset __p_6_c_1 - offset __p_6_cs + 1
                dw              offset __p_6_c_2 - offset __p_6_cs + 1
```

3.7.9. Η κλάση CPattern - Σχισμές

Η κλάση CPattern αναλαμβάνει την παραγωγή τελικού κώδικα για τα ΠΔΚ καθώς και όλη τη διαχείριση. Κληρονομεί τις CQuads, CFinalCode και προσθέτει πεδία και μεθόδους τα οποίες και περιγράφουμε.

Η κλάση CPattern χρησιμοποιή το εγγράφημα PATTERN_SLOT για την αναπαράσταση της σχισμής

```
struct PATTERN_SLOT
{
  SFinalCmd * cmd;
  WORD offset;
};
```

Όπως βλέπουμε το 1^{ο} πεδίο είναι δείκτης σε μια εντολή τελικού κώδικα, η οποία είναι η εντολή στην οποία ανήκει η σχισμή ενώ το δεύτερο είναι η απόσταση σε bytes από την αρχή της εντολής.

Δεδομένα

- `WORD id: Το μοναδικό αναγνωριστικό του ΠΔΚ`
- `WORD cmd_labels: Μετρητής ετικετών εντολών`
- `WORD slot_count,cs: Αριθμός σχισμών`
- `PATTERN_SLOT * slots: Πίνακας Σχισμών`
- `static int LAST_ID: Τελευταίο Αναγνωριστικό ΠΔΚ που χρησιμοποιήθηκε`

Μεθοδοί

- `WORD GetId()`

  Επιστρέφει το αναγνωριστικό του προτύπου

- `void SetParams(WORD id=0,WORD sc=0);`

  Αρχικοποιεί το ΑΔΚ θέτοντας το id(για μη αυτόματο) και τον αριθμό σχισμών

- `void SetSlotPos(SFinalCmd * cmd,WORD offset,int slot=-1);`

  Θέτει τις παραμέτρους μιας σχισμής(slot η θέση της στον πίνακα).

- `char * GetLabel(ELabelKind kind,DWORD param=0);`

  Επιστρέφει μια ετικέτα του προτύπου από αυτές που περιγράφηκαν στην

προηγούμεη παράγραφο ανάλογα με τις παραμέτρους kind(είδος ετικέτας (αρχής, τέλους, αρχής κώδικα, σχισμής κ.τ.λ.), και param (παράμετρος για κάποιους τύπους (π.χ. αρ. σχισμής).

- `void ProduceQuadCode(SQuad * q);`

  Παράγει τον τελικό κώδικα του ΠΔΚ.

- `void PrintCode(FILE * f,DWORD size_dvarray=0);`

  Εκτυπώνει τον τελικό κώδικα σε αρχείο στη μορφή που παρουσιάστηκε παραπάνω.

- `CPattern(WORD sc=0);`

  Κατακευαστής

- `~CPattern(void);`

  Καταστροφέας

3.7.10. Παραγωγή Τελικού Κώδικα για Δυναμικές Τετράδες

Παρακάτω αναλύουμε τον τελικό κώδικα που παράγεται από τετράδες που ασχολούνται με το δυναμικό κώδικα. Επίσης θα αναλύσουμε τις βοηθητικές τετράδες. Σημειώνουμε ότι θα χρησιμοποιούμε τις βοηθητικές συναρτήσεις της ενότητας 3.7.6. Κάποιες τετράδες ενδέχεται να περιέχουν σχισμές οπότε ο τελικός κώδικας αλλάζει. Όπου psize εννοούμε το μέγεθος των τοπικών μεταβλητών της ενότητας (προσωρινών ή όχι). Η συνάρτηση size(T) επιστρέφει το μέγεθος του τύπου. Ακόμη στα patterns παραλείπονται οι ετικέττες καθώς έχουν αναλυθεί σε επόμενο κεφάλαιο.

Τετράδα: `GETFP,x,_,z`

```
n = name(x)
mov      ax,  offset x
store (ax,z)
```

Τετράδα: `EXEC,x,y,_`

```
load(ax,x)
call     ax
store (ax,y)
```
(Αν ορίζεται το y)

Τετράδα: `EXEC,_,y,_` (Με σχισμή)

```
mov      ax,offset __p_(id)
call     ax
store (ax,y)
```
(Αν ορίζεται το y)

Τετράδα: `PATTERN,_,_,x`

```
__p_(id)_ce:     push     bp
                 mov      bp, sp
```

```
                    mov     word ptr display[2], bp
                    sub     sp, psize
```

Τετράδα: `ENDP,_,_,_`

```
__p_(id)_rq:        mov     sp, bp
                    pop     bp
                    ret
```

Τετράδα: `DRET,x,_,_`

```
                    load    (ax,x)
                    jmp     __p_(id)_rq
```

Τετράδα: `DRET,_,_,_` (Με σχισμή)

```
                    mov     ax, offset __p_(id)
                    jmp     __p_(id)_rq
```

Τετράδα: `DVAR,x,_,z` (Με σχισμή - Δεν χρησιμοποιείται αλλιώς)

```
                    mov     ax,  offset __p_0
                    mov     cx, size(x)
                    mul     cx
                    lea     di, __DVARS_(x id)
                    add     di, ax
                    store(di,z)
```

Τετράδα: `GETP,x,_,z`

```
                    mov     ax,  offset __p_(x id)
                    store (ax,z)
```

3.7.11. Βιβλιοθήκη Χρόνου Εκτέλεσης

Εκτός από τις συναρτήσεις βιβλιοθήκης που χρησιμοποιούνται εσωτερικά έχουμε κάποιες άλλες συναρτήσεις τις οποίες και έχουμε υλοποιήσει σε assembly. Οι χρήστες μπορούν να αναφέρονται στις συναρτήσεις αυτές μέσω των αρχείων επικεφαλίδων που περιλαμβάνουν στον κώδικά τους:

Παρατίθενται οι δηλώσεις των συναρτήσεων αυτών:

```
int    abs  (int n);

void   writeInteger (int n);
void   writeBoolean (bool b);
void   writeChar    (char c);
```

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

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

void exit  (int code);
```

3.8. Τρόπος Κλήσης του Μεταγλωττιστή

Ο μεταγλωττιστής καλείται από γραμμή εντολών και παίρνει τις ακόλουθες παραμέτρους:

- Εκτύπωση του ενδιάμεσου κώδικα στην οθόνη με την παράμετρο –f
- Εκτύπωση του τελικού κώδικα στην οθόνη με την παράμετρο –i
- Μέγεθος των πινάκων δυναμικών μεταβλητών(σε bytes) με την παράμετρο –s. πχ.

  ```
  smec ex1 -s1000
  ```

- Αρχείο Εισόδου (Κώδικα) Το αρχείο εισόδου πρέπει υποχρεωτικά να έχει επέκταση .smc και δίνεται χωρίς, πχ:

  ```
  smec ex1 -Id:\includes
  ```

- Δυνατότητα προσθήκης ενός include path η οποία γίνεται με τη σημαία –I<path> με το <Path> να είναι το path χωρίς κενούς χαρακτήρες. π.χ. :

  ```
  smec ex1 -Id:\includes
  ```

Αν δεν δοθούν παράμετροι ο μεταγλωττιστής παράγει:

- Τον τελικό κώδικα σε αρχείο input.asm
- Τον ενδιάμεσο κώδικα σε αρχείο input.imm
- Τον ενδιάμεσο κώδικα των ΠΔΚ σε αρχείο input_p.imm

όπου input το όνομα (χωρίς επέκταση) του αρχείου εισόδου.

4. Παραδείγματα

4.1. Γενικά

Στην ενότητα αυτή θα παρουσιαστούν προγράμματα γραμμένα σε SMEC τα οποία δείχνουν τη λειτουργία της γλώσσας καθώς και του μεταγλωττιστή.

Για κάθε παράδειγμα θα δίνεται μια περιγραφή του προβλήματος, ο κώδικας που το επιλύει σε SMEC, ο ενδιάμεσος κώδικας και ο τελικός κώδικας καθώς και παραδείγματα εκτέλεσης.

Θα παρατηρήσουμε ότι ο τελικός κώδικας των παραδειγμάτων είναι αρκετά μεγάλος. Αυτό γίνεται γιατί η παραγωγή τελικού κώδικα, όπως αναφέρθηκε, γίνεται με ένα σχέδιο που μπορεί να βελτιωθεί σε μεγάλο βαθμό καθώς και γιατί δεν γίνεται καμία επιμέρους βελτίωση στον τελικό κώδικα (αν και ο μεταγλωττιστής υποστηρίζει κάποιες) για να μπορεί να γίνει αντιληπτός ο τρόπος παραγωγής του.

4.2. Εκτύπωση Αριθμών Ν Ψηφίων

4.2.1. Περιγραφή

Στο παράδειγμα αυτό ζητείται η εκτύπωση όλων των δεκαδικών αριθμών Ν ψηφίων όπου το Ν δίνεται από το χρήστη. Αν θεωρήσουμε μεγάλα Ν (>11) αυτό σε μια κλασσική γλώσσα προγραμματισμού παρόμοια με τη SMEC θα απαιτούσε ειδικές δομές και ένα πολύπλοκο πρόγραμμα. Λόγω της ικανότητας δυναμικής δημιουργίας κώδικα λύνουμε το ίδιο πρόβλημα πολύ πιο εύκολα.

4.2.2. Κώδικας

```
#include "stdio.h"

void print_all_nums(int digits)
{
   codeof void res;
   int i;
   res = #writeChar(#'\n');
   for (i=digits-1;i>=0;i--)
         res = res #;  #writeInteger(I#i);
   for (i=0;i<digits;i++)
         res = #for (I#i #= #0;I#i #< #10; I#i #++) res;
   exec res;
}

void main()
{
   int n;
   n = readInteger();
   print_all_nums(n);
}
```

Εδώ λύνουμε το πρόβλημα δημιουργώντας δυναμικά μια σειρά από φωλιασμένα for που αναλαμβάνουν την εκτύπωση. Το μέγεθος του Ν εξαρτάται μόνο από το μέγεθος του πίνακα δυναμικών μεταβλητών.

4.2.3. Ενδιάμεσος Κώδικας Προγράμματος

```
1: UNIT, print_all_nums, _, _
5: GETP, pattern 0, _, $1
6: PAR, '\n', V, _
7: PAR, 0, V, _
8: PAR, 1027, V, _
9: PAR, $1, V, _
10: PAR, $2, RES, _
11: CALL, 7, _, _makecode
12: GETFP, writeChar, _, $3
19: GETP, pattern 1, _, $5
```

```
  20: PAR, $2, V, _
  21: PAR, $3, V, _
  22: PAR, 5, V, _
  23: PAR, $5, V, _
  24: PAR, $6, RES, _
  25: CALL, 8, _, _makecode
  26: :=, $6, _, res
  27: PAR, res, V, _
  28: CALL, 2, _, _addref
  29: -, digits, 1, $7
  30: :=, $7, _, i
  31: >=, i, 0, 34
  32: :=, false, _, $8
  33: JUMP, _, _, 35
  34: :=, true, _, $8
  35: IFB, $8, _, 44
  36: JUMP, _, _, 81
  37: :=, i, _, $9
  38: -, i, 1, i
  39: JUMP, _, _, 31
  44: GETP, pattern 2, _, $11
  45: PAR, i, V, _
  46: PAR, 1, V, _
  47: PAR, 260, V, _
  48: PAR, $11, V, _
  49: PAR, $12, RES, _
  50: CALL, 8, _, _makecode
  51: GETFP, writeInteger, _,
$13
  58: GETP, pattern 3, _, $15
  59: PAR, $12, V, _
  60: PAR, $13, V, _
  61: PAR, 5, V, _
  62: PAR, $15, V, _
  63: PAR, $16, RES, _
  64: CALL, 8, _, _makecode
  69: GETP, pattern 4, _, $17
  70: PAR, $16, V, _
  71: PAR, res, V, _
  72: PAR, 3222, V, _
  73: PAR, 7, V, _
  74: PAR, $17, V, _
  75: PAR, $18, RES, _
  76: CALL, 10, _, _makecode
  77: :=, $18, _, res
  78: PAR, res, V, _
  79: CALL, 2, _, _addref
  80: JUMP, _, _, 37
  81: :=, 0, _, i
  82: <, i, digits, 85
  83: :=, false, _, $19
  84: JUMP, _, _, 86
  85: :=, true, _, $19
  86: IFB, $19, _, 95
  87: JUMP, _, _, 211
  88: :=, i, _, $20
  89: +, i, 1, i
  90: JUMP, _, _, 82
  95: GETP, pattern 5, _, $22
  96: PAR, i, V, _
  97: PAR, 1, V, _
  98: PAR, 260, V, _
  99: PAR, $22, V, _
 100: PAR, $23, RES, _
 101: CALL, 8, _, _makecode
 105: GETP, pattern 6, _, $24
 106: PAR, 0, V, _
 107: PAR, 0, V, _
 108: PAR, 259, V, _
 109: PAR, $24, V, _
 110: PAR, $25, RES, _
 111: CALL, 8, _, _makecode
 118: GETP, pattern 7, _, $28
 119: PAR, $25, V, _
 120: PAR, $23, V, _
 121: PAR, 1658, V, _
 122: PAR, 262, V, _
 123: PAR, $28, V, _
 124: PAR, $29, RES, _
 125: CALL, 10, _, _makecode
 130: GETP, pattern 8, _, $31
 131: PAR, i, V, _
 132: PAR, 1, V, _
 133: PAR, 260, V, _
 134: PAR, $31, V, _
 135: PAR, $32, RES, _
 136: CALL, 8, _, _makecode
 140: GETP, pattern 9, _, $33
 141: PAR, 10, V, _
 142: PAR, 0, V, _
 143: PAR, 259, V, _
 144: PAR, $33, V, _
 145: PAR, $34, RES, _
 146: CALL, 8, _, _makecode
 156: GETP, pattern 10, _, $38
 157: PAR, $34, V, _
 158: PAR, $32, V, _
 159: PAR, 1150, V, _
 160: PAR, 518, V, _
 161: PAR, $38, V, _
 162: PAR, $39, RES, _
 163: CALL, 10, _, _makecode
 168: GETP, pattern 11, _, $41
 169: PAR, i, V, _
 170: PAR, 1, V, _
 171: PAR, 260, V, _
 172: PAR, $41, V, _
 173: PAR, $42, RES, _
 174: CALL, 8, _, _makecode
 181: GETP, pattern 12, _, $45
 182: PAR, $42, V, _
 183: PAR, 3469, V, _
```

```
184: PAR, 263, V, _
185: PAR, $45, V, _
186: PAR, $46, RES, _
187: CALL, 8, _, _makecode
197: GETP, pattern 13, _, $48
198: PAR, $46, V, _
199: PAR, res, V, _
200: PAR, $39, V, _
201: PAR, $29, V, _
202: PAR, 0, V, _
203: PAR, 2, V, _
204: PAR, $48, V, _
205: PAR, $49, RES, _
206: CALL, 14, _, _makecode
207: :=, $49, _, res
208: PAR, res, V, _
209: CALL, 2, _, _addref
210: JUMP, _, _, 88
211: EXEC, res, _, _
212: PAR, $49, V, _
213: CALL, 2, _, _destroy
214: PAR, $46, V, _
215: CALL, 2, _, _destroy
216: PAR, $42, V, _
217: CALL, 2, _, _destroy
218: PAR, $39, V, _
219: CALL, 2, _, _destroy
220: PAR, $34, V, _
221: CALL, 2, _, _destroy
222: PAR, $32, V, _
223: CALL, 2, _, _destroy
224: PAR, $29, V, _
225: CALL, 2, _, _destroy
226: PAR, $25, V, _
227: CALL, 2, _, _destroy
228: PAR, $23, V, _
229: CALL, 2, _, _destroy
230: PAR, $18, V, _
231: CALL, 2, _, _destroy
232: PAR, $16, V, _
233: CALL, 2, _, _destroy
234: PAR, $12, V, _
235: CALL, 2, _, _destroy
236: PAR, $6, V, _
237: CALL, 2, _, _destroy
238: PAR, $2, V, _
239: CALL, 2, _, _destroy
240: PAR, res, V, _
241: CALL, 2, _, _destroy
242: ENDU, print_all_nums, _, _
243: UNIT, main, _, _
244: PAR, $50, RES, _
245: CALL, 0, _, readInteger
246: :=, $50, _, n
247: PAR, n, V, _
248: CALL, 2, _, print_all_nums
249: ENDU, main, _, _
```

4.2.4. Ενδιάμεσος Κώδικας ΠΔΚ

```
  2: PATTERN, pattern 0, _, _
  3: DRET, _, _, _
  4: ENDP, pattern 0, _, _
 13: PATTERN, pattern 1, _, _
 14: EXEC, _, $4, _
 15: PAR, $4, V, _
 16: CALL, 1, _, writeChar
 17: DRET, _, _, _
 18: ENDP, pattern 1, _, _
 40: PATTERN, pattern 2, _, _
 41: DVAR, integer, _, $10
 42: DRET, $10, _, _
 43: ENDP, pattern 2, _, _
 52: PATTERN, pattern 3, _, _
 53: EXEC, _, $14, _
 54: PAR, [$14], V, _
 55: CALL, 2, _, writeInteger
 56: DRET, _, _, _
 57: ENDP, pattern 3, _, _
 65: PATTERN, pattern 4, _, _
 66: EXEC, _, _, _
 67: EXEC, _, _, _
 68: ENDP, pattern 4, _, _
 91: PATTERN, pattern 5, _, _
 92: DVAR, integer, _, $21
 93: DRET, $21, _, _
 94: ENDP, pattern 5, _, _
102: PATTERN, pattern 6, _, _
103: DRET, _, _, _
104: ENDP, pattern 6, _, _
112: PATTERN, pattern 7, _, _
113: EXEC, _, $26, _
114: EXEC, _, $27, _
115: :=, $27, _, [$26]
116: DRET, [$26], _, _
117: ENDP, pattern 7, _, _
126: PATTERN, pattern 8, _, _
127: DVAR, integer, _, $30
128: DRET, $30, _, _
129: ENDP, pattern 8, _, _
137: PATTERN, pattern 9, _, _
138: DRET, _, _, _
139: ENDP, pattern 9, _, _
147: PATTERN, pattern 10, _, _
```

```
148: EXEC, _, $35, _
149: EXEC, _, $36, _
150: <, [$35], $36, 153
151: :=, false, _, $37
152: JUMP, _, _, 154
153: :=, true, _, $37
154: DRET, $37, _, _
155: ENDP, pattern 10, _, _
164: PATTERN, pattern 11, _, _
165: DVAR, integer, _, $40
166: DRET, $40, _, _
167: ENDP, pattern 11, _, _
175: PATTERN, pattern 12, _, _
176: EXEC, _, $43, _
177: :=, [$43], _, $44
178: +, [$43], 1, [$43]
179: DRET, $44, _, _
180: ENDP, pattern 12, _, _
188: PATTERN, pattern 13, _, _
189: EXEC, _, _, _
190: EXEC, _, $47, _
191: IFB, $47, _, 193
192: JUMP, _, _, 196
193: EXEC, _, _, _
194: EXEC, _, _, _
195: JUMP, _, _, 190
196: ENDP, pattern 13, _, _
```

4.2.5. Τελικός Κώδικας Προγράμματος

```
xseg  segment public 'code'
      assume cs:xseg,ds:xseg,ss:xseg

                    .8087

                    org       100h
__DVARS_1           db        300 DUP(?)
__DVARS_2           db        300 DUP(?)
__DVARS_4           db        300 DUP(?)
main                proc near
                    mov       ax,cs
                    mov       ds,ax
                    mov       ss,ax
                    mov       es,ax
                    call near ptr    _main
                    mov       ax, 4C00h
                    int       21h
main                endp
_print_all_nums     proc near
                    push      bp
                    mov       bp, sp
                    mov       word ptr display, bp
                    sub       sp, 72
@5:                 mov       ax,  offset __p_0
                    mov       word ptr [bp + (-6)], ax
@6:                 mov       al, 10
                    sub       sp, 1
                    mov       si, sp
                    mov       byte ptr [si + (0)], al
@7:                 mov       ax, 0
                    push      ax
@8:                 mov       ax, 1027
                    push      ax
@9:                 mov       ax, word ptr [bp + (-6)]
                    push      ax
@10:                lea       si, word ptr [bp + (-8)]
                    push      si
@11:                push      word ptr display
                    lea       ax, __makecode
                    call      ax
```

```
                pop     word ptr display
                add     sp, 7
@12:            mov     ax,  offset _writeChar
                mov     word ptr [bp + (-10)], ax
@19:            mov     ax,  offset __p_1
                mov     word ptr [bp + (-12)], ax
@20:            mov     ax, word ptr [bp + (-8)]
                push    ax
@21:            mov     ax, word ptr [bp + (-10)]
                push    ax
@22:            mov     ax, 5
                push    ax
@23:            mov     ax, word ptr [bp + (-12)]
                push    ax
@24:            lea     si, word ptr [bp + (-14)]
                push    si
@25:            push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@26:            mov     ax, word ptr [bp + (-14)]
                mov     word ptr [bp + (-2)], ax
@27:            mov     ax, word ptr [bp + (-2)]
                push    ax
@28:            sub     sp, 2
                push    word ptr display
                lea     ax, __addref
                call    ax
                pop     word ptr display
                add     sp, 2
@29:            mov     ax, word ptr [bp + (8)]
                mov     cx, 1
                sub     ax, cx
                mov     word ptr [bp + (-16)], ax
@30:            mov     ax, word ptr [bp + (-16)]
                mov     word ptr [bp + (-4)], ax
@31:            mov     ax, word ptr [bp + (-4)]
                mov     cx, 0
                cmp     ax, cx
                jge     @34
@32:            mov     ax, 0
                mov     byte ptr [bp + (-17)], al
@33:            jmp     @35
@34:            mov     ax, 1
                mov     byte ptr [bp + (-17)], al
@35:            mov     al, byte ptr [bp + (-17)]
                or      al, al
                jnz     @44
@36:            jmp     @81
@37:            mov     ax, word ptr [bp + (-4)]
                mov     word ptr [bp + (-19)], ax
@38:            mov     ax, word ptr [bp + (-4)]
                mov     cx, 1
                sub     ax, cx
                mov     word ptr [bp + (-4)], ax
@39:            jmp     @31
@44:            mov     ax,  offset __p_2
                mov     word ptr [bp + (-21)], ax
@45:            mov     ax, word ptr [bp + (-4)]
                push    ax
```

```
@46:            mov     ax, 1
                push    ax
@47:            mov     ax, 260
                push    ax
@48:            mov     ax, word ptr [bp + (-21)]
                push    ax
@49:            lea     si, word ptr [bp + (-23)]
                push    si
@50:            push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@51:            mov     ax,  offset _writeInteger
                mov     word ptr [bp + (-25)], ax
@58:            mov     ax,  offset __p_3
                mov     word ptr [bp + (-27)], ax
@59:            mov     ax, word ptr [bp + (-23)]
                push    ax
@60:            mov     ax, word ptr [bp + (-25)]
                push    ax
@61:            mov     ax, 5
                push    ax
@62:            mov     ax, word ptr [bp + (-27)]
                push    ax
@63:            lea     si, word ptr [bp + (-29)]
                push    si
@64:            push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@69:            mov     ax,  offset __p_4
                mov     word ptr [bp + (-31)], ax
@70:            mov     ax, word ptr [bp + (-29)]
                push    ax
@71:            mov     ax, word ptr [bp + (-2)]
                push    ax
@72:            mov     ax, 3222
                push    ax
@73:            mov     ax, 7
                push    ax
@74:            mov     ax, word ptr [bp + (-31)]
                push    ax
@75:            lea     si, word ptr [bp + (-33)]
                push    si
@76:            push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 10
@77:            mov     ax, word ptr [bp + (-33)]
                mov     word ptr [bp + (-2)], ax
@78:            mov     ax, word ptr [bp + (-2)]
                push    ax
@79:            sub     sp, 2
                push    word ptr display
                lea     ax, __addref
                call    ax
                pop     word ptr display
                add     sp, 2
```

```
@80:            jmp     @37
@81:            mov     ax, 0
                mov     word ptr [bp + (-4)], ax
@82:            mov     ax, word ptr [bp + (-4)]
                mov     cx, word ptr [bp + (8)]
                cmp     ax, cx
                jl      @85
@83:            mov     ax, 0
                mov     byte ptr [bp + (-34)], al
@84:            jmp     @86
@85:            mov     ax, 1
                mov     byte ptr [bp + (-34)], al
@86:            mov     al, byte ptr [bp + (-34)]
                or      al, al
                jnz     @95
@87:            jmp     @211
@88:            mov     ax, word ptr [bp + (-4)]
                mov     word ptr [bp + (-36)], ax
@89:            mov     ax, word ptr [bp + (-4)]
                mov     cx, 1
                add     ax, cx
                mov     word ptr [bp + (-4)], ax
@90:            jmp     @82
@95:            mov     ax,  offset __p_5
                mov     word ptr [bp + (-38)], ax
@96:            mov     ax, word ptr [bp + (-4)]
                push    ax
@97:            mov     ax, 1
                push    ax
@98:            mov     ax, 260
                push    ax
@99:            mov     ax, word ptr [bp + (-38)]
                push    ax
@100:           lea     si, word ptr [bp + (-40)]
                push    si
@101:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@105:           mov     ax,  offset __p_6
                mov     word ptr [bp + (-42)], ax
@106:           mov     ax, 0
                push    ax
@107:           mov     ax, 0
                push    ax
@108:           mov     ax, 259
                push    ax
@109:           mov     ax, word ptr [bp + (-42)]
                push    ax
@110:           lea     si, word ptr [bp + (-44)]
                push    si
@111:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@118:           mov     ax,  offset __p_7
                mov     word ptr [bp + (-46)], ax
@119:           mov     ax, word ptr [bp + (-44)]
                push    ax
```

```
@120:           mov     ax, word ptr [bp + (-40)]
                push    ax
@121:           mov     ax, 1658
                push    ax
@122:           mov     ax, 262
                push    ax
@123:           mov     ax, word ptr [bp + (-46)]
                push    ax
@124:           lea     si, word ptr [bp + (-48)]
                push    si
@125:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 10
@130:           mov     ax,  offset __p_8
                mov     word ptr [bp + (-50)], ax
@131:           mov     ax, word ptr [bp + (-4)]
                push    ax
@132:           mov     ax, 1
                push    ax
@133:           mov     ax, 260
                push    ax
@134:           mov     ax, word ptr [bp + (-50)]
                push    ax
@135:           lea     si, word ptr [bp + (-52)]
                push    si
@136:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@140:           mov     ax,  offset __p_9
                mov     word ptr [bp + (-54)], ax
@141:           mov     ax, 10
                push    ax
@142:           mov     ax, 0
                push    ax
@143:           mov     ax, 259
                push    ax
@144:           mov     ax, word ptr [bp + (-54)]
                push    ax
@145:           lea     si, word ptr [bp + (-56)]
                push    si
@146:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@156:           mov     ax,  offset __p_10
                mov     word ptr [bp + (-58)], ax
@157:           mov     ax, word ptr [bp + (-56)]
                push    ax
@158:           mov     ax, word ptr [bp + (-52)]
                push    ax
@159:           mov     ax, 1150
                push    ax
@160:           mov     ax, 518
                push    ax
@161:           mov     ax, word ptr [bp + (-58)]
                push    ax
```

```
@162:           lea     si, word ptr [bp + (-60)]
                push    si
@163:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 10
@168:           mov     ax,  offset __p_11
                mov     word ptr [bp + (-62)], ax
@169:           mov     ax, word ptr [bp + (-4)]
                push    ax
@170:           mov     ax, 1
                push    ax
@171:           mov     ax, 260
                push    ax
@172:           mov     ax, word ptr [bp + (-62)]
                push    ax
@173:           lea     si, word ptr [bp + (-64)]
                push    si
@174:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@181:           mov     ax,  offset __p_12
                mov     word ptr [bp + (-66)], ax
@182:           mov     ax, word ptr [bp + (-64)]
                push    ax
@183:           mov     ax, 3469
                push    ax
@184:           mov     ax, 263
                push    ax
@185:           mov     ax, word ptr [bp + (-66)]
                push    ax
@186:           lea     si, word ptr [bp + (-68)]
                push    si
@187:           push    word ptr display
                lea     ax, __makecode
                call    ax
                pop     word ptr display
                add     sp, 8
@197:           mov     ax,  offset __p_13
                mov     word ptr [bp + (-70)], ax
@198:           mov     ax, word ptr [bp + (-68)]
                push    ax
@199:           mov     ax, word ptr [bp + (-2)]
                push    ax
@200:           mov     ax, word ptr [bp + (-60)]
                push    ax
@201:           mov     ax, word ptr [bp + (-48)]
                push    ax
@202:           mov     ax, 0
                push    ax
@203:           mov     ax, 2
                push    ax
@204:           mov     ax, word ptr [bp + (-70)]
                push    ax
@205:           lea     si, word ptr [bp + (-72)]
                push    si
@206:           push    word ptr display
                lea     ax, __makecode
```

```
                    call      ax
                    pop       word ptr display
                    add       sp, 14
@207:               mov       ax, word ptr [bp + (-72)]
                    mov       word ptr [bp + (-2)], ax
@208:               mov       ax, word ptr [bp + (-2)]
                    push      ax
@209:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __addref
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@210:               jmp       @88
@211:               sub       sp, 2
                    push      word ptr display[2]
                    mov       ax, word ptr [bp + (-2)]
                    call      ax
                    pop       word ptr display[2]
@212:               mov       ax, word ptr [bp + (-72)]
                    push      ax
@213:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __destroy
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@214:               mov       ax, word ptr [bp + (-68)]
                    push      ax
@215:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __destroy
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@216:               mov       ax, word ptr [bp + (-64)]
                    push      ax
@217:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __destroy
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@218:               mov       ax, word ptr [bp + (-60)]
                    push      ax
@219:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __destroy
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@220:               mov       ax, word ptr [bp + (-56)]
                    push      ax
@221:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __destroy
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@222:               mov       ax, word ptr [bp + (-52)]
                    push      ax
```

```
@223:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
                add     sp, 2
@224:           mov     ax, word ptr [bp + (-48)]
                push    ax
@225:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
                add     sp, 2
@226:           mov     ax, word ptr [bp + (-44)]
                push    ax
@227:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
                add     sp, 2
@228:           mov     ax, word ptr [bp + (-40)]
                push    ax
@229:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
                add     sp, 2
@230:           mov     ax, word ptr [bp + (-33)]
                push    ax
@231:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
                add     sp, 2
@232:           mov     ax, word ptr [bp + (-29)]
                push    ax
@233:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
                add     sp, 2
@234:           mov     ax, word ptr [bp + (-23)]
                push    ax
@235:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
                add     sp, 2
@236:           mov     ax, word ptr [bp + (-14)]
                push    ax
@237:           sub     sp, 2
                push    word ptr display
                lea     ax, __destroy
                call    ax
                pop     word ptr display
```

```
                    add       sp, 2
@238:               mov       ax, word ptr [bp + (-8)]
                    push      ax
@239:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __destroy
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@240:               mov       ax, word ptr [bp + (-2)]
                    push      ax
@241:               sub       sp, 2
                    push      word ptr display
                    lea       ax, __destroy
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@242:
@print_all_nums_1:  mov       sp, bp
                    pop       bp
                    ret
_print_all_nums     endp
_main               proc near
                    push      bp
                    mov       bp, sp
                    mov       word ptr display, bp
                    sub       sp, 4
@244:               lea       si, word ptr [bp + (-4)]
                    push      si
@245:               push      word ptr display
                    lea       ax, _readInteger
                    call      ax
                    pop       word ptr display
                    add       sp, 0
@246:               mov       ax, word ptr [bp + (-4)]
                    mov       word ptr [bp + (-2)], ax
@247:               mov       ax, word ptr [bp + (-2)]
                    push      ax
@248:               sub       sp, 2
                    push      word ptr display
                    lea       ax, _print_all_nums
                    call      ax
                    pop       word ptr display
                    add       sp, 2
@249:
@main_1:            mov       sp, bp
                    pop       bp
                    ret
_main               endp
__p_0               dw        offset __p_0_ce - offset __p_0_cs
                    dw        1
__p_0_cs:
@2:                 push      bp
                    mov       bp, sp
                    mov       word ptr display[2], bp
                    sub       sp, 0
__p_0_c_0:          mov       ax,  offset __p_0
                    jmp       __p_0_rq
@4:
__p_0_rq:           mov       sp, bp
                    pop       bp
```

```
                    ret
__p_0_ce:
                    dw      offset __p_0_c_0 - offset __p_0_cs + 1
__p_1               dw      offset __p_1_ce - offset __p_1_cs
                    dw      1
__p_1_cs:
@13:                push    bp
                    mov     bp, sp
                    mov     word ptr display[2], bp
                    sub     sp, 1
@14:                sub     sp, 2
                    push    word ptr display[2]
__p_1_c_0:          mov     ax,  offset __p_1
                    call    ax
                    mov     byte ptr [bp + (-1)], al
                    pop     word ptr display[2]
@15:                mov     al, byte ptr [bp + (-1)]
                    sub     sp, 1
                    mov     si, sp
                    mov     byte ptr [si + (0)], al
@16:                sub     sp, 2
                    push    word ptr display
                    lea     ax, _writeChar
                    call    ax
                    pop     word ptr display
                    add     sp, 1
__p_1_c_1:          mov     ax,  offset __p_1
                    jmp     __p_1_rq
@18:
__p_1_rq:           mov     sp, bp
                    pop     bp
                    ret
__p_1_ce:
                    dw      offset __p_1_c_0 - offset __p_1_cs + 1
__p_2               dw      offset __p_2_ce - offset __p_2_cs
                    dw      1
__p_2_cs:
@40:                push    bp
                    mov     bp, sp
                    mov     word ptr display[2], bp
                    sub     sp, 2
__p_2_c_0:          mov     ax,  offset __p_2
                    mov     cx, 2
                    mul     cx
                    lea     di, __DVARS_1
                    add     di, ax
                    mov     word ptr [bp + (-2)], di
@42:                mov     ax, word ptr [bp + (-2)]
                    jmp     __p_2_rq
@43:
__p_2_rq:           mov     sp, bp
                    pop     bp
                    ret
__p_2_ce:
                    dw      offset __p_2_c_0 - offset __p_2_cs + 1
__p_3               dw      offset __p_3_ce - offset __p_3_cs
                    dw      1
__p_3_cs:
@52:                push    bp
                    mov     bp, sp
                    mov     word ptr display[2], bp
```

```
                    sub     sp, 2
@53:                sub     sp, 2
                    push    word ptr display[2]
__p_3_c_0:          mov     ax,  offset __p_3
                    call    ax
                    mov     word ptr [bp + (-2)], ax
                    pop     word ptr display[2]
@54:                mov     di, word ptr [bp + (-2)]
                    mov     ax, word ptr [di + (0)]
                    push    ax
@55:                sub     sp, 2
                    push    word ptr display
                    lea     ax, _writeInteger
                    call    ax
                    pop     word ptr display
                    add     sp, 2
__p_3_c_1:          mov     ax,  offset __p_3
                    jmp     __p_3_rq
@57:
__p_3_rq:           mov     sp, bp
                    pop     bp
                    ret
__p_3_ce:
                    dw      offset __p_3_c_0 - offset __p_3_cs + 1
__p_4               dw      offset __p_4_ce - offset __p_4_cs
                    dw      2
__p_4_cs:
@65:                push    bp
                    mov     bp, sp
                    mov     word ptr display[2], bp
                    sub     sp, 0
@66:                sub     sp, 2
                    push    word ptr display[2]
__p_4_c_0:          mov     ax,  offset __p_4
                    call    ax
                    pop     word ptr display[2]
@67:                sub     sp, 2
                    push    word ptr display[2]
__p_4_c_1:          mov     ax,  offset __p_4
                    call    ax
                    pop     word ptr display[2]
@68:
__p_4_rq:           mov     sp, bp
                    pop     bp
                    ret
__p_4_ce:
                    dw      offset __p_4_c_0 - offset __p_4_cs + 1
                    dw      offset __p_4_c_1 - offset __p_4_cs + 1
__p_5               dw      offset __p_5_ce - offset __p_5_cs
                    dw      1
__p_5_cs:
@91:                push    bp
                    mov     bp, sp
                    mov     word ptr display[2], bp
                    sub     sp, 2
__p_5_c_0:          mov     ax,  offset __p_5
                    mov     cx, 2
                    mul     cx
                    lea     di, __DVARS_1
                    add     di, ax
                    mov     word ptr [bp + (-2)], di
```

```
@93:            mov     ax, word ptr [bp + (-2)]
                jmp     __p_5_rq
@94:
__p_5_rq:       mov     sp, bp
                pop     bp
                ret
__p_5_ce:
                dw      offset __p_5_c_0 - offset __p_5_cs + 1
__p_6           dw      offset __p_6_ce - offset __p_6_cs
                dw      1
__p_6_cs:
@102:           push    bp
                mov     bp, sp
                mov     word ptr display[2], bp
                sub     sp, 0
__p_6_c_0:      mov     ax,  offset __p_6
                jmp     __p_6_rq
@104:
__p_6_rq:       mov     sp, bp
                pop     bp
                ret
__p_6_ce:
                dw      offset __p_6_c_0 - offset __p_6_cs + 1
__p_7           dw      offset __p_7_ce - offset __p_7_cs
                dw      2
__p_7_cs:
@112:           push    bp
                mov     bp, sp
                mov     word ptr display[2], bp
                sub     sp, 4
@113:           sub     sp, 2
                push    word ptr display[2]
__p_7_c_0:      mov     ax,  offset __p_7
                call    ax
                mov     word ptr [bp + (-2)], ax
                pop     word ptr display[2]
@114:           sub     sp, 2
                push    word ptr display[2]
__p_7_c_1:      mov     ax,  offset __p_7
                call    ax
                mov     word ptr [bp + (-4)], ax
                pop     word ptr display[2]
@115:           mov     ax, word ptr [bp + (-4)]
                mov     di, word ptr [bp + (-2)]
                mov     word ptr [di + (0)], ax
@116:           mov     di, word ptr [bp + (-2)]
                mov     ax, word ptr [di + (0)]
                jmp     __p_7_rq
@117:
__p_7_rq:       mov     sp, bp
                pop     bp
                ret
__p_7_ce:
                dw      offset __p_7_c_0 - offset __p_7_cs + 1
                dw      offset __p_7_c_1 - offset __p_7_cs + 1
__p_8           dw      offset __p_8_ce - offset __p_8_cs
                dw      1
__p_8_cs:
@126:           push    bp
                mov     bp, sp
                mov     word ptr display[2], bp
```

```
                    sub     sp, 2
__p_8_c_0:          mov     ax,  offset __p_8
                    mov     cx, 2
                    mul     cx
                    lea     di, __DVARS_1
                    add     di, ax
                    mov     word ptr [bp + (-2)], di
@128:               mov     ax, word ptr [bp + (-2)]
                    jmp     __p_8_rq
@129:
__p_8_rq:           mov     sp, bp
                    pop     bp
                    ret
__p_8_ce:
                    dw      offset __p_8_c_0 - offset __p_8_cs + 1
__p_9               dw      offset __p_9_ce - offset __p_9_cs
                    dw      1
__p_9_cs:
@137:               push    bp
                    mov     bp, sp
                    mov     word ptr display[2], bp
                    sub     sp, 0
__p_9_c_0:          mov     ax,  offset __p_9
                    jmp     __p_9_rq
@139:
__p_9_rq:           mov     sp, bp
                    pop     bp
                    ret
__p_9_ce:
                    dw      offset __p_9_c_0 - offset __p_9_cs + 1
__p_10              dw      offset __p_10_ce - offset __p_10_cs
                    dw      2
__p_10_cs:
@147:               push    bp
                    mov     bp, sp
                    mov     word ptr display[2], bp
                    sub     sp, 5
@148:               sub     sp, 2
                    push    word ptr display[2]
__p_10_c_0:         mov     ax,  offset __p_10
                    call    ax
                    mov     word ptr [bp + (-2)], ax
                    pop     word ptr display[2]
@149:               sub     sp, 2
                    push    word ptr display[2]
__p_10_c_1:         mov     ax,  offset __p_10
                    call    ax
                    mov     word ptr [bp + (-4)], ax
                    pop     word ptr display[2]
@150:               mov     di, word ptr [bp + (-2)]
                    mov     ax, word ptr [di + (0)]
                    mov     cx, word ptr [bp + (-4)]
                    cmp     ax, cx
                    jl      @153
@151:               mov     ax, 0
                    mov     byte ptr [bp + (-5)], al
@152:               jmp     @154
@153:               mov     ax, 1
                    mov     byte ptr [bp + (-5)], al
@154:               mov     ax, byte ptr [bp + (-5)]
                    jmp     __p_10_rq
```

```
@155:
__p_10_rq:          mov       sp, bp
                    pop       bp
                    ret
__p_10_ce:
                    dw        offset __p_10_c_0 - offset __p_10_cs + 1
                    dw        offset __p_10_c_1 - offset __p_10_cs + 1
__p_11              dw        offset __p_11_ce - offset __p_11_cs
                    dw        1
__p_11_cs:
@164:               push      bp
                    mov       bp, sp
                    mov       word ptr display[2], bp
                    sub       sp, 2
__p_11_c_0:         mov       ax,  offset __p_11
                    mov       cx, 2
                    mul       cx
                    lea       di, __DVARS_1
                    add       di, ax
                    mov       word ptr [bp + (-2)], di
@166:               mov       ax, word ptr [bp + (-2)]
                    jmp       __p_11_rq
@167:
__p_11_rq:          mov       sp, bp
                    pop       bp
                    ret
__p_11_ce:
                    dw        offset __p_11_c_0 - offset __p_11_cs + 1
__p_12              dw        offset __p_12_ce - offset __p_12_cs
                    dw        1
__p_12_cs:
@175:               push      bp
                    mov       bp, sp
                    mov       word ptr display[2], bp
                    sub       sp, 4
@176:               sub       sp, 2
                    push      word ptr display[2]
__p_12_c_0:         mov       ax,  offset __p_12
                    call      ax
                    mov       word ptr [bp + (-2)], ax
                    pop       word ptr display[2]
@177:               mov       di, word ptr [bp + (-2)]
                    mov       ax, word ptr [di + (0)]
                    mov       word ptr [bp + (-4)], ax
@178:               mov       di, word ptr [bp + (-2)]
                    mov       ax, word ptr [di + (0)]
                    mov       cx, 1
                    add       ax, cx
                    mov       di, word ptr [bp + (-2)]
                    mov       word ptr [di + (0)], ax
@179:               mov       ax, word ptr [bp + (-4)]
                    jmp       __p_12_rq
@180:
__p_12_rq:          mov       sp, bp
                    pop       bp
                    ret
__p_12_ce:
                    dw        offset __p_12_c_0 - offset __p_12_cs + 1
__p_13              dw        offset __p_13_ce - offset __p_13_cs
                    dw        4
__p_13_cs:
```

```
@188:           push    bp
                mov     bp, sp
                mov     word ptr display[2], bp
                sub     sp, 1
@189:           sub     sp, 2
                push    word ptr display[2]
__p_13_c_0:     mov     ax,  offset __p_13
                call    ax
                pop     word ptr display[2]
@190:           sub     sp, 2
                push    word ptr display[2]
__p_13_c_1:     mov     ax,  offset __p_13
                call    ax
                mov     byte ptr [bp + (-1)], al
                pop     word ptr display[2]
@191:           mov     al, byte ptr [bp + (-1)]
                or      al, al
                jnz     @193
@192:           jmp     @196
@193:           sub     sp, 2
                push    word ptr display[2]
__p_13_c_2:     mov     ax,  offset __p_13
                call    ax
                pop     word ptr display[2]
@194:           sub     sp, 2
                push    word ptr display[2]
__p_13_c_3:     mov     ax,  offset __p_13
                call    ax
                pop     word ptr display[2]
@195:           jmp     @190
@196:
__p_13_rq:      mov     sp, bp
                pop     bp
                ret
__p_13_ce:
                dw      offset __p_13_c_0 - offset __p_13_cs + 1
                dw      offset __p_13_c_1 - offset __p_13_cs + 1
                dw      offset __p_13_c_2 - offset __p_13_cs + 1
                dw      offset __p_13_c_3 - offset __p_13_cs + 1
display         dw      10 dup(0)
                extrn   _readInteger : proc
                extrn   _writeChar : proc
                extrn   _writeInteger : proc
                extrn   __makecode : proc
                extrn   __destroy : proc
                extrn   __addref : proc
xseg            ends
                end     main
```

4.2.6. Παράδειγμα Εκτέλεσης

Τα αποτελέσματα εκτέλεσης δεν αναφέρονται καθώς είναι τετριμένα.

4.3. Αναδρομική Ανάλυση Κώδικα

4.3.1. Περιγραφή

Εδώ θα παραθέσουμε μια συνάρτηση η οποία αναλύει αναδρομικά ένα κώδικα και τυπώνει μια περιγραφή του. Η συνάρτηση αναλύει κώδικα και αναγνωρίζει ένα υποσύνολο των δομών τις γλώσσας. Θα μπορούσαν να τοποθετηθούν όλες αλλά αυτό δεν έγινε για λόγους συντομίας καθώς δεν θα πρόσθετε κάτι καινούριο.

4.3.2. Κώδικας

```
#include "stdio.h"

void check_code(codeof void c)
{
	int t;
	typematch (c)
	{
		case void : c_void : {t=0;}
		case int : c_int : {t=1;}
		case bool : c_bool : {t=2;}
		case char : c_char : {t=3;}
	};
	writeInteger(t);
	writeChar('\n');

	match (c)
	{
		case int const:
		{
			writeChar('C'); writeChar('I'); writeChar(':');
			writeInteger(const);
		}
		case I # var_num:
		{
			writeChar('V'); writeChar('I'); writeChar(':');
			writeInteger(var_num);
		}
		case op1 #== op2:
		{
			writeChar('=');
			writeChar('=');
			writeChar('\n');
			writeChar('\t');
			check_code(op1);
			writeChar('\n');
			writeChar('\t');
			check_code(op2);
		}
		case op1 #= op2:
		{
```

```
                writeChar('\n');
                writeChar('\t');
                check_code(op1);
                writeChar('=');
                writeChar('\n');
                writeChar('\t');
                check_code(op2);
            }
            case #if (cond) stmt #else else_stmt:
            {
                writeChar('I');
                writeChar('F');
                writeChar('\n');
                writeChar('\t');
                check_code(cond);
                writeChar('\n');
                writeChar('\t');
                check_code(stmt);
                writeChar('\n');
                writeChar('\t');
                check_code(else_stmt);
            }
            case #for (init;cond;iter) body:
            {
                writeChar('F');writeChar('O');writeChar('R');
                writeChar(':');
                writeChar('\n');
                writeChar('\t');
                check_code(init);
                writeChar('\n');
                writeChar('\t');
                check_code(cond);
                writeChar('\n');
                writeChar('\t');
                check_code(iter);
                writeChar('\n');
                writeChar('\t');
                check_code(body);
            }
            case otherwise:
            {
                writeChar('N');writeChar('/');writeChar('A');
            }
        }
        writeChar('\n');
}


void main()
{
  int i;
  codeof int k;
  codeof int j;
  codeof int l;
  codeof void m;
```

```
  m = #if (#3 #== #5) (I#0 #= #0)  #else (I#0 #= #1) ;

  check_code(m);

}
```

4.3.3. Ενδιάμεσος Κώδικας Προγράμματος

```
 1: UNIT, check_code, _, _
 2: PAR, 0, V, _
 3: PAR, {$1}, V, _
 4: PAR, c, V, _
 5: CALL, 6, _, _gettype
 6: =, 0, $1, 8
 7: JUMP, _, _, 11
 8: :=, c, _, c_void
 9: :=, 0, _, t
10: JUMP, _, _, 26
11: =, 256, $1, 13
12: JUMP, _, _, 16
13: :=, c, _, c_int
14: :=, 1, _, t
15: JUMP, _, _, 26
16: =, 512, $1, 18
17: JUMP, _, _, 21
18: :=, c, _, c_bool
19: :=, 2, _, t
20: JUMP, _, _, 26
21: =, 1024, $1, 23
22: JUMP, _, _, 26
23: :=, c, _, c_char
24: :=, 3, _, t
25: JUMP, _, _, 26
26: PAR, t, V, _
27: CALL, 2, _, writeInteger
28: PAR, '\n', V, _
29: CALL, 1, _, writeChar
30: PAR, {$3}, V, _
31: PAR, {$2}, V, _
32: PAR, c, V, _
33: CALL, 6, _, _gettype
34: =, 3, $2, 36
35: JUMP, _, _, 50
36: =, 0, $3, 38
37: JUMP, _, _, 50
38: PAR, {const}, V, _
39: PAR, c, V, _
40: CALL, 4, _, _getdeps
41: PAR, 'C', V, _
42: CALL, 1, _, writeChar
43: PAR, 'I', V, _
44: CALL, 1, _, writeChar
45: PAR, ':', V, _
46: CALL, 1, _, writeChar
47: PAR, const, V, _
48: CALL, 2, _, writeInteger
49: JUMP, _, _, 196
50: =, 4, $2, 52
51: JUMP, _, _, 66
52: =, 1, $3, 54
53: JUMP, _, _, 66
54: PAR, {var_num}, V, _
55: PAR, c, V, _
56: CALL, 4, _, _getdeps
57: PAR, 'V', V, _
58: CALL, 1, _, writeChar
59: PAR, 'I', V, _
60: CALL, 1, _, writeChar
61: PAR, ':', V, _
62: CALL, 1, _, writeChar
63: PAR, var_num, V, _
64: CALL, 2, _, writeInteger
65: JUMP, _, _, 196
66: =, 6, $2, 68
67: JUMP, _, _, 91
68: =, 1147, $3, 70
69: JUMP, _, _, 91
70: PAR, {op2}, V, _
71: PAR, {op1}, V, _
72: PAR, c, V, _
73: CALL, 6, _, _getdeps
74: PAR, '=', V, _
75: CALL, 1, _, writeChar
76: PAR, '=', V, _
77: CALL, 1, _, writeChar
78: PAR, '\n', V, _
79: CALL, 1, _, writeChar
80: PAR, '\t', V, _
81: CALL, 1, _, writeChar
82: PAR, op1, V, _
83: CALL, 2, _, check_code
84: PAR, '\n', V, _
85: CALL, 1, _, writeChar
86: PAR, '\t', V, _
87: CALL, 1, _, writeChar
88: PAR, op2, V, _
89: CALL, 2, _, check_code
90: JUMP, _, _, 196
```

```
 91: =, 6, $2, 93
 92: JUMP, _, _, 114
 93: =, 1658, $3, 95
 94: JUMP, _, _, 114
 95: PAR, {op2}, V, _
 96: PAR, {op1}, V, _
 97: PAR, c, V, _
 98: CALL, 6, _, _getdeps
 99: PAR, '\n', V, _
100: CALL, 1, _, writeChar
101: PAR, '\t', V, _
102: CALL, 1, _, writeChar
103: PAR, op1, V, _
104: CALL, 2, _, check_code
105: PAR, '=', V, _
106: CALL, 1, _, writeChar
107: PAR, '\n', V, _
108: CALL, 1, _, writeChar
109: PAR, '\t', V, _
110: CALL, 1, _, writeChar
111: PAR, op2, V, _
112: CALL, 2, _, check_code
113: JUMP, _, _, 196
114: =, 8, $2, 116
115: JUMP, _, _, 146
116: =, 0, $3, 118
117: JUMP, _, _, 146
118: PAR, {else_stmt}, V, _
119: PAR, {stmt}, V, _
120: PAR, {cond}, V, _
121: PAR, c, V, _
122: CALL, 8, _, _getdeps
123: PAR, 'I', V, _
124: CALL, 1, _, writeChar
125: PAR, 'F', V, _
126: CALL, 1, _, writeChar
127: PAR, '\n', V, _
128: CALL, 1, _, writeChar
129: PAR, '\t', V, _
130: CALL, 1, _, writeChar
131: PAR, cond, V, _
132: CALL, 2, _, check_code
133: PAR, '\n', V, _
134: CALL, 1, _, writeChar
135: PAR, '\t', V, _
136: CALL, 1, _, writeChar
137: PAR, stmt, V, _
138: CALL, 2, _, check_code
139: PAR, '\n', V, _
140: CALL, 1, _, writeChar
141: PAR, '\t', V, _
142: CALL, 1, _, writeChar
143: PAR, else_stmt, V, _
144: CALL, 2, _, check_code
145: JUMP, _, _, 196
146: =, 2, $2, 148
147: JUMP, _, _, 189
148: =, 0, $3, 150
149: JUMP, _, _, 189
150: PAR, {body}, V, _
151: PAR, {iter}, V, _
152: PAR, {cond}, V, _
153: PAR, {init}, V, _
154: PAR, c, V, _
155: CALL, 10, _, _getdeps
156: PAR, 'F', V, _
157: CALL, 1, _, writeChar
158: PAR, 'O', V, _
159: CALL, 1, _, writeChar
160: PAR, 'R', V, _
161: CALL, 1, _, writeChar
162: PAR, ':', V, _
163: CALL, 1, _, writeChar
164: PAR, '\n', V, _
165: CALL, 1, _, writeChar
166: PAR, '\t', V, _
167: CALL, 1, _, writeChar
168: PAR, init, V, _
169: CALL, 2, _, check_code
170: PAR, '\n', V, _
171: CALL, 1, _, writeChar
172: PAR, '\t', V, _
173: CALL, 1, _, writeChar
174: PAR, cond, V, _
175: CALL, 2, _, check_code
176: PAR, '\n', V, _
177: CALL, 1, _, writeChar
178: PAR, '\t', V, _
179: CALL, 1, _, writeChar
180: PAR, iter, V, _
181: CALL, 2, _, check_code
182: PAR, '\n', V, _
183: CALL, 1, _, writeChar
184: PAR, '\t', V, _
185: CALL, 1, _, writeChar
186: PAR, body, V, _
187: CALL, 2, _, check_code
188: JUMP, _, _, 196
189: PAR, 'N', V, _
190: CALL, 1, _, writeChar
191: PAR, '/', V, _
192: CALL, 1, _, writeChar
193: PAR, 'A', V, _
194: CALL, 1, _, writeChar
195: JUMP, _, _, 196
196: PAR, '\n', V, _
197: CALL, 1, _, writeChar
198: ENDU, check_code, _, _
199: UNIT, main, _, _
203: GETP, pattern 0, _, $4
204: PAR, 3, V, _
205: PAR, 0, V, _
```

```
206: PAR, 259, V, _
207: PAR, $4, V, _
208: PAR, $5, RES, _
209: CALL, 8, _, _makecode
213: GETP, pattern 1, _, $6
214: PAR, 5, V, _
215: PAR, 0, V, _
216: PAR, 259, V, _
217: PAR, $6, V, _
218: PAR, $7, RES, _
219: CALL, 8, _, _makecode
229: GETP, pattern 2, _, $11
230: PAR, $7, V, _
231: PAR, $5, V, _
232: PAR, 1147, V, _
233: PAR, 518, V, _
234: PAR, $11, V, _
235: PAR, $12, RES, _
236: CALL, 10, _, _makecode
241: GETP, pattern 3, _, $14
242: PAR, 0, V, _
243: PAR, 1, V, _
244: PAR, 260, V, _
245: PAR, $14, V, _
246: PAR, $15, RES, _
247: CALL, 8, _, _makecode
251: GETP, pattern 4, _, $16
252: PAR, 0, V, _
253: PAR, 0, V, _
254: PAR, 259, V, _
255: PAR, $16, V, _
256: PAR, $17, RES, _
257: CALL, 8, _, _makecode
264: GETP, pattern 5, _, $20
265: PAR, $17, V, _
266: PAR, $15, V, _
267: PAR, 1658, V, _
268: PAR, 262, V, _
269: PAR, $20, V, _
270: PAR, $21, RES, _
271: CALL, 10, _, _makecode
276: GETP, pattern 6, _, $23
277: PAR, 0, V, _
278: PAR, 1, V, _
279: PAR, 260, V, _
280: PAR, $23, V, _
281: PAR, $24, RES, _
282: CALL, 8, _, _makecode
286: GETP, pattern 7, _, $25
287: PAR, 1, V, _
288: PAR, 0, V, _
289: PAR, 259, V, _
290: PAR, $25, V, _
291: PAR, $26, RES, _
292: CALL, 8, _, _makecode
299: GETP, pattern 8, _, $29
300: PAR, $26, V, _
301: PAR, $24, V, _
302: PAR, 1658, V, _
303: PAR, 262, V, _
304: PAR, $29, V, _
305: PAR, $30, RES, _
306: CALL, 10, _, _makecode
315: GETP, pattern 9, _, $32
316: PAR, $30, V, _
317: PAR, $21, V, _
318: PAR, $12, V, _
319: PAR, 0, V, _
320: PAR, 8, V, _
321: PAR, $32, V, _
322: PAR, $33, RES, _
323: CALL, 12, _, _makecode
324: :=, $33, _, m
325: PAR, m, V, _
326: CALL, 2, _, _addref
327: PAR, m, V, _
328: CALL, 2, _, check_code
329: PAR, $33, V, _
330: CALL, 2, _, _destroy
331: PAR, $30, V, _
332: CALL, 2, _, _destroy
333: PAR, $26, V, _
334: CALL, 2, _, _destroy
335: PAR, $24, V, _
336: CALL, 2, _, _destroy
337: PAR, $21, V, _
338: CALL, 2, _, _destroy
339: PAR, $17, V, _
340: CALL, 2, _, _destroy
341: PAR, $15, V, _
342: CALL, 2, _, _destroy
343: PAR, $12, V, _
344: CALL, 2, _, _destroy
345: PAR, $7, V, _
346: CALL, 2, _, _destroy
347: PAR, $5, V, _
348: CALL, 2, _, _destroy
349: PAR, m, V, _
350: CALL, 2, _, _destroy
351: PAR, l, V, _
352: CALL, 2, _, _destroy
353: PAR, j, V, _
354: CALL, 2, _, _destroy
355: PAR, k, V, _
356: CALL, 2, _, _destroy
357: ENDU, main, _, _
```

4.3.4. Ενδιάμεσος Κώδικας ΠΔΚ

```
200: PATTERN, pattern 0, _, _
 201: DRET, _, _, _
 202: ENDP, pattern 0, _, _
 210: PATTERN, pattern 1, _, _
 211: DRET, _, _, _
 212: ENDP, pattern 1, _, _
 220: PATTERN, pattern 2, _, _
 221: EXEC, _, $8, _
 222: EXEC, _, $9, _
 223: =, $8, $9, 226
 224: :=, false, _, $10
 225: JUMP, _, _, 227
 226: :=, true, _, $10
 227: DRET, $10, _, _
 228: ENDP, pattern 2, _, _
 237: PATTERN, pattern 3, _, _
 238: DVAR, integer, _, $13
 239: DRET, $13, _, _
 240: ENDP, pattern 3, _, _
 248: PATTERN, pattern 4, _, _
 249: DRET, _, _, _
 250: ENDP, pattern 4, _, _
 258: PATTERN, pattern 5, _, _
 259: EXEC, _, $18, _
 260: EXEC, _, $19, _
 261: :=, $19, _, [$18]
 262: DRET, [$18], _, _
 263: ENDP, pattern 5, _, _
 272: PATTERN, pattern 6, _, _
 273: DVAR, integer, _, $22
 274: DRET, $22, _, _
 275: ENDP, pattern 6, _, _
 283: PATTERN, pattern 7, _, _
 284: DRET, _, _, _
 285: ENDP, pattern 7, _, _
 293: PATTERN, pattern 8, _, _
 294: EXEC, _, $27, _
 295: EXEC, _, $28, _
 296: :=, $28, _, [$27]
 297: DRET, [$27], _, _
 298: ENDP, pattern 8, _, _
 307: PATTERN, pattern 9, _, _
 308: EXEC, _, $31, _
 309: IFB, $31, _, 311
 310: JUMP, _, _, 313
 311: EXEC, _, _, _
 312: JUMP, _, _, 314
 313: EXEC, _, _, _
 314: ENDP, pattern 9, _, _
```

4.3.5. Τελικός Κώδικας Προγράμματος

Ο τελικός κώδικας δεν περιλαμβάνεται για λόγους συντομίας

4.3.6. Παράδειγμα Εκτέλεσης

```
0
IF
        2
==
        1
CI:3

        1
CI:5


        1

        1
VI:0
=
        1
CI:0


        1
```

```
        1
VI:0
=
        1
CI:1
```

5. Συμπεράσματα

5.1. Συνεισφορά

Τελειώνοντας θα θέλαμε να πούμε ότι η εργασία αυτή εκπλήρωσε το σκοπό της , ο οποίος ήταν η διερεύνηση της δυνατότητας να χρησιμοποιήσουμε τη φιλοσοφία του αυτομεταβαλλόμενου κώδικα στις σύγχρονες γλώσσες προγραμματισμού με ισχυρά συστήματα τύπων.

Συγκεκριμένα κατασκευάσθηκε μία γλώσσα που μπορεί να κατασκευάζει, να αναλύει και να εκτελεί κώδικα στο χρόνο εκτέλεσης διατηρώντας ένα αποτελεσματικό σύστημα τύπων ανάλογης ασφάλειας με αυτή των κλασσικών γλωσσών προγραμματισμού.

Παράλληλα, υλοποιήθηκε η γλώσσα αυτή με έναν μεταγλωττιστή και με τέτοιο τρόπο ώστε οι παραπάνω διαδικασίες να εκτελούνται με τη μικρότερη δυνατή χρονική επιβάρυνση. Το παραπάνω σημείο θεωρούμε ως ιδιαίτερα σημαντικό αφού μια γλώσσα με μεγαλύτερες δυνατότητες που θα επέβαλλαν όμως μεγάλη χρονική επιβάρυνση θα είχε μειωμένη λειτουργικότητα. Επιπλέον ο μεταγλωττιστής επιδέχεται πολλές

βελτιώσεις που θα μπορούσαν

Από την άλλη μεριά η γλώσσα αυτή δεν παρέχει την ίδια ευελιξία ως προς τη διαχείριση του κώδικα με γλώσσες χαμηλότερου επιπέδου όπως η συμβολική. Αυτό όμως είναι ένα μικρό τίμημα σε σχέση με την ασφάλεια που παρέχει στον προγραμματιστή αυτό το μοντέλο και μάλλον αναπόφευκτο από αυτή τη σκοπιά.

5.2. Μελλοντικές Επεκτάσεις

Η γλώσσα που κατασκευάσαμε είναι απλή και πρακτικά ελάχιστα χρήσιμη. Πάντως οι αρχές που χρησιμοποιήθηκαν για την κατασκευή των δυναμικών επεκτάσεων μπορούν να χρησιμοποιηθούν και σε πιο πολύπλοκες γλώσσες προγραμματισμού και μάλιστα με καλύτερα αποτελέσματα. Η κατασκευή μιας πλήρους γλώσσας με δυναμικές επεκτάσεις θα ήταν ίσως η πιο προφανής συνέχεια της παρούσας εργασίας.

Σε μια περίπτωση όπως αυτή περιγράφεται παραπάνω ο τρόπος υλοποίησης των δυναμικών επεκτάσεων θα μπορούσε να είναι ο ίδιος που παρουσιάστηκε στην ενότητα 3.4. Παραπέρα τα ίδια τα επιπλέον χαρακτηριστικά της νέας γλώσσας θα έδιναν στο δυναμικό κώδικα επιπλέον δυνατότητες. Για παράδειγμα η δυναμική κλήση συνάρτησης θα μπορούσε να αναλυθεί πολύ καλύτερα και να αποβεί πολύ περισσότερο χρήσιμη σε μια γλώσσα η οποία θα επέτρεπε δείκτες σε συναρτήσεις (ή γενικότερα συναρτήσεις ως γλωσσικά στοιχεία 1ης τάξης).

Συνεχίζοντας θα μπορούσαμε να αναφέρουμε ότι η υλοποίηση του δυναμικού κώδικα δέχεται τόσο τις γνωστές μεθόδους βελτιστοποίησης ενδιάμεσου και τελικού κώδικα, τις οποίες δεν υλοποιήσαμε στη γλώσσα που κατασκευάσαμε, αλλά και περαιτέρω μεθόδους.

Ως παράδειγμα στα παραπάνω αναφέρουμε ότι η γλώσσα που κατασκευάστηκε κατασκευάζει για κάθε δυναμική έκφραση που συναντάει ένα πρότυπο. Παρ' όλα αυτά μπορεί να υπάρχουν σε ένα πρόγραμμα εκφράσεις που παράγουν κοινά πρότυπα. Για παράδειγμα όλα τα IF και όλα τα FOR παράγουν ακριβώς το ίδιο ΠΔΚ. Ένας βελτιστοποιητής που θα εντόπιζε τέτοιες εκφράσεις και θα τις συνέδεε με ένα μόνο πρότυπο θα εξοικονομούσε μνήμη στο τελικό σύστημα.

Βιβλιογραφία

[1] . Νικόλαος Παπασπύρου και Εμμανουήλ Σκορδαλάκης, *Μεταγλωττιστές*, Εκδ. Συμμετρία, 2002

[2] . Benjamin C. Pierce, *Types and Programming Languages*, The MIT Press, 2002

[3] . Edsger W. Dijkstra, *EWD 1284*

[4] . Kenneth C. Louden, *Programming Languages,* Course Technology, 2002

[5] . Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms,* Second Edition, MIT Press, 2001.