Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Έλεγχος Ορθότητας Μεταγλωττιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ Γ. ΚΑΡΑΚΟΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Έλεγχος Ορθότητας Μεταγλωττιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ Γ. ΚΑΡΑΚΟΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Νοεμβρίου 2019.

Νικόλαος Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Δημήτριος Φωτάκης Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2019

Γεώργιος Γ. Καράκος

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

Περίληψη

Στη σημερινή εποχή, η ανάγκη για αξιόπιστο και πιστοποιημένα ασφαλή κώδικα γίνεται διαρκώς ευρύτερα αντιληπτή. Προγράμματα που παρουσιάζουν σφάλματα έχουν ως αποτέλεσμα προβλήματα στην λειτουργία μεγάλων συστημάτων και συνεπώς οικονομικές επιπτώσεις στους οργανισμούς που τα χρησιμοποιούν. Οι μεταγλωττιστές γλωσσών προγραμματισμού είναι εργαλεία υπεύθυνα για τη μετατροπή κώδικα γραμμένου σε γλώσσα υψηλού επιπέδου σε εκτελέσιμο πρόγραμμα. Επομένως, το να υπάρχουν λάθη σε έναν μεταγλωττιστή καθιστά κάθε πρόγραμμα που δημιουργήθηκε από αυτόν επικίνδυνο στη χρήση. Είναι σημαντικό λοιπόν να δοκιμάζονται οι μεταγλωττιστές προκειμένου να εντοπίζονται πιθανά λάθη. Σκοπός της παρούσας εργασίας είναι η σχεδίαση ενός συστήματος ελέγχου ορθότητας ενός μεταγλωττιστή έχοντας ένα πρόγραμμα εισόδου καθώς και ο έλεγχος της σωστής λειτουργίας του εκτελέσιμου που παράγεται.

Λέξεις κλειδιά

Μεταγλωττιστές, Προγραμματισμός με αποδείξεις, Ορθότητα, Black-box testing, Πιστοποιημένος κώδικας.

Abstract

The need for reliable and certifiably secure code is even more pressing today than it was in the past. Programs containing errors put in danger the operation of large systems, with substancial financial consequences. A compiler is a tool tasked with translating code written in a high-level programming language to an executable program. Therefore, the existence of errors in a compiler compromises any program created by it. That means it's important to test a compiler in order to discover an error through use. The purpose of this diploma dissertation is the design of a system that checks the correctness of a compiler when given a suitable input program. The system then checks the behavior of the executable if one is produced.

Key words

Compilers, Programming with proofs, Correctness, Black-box testing, Secure programming, Certified code.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Νικόλαο Παπασπύρου, για τη συνεχή καθοδήγηση και εμπιστοσύνη του. Θα ήθελα να ευχαριστήσω τους φίλους και συμφοιτητές που ήταν δίπλα μου καθόλη τη διάρκεια των σπουδών μου. Θα ήθελα τέλος να ευχαριστήσω την οικογένειά μου και κυρίως τους γονείς μου, οι οποίοι με υποστήριξαν και έκαναν δυνατή την απερίσπαστη ενασχόλησή μου τόσο με την εκπόνηση της διπλωματικής μου, όσο και συνολικά με τις σπουδές μου.

Γεώργιος Γ. Καράκος,

Αθήνα, 5η Νοεμβρίου 2019

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-7-19, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Νοέμβριος 2019.

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

Περιεχόμενα

Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Ο μεταγλωττιστής (compiler) [Moge10] είναι ένα πρόγραμμα που μεταφράζει τον κώδικα ενός προγράμματος γραμμένου σε μια συγκεκριμένη γλώσσα προγραμματισμού σε ένα ισοδύναμο γραμμένο σε μια διαφορετική γλώσσα. Οι μεταγλωττιστές χρησιμοποιούνται συνήθως για τη μετατροπή κώδικα από μια γλώσσα υψηλού επιπέδου, όπως για παράδειγμα η C, σε μια χαμηλού επιπέδου η οποία στις περισσότερες περιπτώσεις είναι η γλώσσα μηχανής του τοπικού μηχανήματος με σκοπό να παραχθεί ένα εκτελέσιμο πρόγραμμα. Είναι καίριας σημασίας το να γνωρίζει ένας προγραμματιστής για θέματα που αφορούν μεταγλωττιστές, καθώς ο μεταγλωττιστής αποτελεί ένα εργαλείο που σχετίζεται άμεσα με τον τομέα εργασίας του. Αρκεί να σκεφτούμε ότι τα εργαλεία που χρησιμοποιούνται καθημερινά στην ανάπτυξη λογισμικού είναι γραμμένα σε διαφορετικές γλώσσες προγραμματισμού και είναι ευθύνη του μεταγλωττιστή να τα μετατρέψει σε μια μορφή ώστε να μπορούν να εκτελεστούν από τον προγραμματιστή.

Έχοντας γνώση των παραπάνω κατανοούμε ότι για τον τομέα της επιστήμης λογισμικού οι μεταγλωττιστές είναι καίριας σημασίας, εφόσον είναι υπεύθυνοι για τη δημιουργία αξιόπιστων εκτελέσιμων προγραμμάτων. Αν υπάρξει λάθος στον μεταγλωττιστή τότε το πρόγραμμα που δημιουργείται δεν εκτελεί τις ενέργειες που έχουν οριστεί από τον προγραμματιστή και τη σημασιολογία της γλώσσας. Εφόσον η λειτουργία του παρεκλίνει από αυτή που έχει οριστεί τότε η εκτέλεση του μπορεί να προκαλέσει προβλήματα.

Γνωρίζοντας ότι λάθη αυτού του τύπου είναι δύσκολο να εντοπιστούν μετά τη παραγωγή του προγράμματος, είναι σημαντικό να ελέγχουμε ότι ένας μεταγλωττιστής λειτουργεί σωστά. Αποφασίζοντας να ασχοληθούμε με αυτό το ζήτημα, στοχεύουμε σε αυτή τη διπλωματική εργασία στη δημιουργία ενός συστήματος για αναλυτικό έλεγχο της ορθής λειτουργίας ενός μεταγλωττιστή. Ο έλεγχος γίνεται για κάθε στάδιο του μεταγλωττιστή ξεχωριστά καθώς και για τη συνολική του λειτουργία. Έπειτα ελέγχεται το πρόγραμμα που παράγεται για το αν εκτελεί την επιθυμητή λειτουργία έχοντας δεδομένη είσοδο. Το σύστημα αυτό είναι σχεδιασμένο για χρήση από προγραμματιστές που θέλουν να ελέγξουν τον compiler που έχουν γράψει για λάθη και για το αν καλύπτει επαρκώς τις απαιτήσεις της αρχικής γλώσσας.

1.2 Ιστορική αναδρομή

Η εξέλιξη των μεταγλωττιστών είναι μια προσπάθεια η οποία συνεχίζεται μέχρι και σήμερα. Νέες γλώσσες προγραμματισμού εμφανίζονται με σκοπό να καλύψουν νέες ανάγκες στην ανάπτυξη λογισμικού. Μπορούμε να ξεχωρίσουμε σημαντικές εξελίξεις στο σχεδιασμό μεταγλωττιστών σε συγκεκριμένες χρονολογικές περιόδους [Grun12].

Κατά τη περίοδο 1945 - 1960 οι γλώσσες προγραμματισμού αναπτύχθηκαν αργά και το κύριο πρόβλημα ήταν το πως να παραχθεί κώδικας σε ένα μηχάνημα. Ο προγραμματισμός σε γλώσσα assembly ήταν ακόμα αρκετά χρησιμοποιούμενη πρακτική καθώς υπήρχε η πεποίθηση ότι ο κώδικας που παραγόταν από μεταγλωττιστές δεν ήταν εξίσου καλώς με αυτόν που ο προγραμματιστής έγραφε με το χέρι. Προκειμένου λοιπόν να εδραιωθεί ο προγραμαμτισμός σε γλώσσες υψηλού επιπέδου έπρεπε αυ-

τός ο φόβος να ξεπεραστεί. Έχοντας γνώση αυτών, μπορούμε να κατανοήσουμε ότι η ανάπτυξη του πρώτου μεταγλωττιστή της γλώσσας FORTRAN το 1959 ήταν ένα τολμηρό εγχείρημα και θεωρείται ως ένα από τα σημαντικότερα προγραμματιστικά projects της εποχής εκείνης.

Κατά τη περίοδο 1960 - 1975 αναπτύχθηκαν πολλές νέες γλώσσες προγραμματισμού και οι σχεδιαστές τους άρχισαν να πιστεύουν ότι το να διαθέτουν έναν compiler για μια νέα γλώσσα γρήγορα ήταν σημαντικότερο από το να έχουν έναν που να παράγει πολύ αποδοτικό κώδικα. Αυτό έφερε μια στροφή στην κατασκευή των μεταγλωττιστών, καθώς άρχισε να δίνεται έμφαση στο front-end κομμάτι απ'ότι στο back-end.

Μετά το 1975 ο αριθμός των νέων γλωσσών που σχεδιάζονταν καθώς και ο αριθμός των διαφορετικών τύπων μηχανημάτων που χρησιμοποιούνταν άρχισε να μειώνεται. Αυτό μείωσε την ανάγκη για τη γρήγορη κατασκευή απλών μεταγλωττιστών. Έτσι άρχισε η ζήτηση για μεταγλωττιστές που ήταν αξιόπιστοι και αποδοτικοί στην παραγωγή κώδικα.

1.3 Δομή της εργασίας

Στο δεύτερο κεφάλαιο παρατίθεται θεωρία γύρω από τους μεταγλωττιστές και τις μεθόδους ελέγχου τους. Αναλύουμε τα στάδια λειτουργίας ενός μεταγλωττιστή, τεχνικές ελέγχου λογισμικού και αναφερόμαστε σε ορισμένες τεχνικές που διακρίνονται για τον έλεγχο μεταγλωττιστών. Στο τρίτο κεφάλαιο παρουσιάζεται αναλυτικά ο τρόπος που δουλεύει το σύστημα που έχουμε δημιουργήσει. Στο τελευταίο κομμάτι διατυπώνονται η σύνοψη της δουλειάς που έγινε καθώς και κάποιες σκέψεις για μελλοντική έρευνα. Ακολουθεί στο τέλος η βιβλιογραφία η οποία μελετήθηκε για την υλοποίηση αυτής της εργασίας.

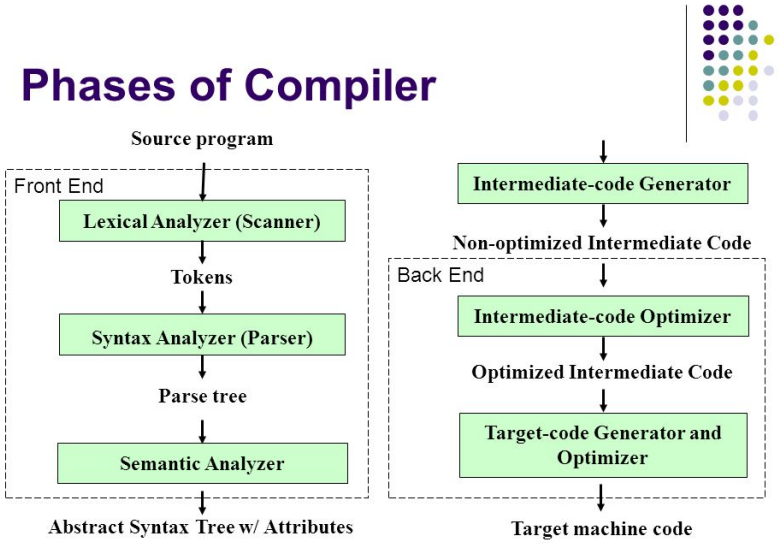
Κεφάλαιο 2

Θεωρία

2.1 Θεωρία για μεταγλωττιστές

Μεταγλωττιστής ονομάζεται ένα πρόγραμμα ηλεκτρονικού υπολογιστή το οποίο διαβάζει κώδικα γραμμένο σε μια γλώσσα προγραμματισμού (πηγαία γλώσσα) και τον μεταφράζει σε ισοδύναμο κώδικα σε μια άλλη γλώσσα προγραμματισμού (γλώσσα στόχο). Όταν μιλάμε για μεταγλωττιστές αναφερόμαστε συνήθως σε λογισμικό που μεταφράζει μια γλώσσα προγραμματισμού υψηλού επιπέδου σε μια χαμηλού επιπέδου όπως για παράδειγμα η γλώσσα μηχανής που είναι κατανοητή από τον ηλεκτρονικό υπολογιστή. Ο λόγος που αυτή είναι η πιο συνηθισμένη περίπτωση χρήσης είναι ότι το hardware του υπολογιστή μπορεί να ”τρέξει” το μεταφρασμένο πρόγραμμα αν αυτό είναι γραμμένο σε κατανοητή προς τον υπολογιστή γλώσσα. Δηλαδή το hardware θα εκτελέσει τις ενέργειες που έχουν οριστεί από τη σημασιολογία του προγράμματος.

Ο μεταγλωττιστής χωρίζεται σε δύο μέρη. Το μέρος που υλοποιεί την ανάλυση του κειμένου της αρχικής γλώσσας ονομάζεται front-end και το μέρος που υλοποιεί την σύνθεση της γλώσσας στόχου είναι το back-end του μεταγλωττιστή. Παρακάτω περιγράφεται ένας συνηθισμένος διαχωρισμός της συγγραφής ενός compiler σε ξεχωριστές φάσεις. Σε πολλούς μεταγλωττιστές μερικές από τις φάσεις αυτές μπορεί να ενώνονται ή άλλες να διαχωρίζονται σε περαιτέρω στάδια ή ακόμα και να προστίθενται καινούργιες φάσεις στη διαδικασία. Το σύστημα που υλοποιήθηκε στα πλαίσια αυτής της διπλωματικής ακολουθεί αυτόν τον διαχωρισμό σε φάσεις.



Σχήμα 2.1: Παράδειγμα παρουσίασης των φάσεων ενός μεταγλωττιστή
Από: `https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design`

2.1.1 Λεκτική ανάλυση

Στη φάση της λεκτικής ανάλυσης (lexical analysis) ο μεταγλωττιστής δέχεται ως είσοδο ένα πρόγραμμα με τη μορφή συμβολοσειράς χαρακτήρων και δίνει ως έξοδο ένα ισοδύναμο πρόγραμμα με τη μορφή λεκτικών μονάδων οι οποίες ονομάζονται αλλιώς tokens. Οι λεκτικές αυτές μονάδες είναι τα τερματικά σύμβολα της γραμματικής που περιγράφει τη σύνταξη της αρχικής γλώσσας προγραμματισμού στη φάση της συντακτικής ανάλυσης. Αυτός είναι ο λόγος που η συντακτική και η λεκτική ανάλυση είναι συνδεδεμένες φάσεις του μεταγλωττιστή. Ο διαχωρισμός τους γίνεται για λόγους ευκολίας υλοποίησης.

Το τμήμα του μεταγλωττιστή που υλοποιεί τη φάση της λεκτικής ανάλυσης ονομάζεται λεκτικός αναλυτής (lexical analyzer ή lexer). Σε περίπτωση σφαλμάτων μπορεί να καλεί το κομμάτι του κώδικα που είναι υπεύθυνο για τη διαχείριση σφαλμάτων και να εμφανίζει κατάλληλο μήνυμα.

2.1.2 Συντακτική ανάλυση

Στη φάση της συντακτικής ανάλυσης (syntactic analysis ή αλλιώς parsing) ελέγχεται αν το πρόγραμμα εισόδου ανήκει στη γλώσσα της οποίας η σύνταξη ορίζεται από μια δεδομένη context free γραμματική. Κατα τη διάρκεια αυτής της φάσης κατασκευάζεται το συντακτικό δέντρο που αντιστοιχεί στο αρχικό πρόγραμμα. Το δέντρο αυτό χρησιμεύει σε μεταγενέστερες φάσεις του μεταγλωττιστή οι οποίες θα ανεφερθούν στη συνέχεια.

Η είσοδος του συντακτικού αναλυτή (parser) είναι το αρχικό πρόγραμμα με τη μορφή λεκτικών μονάδων, όπως δηλαδή δίνεται από το στάδιο της λεκτικής ανάλυσης. Πέρα από την κατασκευή του abstract syntax tree που αντιστοιχεί στο πρόγραμμα, στη φάση αυτή δίνεται σαν έξοδος και μια ένδειξη που υποδηλώνει αν το πρόγραμμα είναι συνακτικά ορθό ή όχι. Το αν το πρόγραμμα είναι συντακτικά ορθό καθορίζεται από το αν ακολουθεί τους κανόνες της γραμματικής της γλώσσας.

2.1.3 Σημασιολογική ανάλυση

Η σημασιολογία σε μια γλώσσα προγραμματισμού ασχολείται με θέματα ερμηνείας των καλώς σχηματισμένων προγραμμάτων. Αντίθετα από τη σύνταξη της γλώσσας, η οποία ορίζεται με τυπικό τρόπο μέσω context free γραμματικών, η σημασιολογία ορίζεται συνήθως με άτυπο τρόπο χρησιμοποιώντας περιγραφές σε φυσική γλώσσα. Η σημασιολογία μιας γλώσσας διακρίνεται σε στατική και σημασιολογική.

Ένα πρόγραμμα αν και συντακτικά σωστό είναι πιθανό να περιέχει σφάλματα σημασιολογικής φύσης. Για παράδειγμα αν και στις περισσότερες γλώσσες δεν επιτρέπονται πράξεις μεταξύ αριθμών και χαρακτήρων, αυτός ο κανόνας δεν μπορεί να ελεγχθεί στο στάδιο της συντακτικής ανάλυσης. Η στατική σημασιολογία είναι το τμήμα της σημασιολογικής περιγραφής μιας γλώσσας προγραμματισμού που ασχολείται με θέματα απόδοσης ερμηνείας στα σύμβολα της γραμματικής. Ο κύριος σκοπός της στατικής σημασιολογίας είναι ο εντοπισμός σημασιολογικών σφαλμάτων κατά τη διάρκεια της μεταγλώττισης.

Πέρα από τον έλεγχο τύπων και τον εντοπισμό σημασιολογικών σφαλμάτων, είναι επίσης σημαντικό να περιγραφεί η συμπεριφορά των προγραμμάτων κατά τη διάρκεια της εκτέλεσής τους. Αυτό είναι το αντικείμενο της δυναμικής σημασιολογίας (dynamic semantics). Πιο συγκεκριμένα: η δυναμική σημασιολογία περιγράφει τη λειτουργία ενός προγράμματος το οποίο είναι συνακτικά σωστό και δεν έχει στατικά σημασιολογικά λάθη.

Η φάση της σημασιολογικής ανάλυσης (semantic analysis) ενός προγράμματος έχει ως κύριο σκοπό το σημασιολογικό έλεγχο του, δηλαδή τον εντοπισμό σημασιολογικών σφαλμάτων κατά τη διάρκεια της μεταγλώττισης. Εστιάζει δηλαδή περισσότερο σε θέματα στατικής σημασιολογίας της αρχικής γλώσσας προγραμματισμού. Καθώς όμως έχει σαν επιμέρους στόχους το να διευκολύνει την παραγωγή κώδικα και να επιτρέψει τη βελτιστοποίηση του, προσεγγίζει επιπλέον και θέματα που αφορούν τη δυναμική σημασιολογία της αρχικής γλώσσας. Μερικά παραδείγματα σημασιολογικών ελέγχων είναι:

- Έλεγχοι τύπων, για παράδειγμα αν σε μια πράξη οι όροι δεν είναι συμβατοί μεταξύ τους (π.χ άθροιση αριθμού με συμβολοσειρά)

- Έλεγχοι ύπαρξης ονομάτων, για παράδειγμα αν μια μεταβλητή που χρησιμοποιείται έχει οριστεί προηγουμένως στο πρόγραμμα

- Έλεγχοι μοναδικότητας, για παράδειγμα σε κάποιες γλώσσες πρέπει να εμφανίζεται ένδειξη σφάλματος αν μια μεταβλητή δηλώνεται δύο φορές μέσα στην ίδια εμβέλεια

- Έλεγχοι ροής, για παράδειγμα στη γλώσσα C πρέπει να εμφανιστεί σφάλμα αν η εντολή break χρησιμοποιείται εκτός κάποιας δομής όπως βρόχος επανάληψης

2.1.4 Παραγωγή ενδιάμεσου κώδικα

Οι περισσότεροι μεταγλωττιστές επιλέγουν αντί να μεταφράζουν το αρχικό πρόγραμμα κατευθείαν στην τελική γλώσσα, να το μεταφράζουν αρχικά σε ένα ισοδύναμο σε μια ενδιάμεση γλώσσα (intermediate laguage) και να κάνουν στη συνέχεια τη μετάφραση στην τελική γλώσσα. Η ενδιάμεση γλώσσα είναι επιπέδου ενδιάμεσο αυτού της αρχικής και της τελικής γλώσσας. Αυτή η φάση του μεταγλωττιστή ασχολείται με την παραγωγή του ενδιάμεσου κώδικα (intermediate code) για το ισοδύναμο πρόγραμμα. Οι λόγοι που γίνεται αυτή η ενδιάμεση μετάφραση είναι οι ακόλουθοι:

- Διευκολύνεται η μετάφραση καθώς γίνεται σε δύο τμήματα αντί για ένα.

- Η βελτιστοποίηση του παραγόμενου κώδικα είναι πιο εύκολη και πιο αποδοτική όταν εφαρμόζεται στον ενδιάμεσο κώδικα. Αυτό δεν εμποδίζει περαιτέρω βελτιστοποιήσεις στον τελικό κωδικα.

- Διευκολύνεται ο διαχωρισμός του μεταγλωττιστή σε εμπρός και πίσω τμήμα. Η ενδιάμεση γλώσσα είναι το κοινό σύνορο μέσω του οποίου μπορούν αυτά τα δύο τμήματα να επικοινωνούν.

2.1.5 Παραγωγή τελικού κώδικα

Η τελευταία φάση της μεταγλώττισης ενός προγράμματος είναι η παραγωγή του τελικού κώδικα (code generation). Η γεννήτρια παραγωγής του τελικού κώδικα πρέπει να παράγει σωστό και αποδοτικό κώδικα και να το κάνει αποτελεσματικά με βάση το χρόνο και τη μνήμη που απαιτείται. Η γεννήτρια τελικού κώδικα δέχεται στην είσοδο της το πρόγραμμα σε ενδιάμεσο κώδικα και παράγει τον κώδικα στην τελική γλώσσα που είχαμε ορίσει. Ένα συχνό σενάριο είναι η γλώσσα αυτή να είναι η γλώσσα μηχανής του συστήματος στο οποίο χρησιμοποιούμε τον μεταγλωττιστή. Γι' αυτό το λόγο ο μεταγλωττιστής πρέπει να πάρει αποφάσεις όπως ποιες μεταβλητές θα αποθηκευτούν σε καταχωρητές και ποιες στη μνήμη καθώς και να επιλέξει τις κατάλληλες εντολές μηχανής για την εκτέλεση και τη σειρά με την οποία θα εκτελεστούν.

2.2 Έλεγχος ορθής λειτουργίας - Testing

Ο έλεγχος ενός λογισμικού εκτελείται προκειμένου να επιβεβαιωθεί και να επαληθευτεί η ποιότητα του. Ως έλεγχο λογισμικού ονομάζουμε τη διαδικασία στην οποία εκτελούμε ένα πρόγραμμα με σκοπό να εντοπίσουμε σε αυτό λάθη. Ο έλεγχος λογισμικού (software testing) χωρίζεται σε δύο κύριες κατηγορίες: black-box testing και white-box testing [Nidh12].

2.2.1 Black box testing

Το black-box testing ή αλλιώς functional testing είναι η διαδικασία ελέγχου ενός ολοκληρωμένου λογισμικού στην οποία τα test cases που θα χρησιμοποιηθούν για τον έλεγχο σχεδιάζονται με βάση τις απαιτήσεις του συστήματος και δεν υπάρχει εικόνα του κώδικα του συστήματος το οποίο ελέγχεται. Ονομάστηκε black-box testing γιατί θεωρούμε το σύστημα που ελέγχεται σαν ”μαύρο κουτί” και δεν ασχολούμαστε με την εσωτερική του δομή.

Το black box testing χρησιμοποιείται για να επαθηλευτεί η λειτουργικότητα του συστήματος. Κατά την εκτέλεση του μπορούν να ανιχνευτούν προδιαγραφές του συστήματος που προκαλούν προβλήματα για να ξεκινήσει η διαδικασία της διόρθωσης τους. Το black box testing γίνεται από την πλευρά του τελικού χρήστη προκειμένου να διαπιστωθεί αν το σύστημα εκτελεί τις λειτουργίες που του έχουν οριστεί χωρίς λάθη. Αξίζει να σημειωθεί ότι κατά τη διαδικασία αυτή χρησιμοποιούνται σωστά και λανθασμένα παραδείγματα για να ελεγχθεί η λειτουργικότητα.

Το black box testing ελέγχει ένα σύστημα από την αρχή ως την τελική του έξοδο. Το πλεονέκτημα της μεθόδου αυτής είναι ότι οι testers δε χρειάζεται να έχουν κάποια γνώση για το πως υλοποιείται το σύστημα που ελέγχουν. Οι προγραμαμτιστές που σχεδιάζουν το σύστημα και αυτοί που κάνουν τον έλεγχο δεν συνεργάζονται. Ένα άλλο πλεονέκτημα αυτής της μεθόδου είναι ότι βοηθάει στο να ανακαλυφθούν ασάφειες και λάθη στις προδιαγραφές του συστήματος πριν αυτό τεθεί σε λειτουργία.

2.2.2 White box testing

Στο white box testing ή αλλιώς structural testing το κομμάτι του λογισμικού που ελέγχεται αντιμετωπίζεται σαν ”λευκό κουτί”. Αυτό σημαίνει ότι αυτός που κάνει τον έλεγχο έχει εποπτεία για το πως υλοποιείται το σύστημα που ελέγχει. Η επιλογή των test cases σε αυτή την περίπτωση γίνεται με βάση την υλοποίηση του λογισμικού. Με το white box testing μπορούμε να ελέγξουμε συγκεκριμένα κομμάτια του κώδικα ξεχωριστά από το υπόλοιπο πρόγραμμα, όπως μεμονωμένες συναρτήσεις. Για να είναι αποτελεσματικό πρέπει να εκτελεστεί από έναν προγραμματιστή με πολύ καλή κατανόηση της εσωτερικής δομής του συστήματος, εφόσον τα test case που θα γραφούν θα πρέπει να είναι αρκετά εύστοχα ώστε να καλύπτουν κάθε κομμάτι του κώδικα.

Το white box testing χρησιμοποιείται κυρίως για να εντοπιστούν λογικά λάθη στον κώδικα. Είναι χρήσιμο ώστε να κάνουμε debugging τον κώδικα, να βρίσκουμε τυπογραφικά λάθη και να ανακαλύπτουμε ποιες παραδοχές ήταν λάθος κατά τη συγγραφή του προγράμματος. Μπορεί να εφαρμοστεί από χαμηλό επίπεδο του λογισμικού όπως σε μεμονωμένα units στον κώδικα καθώς και σε ανώτερα στάδια.

2.3 Μέθοδοι ελέγχου μεταγλωττιστή

Υπάρχουν δύο τρόποι για να εξεταστεί η ορθότητα ενός μεταγγλωτιστή. Ο ένας είναι με verification και translation validation και ο άλλος με εκτεταμένο testing [Rege].

2.3.1 Μέθοδος του formal verification

Η επαληθευμένη μεταγλώττιση είναι ένας σίγουρος τρόπος να βεβαιωθούμε ότι ο μεταγλωττιστής είναι έμπιστος, αλλά ενέχει μεγάλη δυσκολία στην υλοποίηση της. Απαιτείται μαθηματική περιγραφή της αρχικής και τελικής γλώσσας προγραμματισμού, καθώς και μια απόδειξη ότι κάθε πρόγραμμα γραμμένο στην γλώσσα εισόδου μεταφράζεται σωστά στην τελική γλώσσα. Παρά την ύπαρξη από compilers οι οποίοι είναι επιβεβαιωμένοι θεωρητικά, η μέθοδος καθιστά έναν τέτοιο compiler πολύ δύσκολο να εξελιχθεί. Σαν συνέπεια η μέθοδος αυτή δεν χρησιμοποιείται ευρέως. Περισσότερες λεπτομέρειες σχετικά με επαληθευμένη μεταγλώττιση στα [vKar90, Lero08].

2.3.2 Μέθοδος του testing

Αυτό μας οδηγεί στη δεύτερη μέθοδο, η οποία είναι και αυτή που επιλέξαμε να υλοποιήσουμε σε αυτή τη διπλωματική εργασία, τον έλεγχο του μεταγγλωτιστή μέσω εκτεταμένου testing. Σε αυτήν τρέχουμε τον μεταγλωττιστή με διάφορα παραδείγματα (tests) σαν είσοδο και ελέγχουμε το αποτέλεσμα. Η μέθοδος αυτή είναι αρκετά εύκολο να εφαρμοστεί στην πράξη και δεν απαιτεί μεγάλη προσπάθεια από την πλευρά του προγραμματιστή, καθώς η μόνη δύσκολη εργασία είναι να παράξει μια μεγάλη ποικιλία από παραδείγματα ώστε να καλύπτει κάθε σενάριο χρήσης του compiler. Σαν μέθοδος έχει εμφανά οφέλη: αφού ο μεταγλωττιστής μεταφράζει σωστά τα παραδείγματα που του δίνουμε, θα μπορεί με μεγάλη πιθανότητα να δουλέψει σωστά σε κάθε πρόγραμμα που θα του δώσει ο χρήστης όταν βγει στην παραγωγή. Όμως παράλληλα η μέθοδος του testing έχει μια εμφανή αδυναμία καθώς δεν εγγυάται αποδεδειγμένα ότι ο μεταγλωττιστής καλύπτει κάθε πιθανό πρόγραμμα που μπορεί να δημιουργήσει η γλώσσα εισόδου. Η μέθοδος βασίζεται στην βάση παραδειγμάτων (test suite) που χρησιμοποιεί ο προγραμματιστής, επομένως αν αυτή δεν καλύπτει όλες τις πιθανές περιπτώσεις τότε υπάρχει ο κίνδυνος ο μεταγλωττιστής να εμφανίσει στο μέλλον σφάλμα όταν συναντήσει μια περίπτωση για την οποία δεν είχε γίνει έλεγχος.

Η μέθοδος του testing για τον έλεγχο μεταγλωττιστών έχει τρία σημαντικά ζητήματα που πρέπει να αντιμετωπιστούν αν θέλουμε το σύστημα μας να είναι αποτελεσματικό, όπως αναφέρονται στο [Koss05].

1. Δημιουργία των test cases (test case generation)

2. Ένας τρόπος ώστε να γνωρίζουμε αν ο μεταγλωττιστής πέρασε ή οχι ένα test (test oracle)

3. Εκτίμηση της ποιότητας του test suite

Ξεκινώντας την ανάλυση από την αρχή, η παραγωγή των test cases είναι σημαντική καθώς αποτελεί τη βάση πάνω στην οποία θα στηριχτούμε για να υλοποιήσουμε τον έλεγχο του μεταγλωττιστή. Στα πλαίσια του ελέγχου μεταγλωττιστών ένα test case περιλαμβάνει τον σκοπό ή αλλιώς περιγραφή του test, το test input που αποτελείται από το πρόγραμμα που δίνεται σαν είσοδο στον μεταγγλωτιστή ώστε να επιβεβαιωθεί η συμπεριφορά του και τέλος την αναμενόμενη έξοδο (output) για το συγκεκριμένο πρόγραμμα εισόδου. Στην αναφορά αυτή στη συνέχεια, για χάριν ευκολίας, θα αναφέρεται το πρόγραμμα εισόδου του test case απλώς σαν test case. Περισσότερα για την παραγωγή των test cases αναφέρονται στο [Bouj97], όπως για παράδειγμα στρατηγικές που ακολουθούνται για test case generation, επιλογή ενός υποσυνόλου των test cases για να σχηματιστεί το test suite που χρησισμοποιείται στον έλεγχο κλπ.

Το test verdict αποτελεί το κριτήριο για το αν ο μεταγλωττιστής εμφάνισε την αναμενόμενη έξοδο για ένα δεδομένο test case. Τότε λέμε ότι ο μεταγλωττιστής πέρασε το συγκεκριμένο test. Το κριτήριο αυτό μπορεί για παράδειγμα να είναι ένα μήνυμα ότι η εκτέλεση ολοκληρώθηκε σωστά ή ότι η έξοδος του παραγόμενου εκτελέσιμου από τον μεταγλωττιστή είναι σωστή για τα έγκυρα προγράμματα είσοδου ή ένα σχετικό μήνυμα λάθους για τις μη έγκυρες εισόδους.

Για την ποιότητα του test suite που χρησιμοποιούμε, το κριτήριο για την εκτίμηση της ποιότητας του είναι το κατά πόσο καλύπτει τα χαρακτηριστικά της γλώσσας που καλείται να μεταφράσει ο compiler. Το κριτήριο αυτό ονομάζεται coverage criterion στη βιβλιογραφία [Koss05] και χρησιμεύει επίσης σαν ένδειξη για το πότε το test suite μας καλύπτει επαρκώς τις ανάγκες του ελέγχου που θέλουμε να κάνουμε ώστε να μη χρειάζεται να προστεθούν περαιτέρω test cases σε αυτό. Το coverage criterion μπορεί να είναι διαφορετικό για τις ενδιάμεσες φάσεις ενός μεταγλωττιστή από αυτό για τον έλεγχο του πλήρους μεταγγλωτιστή. Για παράδειγμα για τον έλεγχο του Parser χρειαζόμαστε test cases που να καλύπτουν κάθε κανόνα της γραμματικής της γλώσσας για να κατασκευάσουμε το σύνολο των έγκυρων tests.

Στην εργασία αυτή ο έλεγχος γίνεται χρησιμοποιώντας μια υποπερίπτωση του software testing, το black box testing. Ένα από τα μειονεκτήματα του black box testing είναι η ανάγκη από τον προγραμ-

ματιστή του λογισμικού να ορίσει ξεκάθαρα την είσοδο του. Αυτή η δυσκολία όμως δεν παρουσιάζεται στην περίπτωση των compilers, αφού για μια δεδομένη γλώσσα προγραμματισμού έχουμε στη διάθεση μας τη γραμματική της και έναν ορισμό για τη σημασιολογία που ακολουθεί. Αυτό κάνει τη μέθοδο του black box testing καλή επιλογή για τον έλεγχο της ορθότητας ενός compiler. Τα παραπάνω προσδίδουν έναν ακόμα λόγο για τον οποίο επιλέχθηκε να αξιοποιηθεί αυτός ο τρόπος ελέγχου στο σύστημα που υλοποιήθηκε στα πλαίσια αυτής της εργασίας.

2.4 Γνωστοί μέθοδοι ελέγχου μεταγλωττιστή μέσω testing

Σε αυτό το κομμάτι αναφέρουμε μερικές γνωστές μεθόδους testing από τη βιβλιογραφία [Chen16]. Για συντομία στο ακόλουθο κομμάτι της εργασίας αναφερόμαστε με τον όρο test program στα προγράμματα που δίνονται σαν είσοδο στους μεταγλωττιστές και με τον όρο test inputs στα προγράμματα που δίνονται σαν είσοδο στα test programs.

1. Randomized Differential Testing (RDT)
2. Different Optimization Levels (DOL)
3. Equivalence Modulo Inputs (EMI)

2.4.1 Randomized Differential Testing

Το Randomized Differential Testing (RDT) είναι μια γνωστή τεχνική ελέγχου μεταγλωττιστών που αντιμετωπίζει το πρόβλημα του test oracle με το να χρησιμοποιεί δύο ή περισσότερους παρόμοιους μεταγλωττιστές οι οποίοι να υλοποιούν μετάφραση της ίδιας γλώσσας. Οι compilers αυτοί θα πρέπει να παράγουν τα ίδια αποτελέσματα για το ίδιο σύνολο από tests εισόδου, οπότε μπορούμε να ξεχωρίσουμε ποιος από τους compilers περιέχει bugs παρατηρώντας τα αποτελέσματα της πλειοψηφίας. Αυτό σημαίνει ότι αν περισσότεροι από τους μισούς compilers παράγουν τα ίδια αποτελέσματα αυτά τα αποτελέσματα λαμβάνονται ως σωστά.

Αν έχουμε σαν είσοδο ένα σύνολο από compilers $C_1, C_2, \ldots, C_n$ με $n \geq 3$, η διαδικασία για να εφαρμόσουμε RDT για τον έλεγχο αυτών των compilers είναι η ακόλουθη. Κάθε compiler C_i μεταγλωττίζει ένα πρόγραμμα test P και παράγει ένα εκτελέσιμο E_i, με $1 \leq i \leq n$. Για κάθε σύνολο από test inputs για το P που ονομάζεται I, τα εκτελέσιμα αυτά παράγουν διαφορετικά αποτελέσματα τα οποία χαρακτηρίζουμε ως $O_1, O_2, \ldots, O_n$. Επειδή οι compilers που ελέγχονται είναι σχεδιασμένοι ώστε να ακολουθούν τις ίδιες προδιαγραφές, οι συμπεριφορές τους πρέπει να είναι ίδιες. Επομένως, το RDT ανιχνεύει τα bugs στους compilers μέσω της μεθόδου της πλειοψηφίας για τα $O_1, O_2, \ldots, O_n$.

Το RDT είναι μια γνωστή τεχνική testing που μπορεί να εφαρμοστεί όχι μόνο σε μεταγλωττιστές αλλά και σε άλλα κομμάτια περίπλοκου λογισμικού λόγω της αποτελεσματικότητας και της ευκολίας υλοποίησης του. Παρ'όλα αυτά έχει έναν σημαντικό περιορισμό. Επειδή οι μεταγλωττιστές που χρησιμοποιεί για τον έλεγχο μπορεί να έχουν υλοποιηθεί με παρόμοιο τρόπο ή μπορεί να περιέχουν ακόμα και κοινό πηγαίο κώδικα, είναι πιθανό να παράγουν ίδια λανθασμένα αποτελέσματα για κάποιο συγκεκριμένο test. Σε αυτή την περίπτωση το RDT αδυνατεί να εντοπίσει τα bugs που υπάρχουν. Επιπλέον αν υπάρχει κάποια γλώσσα που για κάποιο λόγο διαθέτει μόνο έναν compiler διαθέσιμο, η μέθοδος του RDT δεν μπορεί να εφαρμοστεί.

2.4.2 Different Optimization Levels

Η μέθοδος του DOL θεωρείται παραλλαγή του RDT. Εδώ αντί να συγκρίνονται αποτελέσματα από διαφορετικούς compilers, συγκρίνονται αποτελέσματα όταν εφαρμόζονται διαφορετικά επίπεδα από optimizations στον ίδιο compiler. Έτσι κάνοουμε compile ένα test program με διαφορετικά επίπεδα optimization και το εκτελούμε το με το ίδιο σύνολο από test inputs. Αν παραχθούν διαφορετικά αποτελέσματα αυτό σημαίναι πως ο μεταγλωττιστής περιέχει λάθη. Θα αναφερόμαστε σε αυτά τα

λάθη ως optimization bugs για να τα διαχωρίσουμε από τα bugs που προκύπτουν από διαφορετικά αίτια.

Όπως και το RDT, η μέθοδος του DOL χρησιμοποιεί ένα test program για κάθε έλεγχο, καθιστώντας την εύκολο να υλοποιηθεί. Επιπλέον, το DOL στοχεύει στον έλεγχο ενός μεμονωμένου compiler στον οποίο εφαρμόζονται διαφορετικά επίπεδα από optimizations, οπότε δίνει τη δυνατότητα ελέγχου αυτού του compiler εις βάθος χωρίς να απαιτείται η ύπαρξη άλλων compilers.

2.4.3 Equivalence Modulo Inputs

Το EMI είναι μια καινούργια μέθοδος ελέγχου μεταγλωττιστών η οποία αντιμετωπίζει το ζήτημα του test oracle συγκρίνοντας ένα test program και κάποιες παραλλαγές του, οι οποίες θεωρείται ότι έχουν ισοδύναμη συμπεριφορά όταν δέχονται το ίδιο σύνολο από test inputs. Πιο συγκεκριμένα, για ένα δεδομένο test program, το EMI αναγνωρίζει ένα σύνολο από statements που επηρεάζουν τη συμπεριφορά του για συγκεκριμένα test εισόδου (test inputs), και κατασκευάζει παραλλαγές του προγράμματος οι οποίες έχουν συμπεριφορά ισοδύναμη με τη δική του.

Για έναν δεδομένο compiler C η διαδικασία ελέγχου είναι η ακόλουθη. Για κάθε test program P και τα test inputs που του αντιστοιχούν, συμβολίζονται ως I, το EMI παράγει αρχικά μερικές παραλλαγές του P οι οποίες συμβολίζονται ως $Q_1, Q_2, \ldots, Q_m$, μέσω της ακόλουθης διαδικασίας. Αρχικά το EMI ταυτοποιεί ένα σύνολο από statements στο P που δεν έχουν εκτελεστεί από το I και παράγει παραλλαγές του προγράμματος διαγράφοντας τυχαία εντολές από αυτό το σύνολο. Έπειτα το test program P και κάθε παραλλαγή του Q_i ($1 \leq i \leq m$) μεταγλωττίζονται από τον μεταγλωττιστή C και παράγονται τα αντίστοιχα εκτελέσιμα E_p και E_i. Παίρνοντας τα I σαν test inputs, τα εκτελέσιμα αυτά παράγουν αποτελέσματα που συμβολίζονται ως O_p και O_i με $1 \leq i \leq m$. Επειδή οι παραλλαγές του προγράμματος πρέπει να έχουν ισοδύναμη συμπεριφορά με το πρόγραμμα P υπό τα test inputs I, το EMI ανιχνεύει τα bugs στον C με το να συγκρίνει κάθε ζευγάρι O_p και O_i, $1 \leq i \leq m$.

Παρά το ότι το EMI είναι πιο περίπλοκη μέθοδος από τα RDT και DOL, το EMI ελέγχει μόνο έναν μεταγλωττιστή ανεξαρτήτως των επιπέδων των optimizations. Έτσι έχει τη δυνατότητα να ελέγξει κατευθείαν τον μεταγλωττιστή που θέλει χωρίς να περιορίζεται από τις δυσκολίες που έχουν τα RDT και DOL.

Κεφάλαιο 3

Λειτουργία του συστήματος ελέγχου

Σε αυτό το κεφάλαιο αναλύουμε τη λειτουργία του συστήματος μας για κάθε στάδιο του compiler το οποίο εξετάζουμε. Ο έλεγχος γίνεται για τα στάδια των Lexer, Parser, Semantic analysis και μετά γίνεται έλεγχος του compiler στο σύνολο του. Δεν γίνεται έλεγχος για το στάδιο παραγωγής ενδιάμεσου κώδικα καθώς ο κάθε προγραμματιστής μπορεί να χρησιμοποιεί διαφορετικό intermediate representation (IR), οπότε το να γινόταν έλεγχος για το καθένα ξεχωριστά θα εισήγαγε μεγάλη πολυπλοκότητα στο σύστημα. Επίσης δεν ελέγχονται τα optimizations που εκτελεί ο μεταγλωττιστής στον ενδιάμεσο και στον τελικό κώδικα, καθώς μια τέτοια λειτουργία θα προυπέθετε test cases μεγαλύτερου μεγέθους από αυτά που χρησιμοποιούμε και θα έπρεπε να είναι στοχευμένα ώστε να αναδεικνύουν το πόσο ένα συγκεκριμένο optimization βελτιώνει το χρόνο εκτέλεσης του προγράμματος. Περισσότερα για optimizations στο κομμάτι των μελλοντικών στόχων στο κεφάλαιο 3. Για τον έλεγχο γίνεται χρήση test suite το οποίο περιέχει παραδείγματα από προγράμματα γραμμένα στη γλώσσα εισόδου του compiler. Προκειμένου να εξετάσουμε το σύστημα χρησιμοποιήσαμε σαν παράδειγμα τη γλώσσα προγραμματισμού Dana. Το test suite που χρησιμοποιήσαμε είναι διαχωρισμένο σε προγράμματα που είναι σωστά και προγράμματα που περιέχουν λάθη. Ο διαχωρισμός αυτός γίνεται προκειμένου να κάνουμε δύο είδη ελέγχου σε κάθε στάδιο του compiler: να ελέγξουμε αν στα έγκυρα tests ο compiler που ελέγχεται τερματίζει κανονικά και τυπώνει τη σωστή έξοδο (αν υπάρχει) και αν στα tests με λάθη εντοπίζεται το λάθος και τερματίζεται σωστά η εκτέλεση.

Για να αξιολογήσουμε την απόδοση του compiler κρατάμε στατιστικά για το πόσα tests πέρασε και πόσα απέτυχε να περάσει. Όταν ο compiler περνάει όλα τα tests θεωρούμε ότι περνάει επιτυχώς το συγκεκριμένο στάδιο και το σύστημα εμφανίζει σχετικό μήνυμα. Τα tests επίσης είναι αριθμημένα ώστε να μπορούμε να εμφανίζουμε ποια δεν πέρασε ο compiler δίνοντας στον προγραμματιστή τη δυνατότητα να βρει εύκολα τα σημεία στα οποία έχει κάνει κάποιο λάθος.

Για την υλοποίηση του συστήματος χρησιμοποιήθηκαν scripts γραμμένα σε Bash το καθένα από τα οποία ελέγχει ένα συγκεκριμένο στάδιο του compiler. Το καθένα από τα scripts αυτά δέχεται σαν είσοδο το path στο οποίο τρέχει και το path προς τον compiler που θέλουμε να ελέγξουμε. Έχοντας αυτά τα δεδομένα, το bash script καλεί ένα αντίστοιχο εκτελέσιμο γραμμένο σε Python 3 στο οποίο γίνεται ο πραγματικός έλεγχος. Τα tests που έχουμε στη διάθεση μας από το test suite που χρησιμοποιούμε εισάγονται σε δύο λίστες χωρισμένα σε έγκυρα και μη έγκυρα tests. Εκτελούμε τον έλεγχο για το καθένα από τα παραδείγματα αυτά μέσα σε έναν επαναληπτικό βρόχο μέχρι να εξαντλήσουμε τη λίστα. Σε κάθε test ανάλογα με τον κωδικό που γυρίζει το πρόγραμμα στην έξοδο (exit code) εκτελούμε την κατάλληλη ενέργεια. Πιο συγκεκριμένα ,για τον έλεγχο των έγκυρων παραδειγμάτων, αν έχουμε exit code 1 θεωρούμε ότι ο compiler αντιμετώπισε σφάλμα που δε θα έπρεπε να υπάρχει οπότε σημειώνεται ότι ο compiler δεν κατάφερε να περάσει το συγκεκριμένο test. Αντίστροφα αν ο κώδικός ήταν 0 τότε σημειώνεται ότι ο compiler πέρασε το συγκεκριμένο test χωρίς πρόβλημα. Ο κώδικας γι' αυτό το κομμάτι παρατίθεται στο Παράρτημα Α. Για τα tests που περιέχουν λάθη η υλοποίηση γίνεται αντίστοιχα με τη διαφορά ότι αντιστρέφεται σε ποιο exit code θεωρείται ότι ο compiler έχει περάσει το test.

Ο πλήρης κώδικας για την υλοποίηση του συστήματος βρίσκεται στο Github:

`https://github.com/giorg0skar/Diploma`

3.1 Παραδοχές

Προκειμένου το σύστημα να λειτουργήσει με τον επιθυμητό τρόπο γίνονται ορισμένες παραδοχές σχετικά με τον compiler που δέχεται ως είσοδο.

1. Ο προγραμματιστής επιστρέφει exit code 1 αν ο compiler τερματίσει λόγω error. Διαφορετικά επιστρέφει exit code 0.

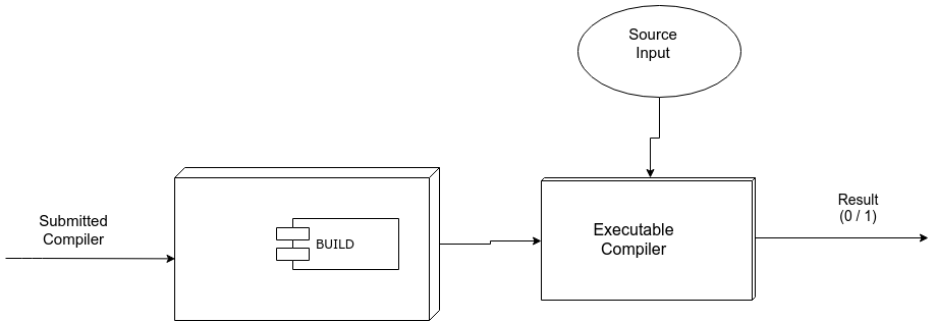
2. Ο προγραμματιστής δημιουργεί το εκτελέσιμο πρόγραμμα που παράγει ο compiler στο ίδιο directory (ή πρέπει να διευκρινίσει σε ποιο directory βρίσκεται).

3. Ο προγραμματιστής πρέπει να παρέχει το όνομα του executable αρχείου που καλεί τον πλήρη compiler σε ένα text αρχείο που ονομάζεται EXECUTABLE. Αντίστοιχα για τα επιμέρους στάδια του compiler πρέπει να δίνεται το όνομα του εκτελέσιμου σε ένα αρχείο που θα λέγεται COMPILER. Αν κάποιο από αυτά δεν υπάρχει θεωρείται πως ονομάζεται απλώς *compiler*.

4. Ο προγραμματιστής δεν έχει δικαίωμα να γράφει στο directory του Testsuite. Δε θα πρέπει λοιπόν να δημιουργεί αρχεία στο φάκελο που βρίσκονται τα source files της γλώσσας.

5. Το στάδιο του Build του συστήματος υλοποιείται μέσω Makefile. Αυτό σημαίνει ότι το σύστημα στην έκδοση που έχουμε υλοποιήσει, για να κάνει build εκτελεί την εντολή make στο directory που που περιέχει το project που έχει δωθεί. Αυτό σημαίνει ότι ο προγραμματιστής πρέπει για κάθε στάδιο του compiler καθώς και για τον συνολικό compiler, να έχει φροντίσει να παράγεται το εκτελέσιμο που θα ελεγχθεί, με τη χρήση της εντολής make χωρίς να απαιτούνται περαιτέρω εντολές. Αν επιθυμεί να τρέξουν εντολές κατά τη διαδικασία του ελέγχου πρέπει να τις συμπεριλάβει στο script το οποίο θα δηλώσει στα αρχεία COMPILER ή EXECUTABLE.

Παρατίθεται μια σχηματική απεικόνιση του συστήματος που κατασκευάζουμε, για τα στάδια του front-end ενός compiler: Lexer, Parser, Semantic analysis. Δίνεται ο compiler σε μορφή πηγαίου κώδικα και αφού γίνεται build σαν εκτελέσιμο πρόγραμμα, του δίνουμε σαν είσοδο τα tests που έχουμε στη διάθεση μας. Σαν έξοδο επιστρέφεται ένας κωδικός που έχει τιμή 1 ή 0 υποδηλώνοντας αν το test πέρασε τον έλεγχο ή όχι αντίστοιχα.



Σχήμα 3.1: Σχήμα συστήματος για στάδια ελέγχου Lexer, Parser και Semantic analysis

3.2 Lexer

Ο έλεγχος για το κομμάτι του Lexer επικεντρώνεται στο να τυπώνονται οι λεκτικές μονάδες (tokens) για κάθε λέξη του προγράμματος εισόδου. Ο χρήστης που σχεδιάζει τον compiler που εξετάζεται επιλέγει αν θα τυπώνεται ένας κωδικός για κάθε λεκτική μονάδα ή θα τυπώνεται η συμβολοσειρά που αντιστοιχεί στη συγκερκιμένη λεκτική μονάδα η οποία ονομάζεται lexeme. Έπειτα γίνεται έλεγχος του output του lexer με ενός σωστού lexer στον οποίο δώθηκε το ίδιο πρόγραμμα εισόδου. Αν οι δύο έξοδοι είναι ίδιες τότε ο Lexer θεωρείται ότι λειτούργησε σωστά. Διαφορετικά το αποτέλεσμα

του μεταγλωττιστή θεωρείται λάθος. Αν ένας compiler περάσει όλα τα tests χωρίς να βρεθεί λάθος έξοδος θεωρείται πως το κομμάτι της λεκτικής ανάλυσης δουλεύει σωστά.

Αξίζει να σημειωθεί πως το στάδιο του Lexer είναι το μοναδικό που απαιτεί στη γενική περίπτωση την ύπαρξη ενός αξιόπιστου lexer προκειμένου να ελέγξει αυτόν που έχει γράψει ο προγραμματιστής. Αυτό συμβαίνει γιατί ο προγραμματιστής σε αυτό το στάδιο δεν έχει κάποιο κριτήριο για να αναγνωρίσει αν το πρόγραμμα του έβγαλε το σωστό αποτέλεσμα.

Συγκεκριμένα για Dana:

Εφόσον το σύστημα δοκιμάστηκε χρησιμοποιώντας compiler της γλώσσας Dana, προστέθηκε μια επέκταση στον έλεγχο του Lexer. Αυτή η επέκταση δεν υπάρχει στην γενική έκδοση του συστήματος, αλλά προστέθηκε για διευκόλυνση ελέγχου του συγκεκριμένου σταδίου για τους μεταγλωττιστές της Dana. Στην περίπτωση που η έξοδος του χρήστη είναι διαφορετική από την αναμενόμενη γίνεται έλεγχος για το αν τα διαφορετικά σημεία οφείλονται σε tokens που έχουν να κάνουν με την αρχή blocκ κώδικα.

3.3 Parser

Για το κομμάτι του Parser ελέγχουμε αν ο compiler μπορεί να εκτελέσει το στάδιο της συντακτικής ανάλυσης σωστά. Έχοντας επίγνωση ποια test cases είναι σωστά συντακτικά και ποια όχι από την test suite που έχουμε διαθέσιμη, γνωρίζουμε εκ των προτέρων ποια tests θα έπρεπε να εμφανίσουν λάθη σε αυτό το στάδιο. Ο έλεγχος του parser γίνεται ελέγχοντας τον κωδικό εξόδου που θα έχει αν τον εκτελέσουμε με είσοδο ένα πρόγραμμα από τα tests. Αν ο κωδικός εξόδου είναι 0 αυτό σημαίνει πως ο parser είχε την αναμενόμενη συμπεριφορά, δηλαδή αν βρισκόμαστε σε test case χωρίς λάθη το πρόγραμμα τερματίσε κανονικά, ενώ αν το test εισόδου περιείχε λάθος ο parser το αναγνώρισε σωστά. Σε κάθε περίπτωση τυπώνουμε το αντίστοιχο μήνυμα για να ενημερώσουμε τον χρήστη. Αν ο κωδικός είναι 1 αυτό σημαίνει ότι έχουμε εσφαλμένη συμπεριφορά. Αν βρισκόμαστε σε error free παράδειγμα αυτό σημαίνει ότι ο parser τερμάτισε λόγω σφάλματος το οποίο δεν υπάρχει, ενώ αν το παράδειγμα είχε συντακτικό σφάλμα στον κώδικα τότε συμπεραίνουμε ότι ο συντακτικός αναλυτής δε το αναγνώρισε. Και στις δύο περιπτώσεις τυπώνεται σχετικό μήνυμα προκειμένου ο προγραμματιστής να μπορέσει να ενημερωθεί για το bug στον parser που έχει γράψει ώστε να μπορέσει να το διορθώσει.

3.4 Semantic analysis

Για το κομμάτι της σημασιολογικής ανάλυσης (semantic analysis) ελέγχουμε αν ο compiler μπορεί να ελέγξει αν ένα πρόγραμμα είναι σημασιολογικά σωστό. Γνωρίζουμε από το test suite ξανά ποια tests είναι σημασιολογικά σωστά και ποια όχι. Έχοντας αυτό σαν γνώμονα τρέχουμε τον compiler με είσοδο τα tests της γλώσσας που εξετάζουμε και ελέγχουμε τον κωδικό που επιστρέφει στην έξοδο το πρόγραμμα. Για τα test case που δεν περιέχουν σφάλματα αν ο κωδικός είναι 0 αυτό σημαίνει πως το πρόγραμμα τερμάτισε κανονικά χωρίς να βρει κάποιο σφάλμα οπότε η εκτέλεση του είναι σωστή. Αν ο κωδικός είναι 1 αυτό σημαίνει πως κάποιο error βρέθηκε κατά την εκτέλεση και τυπώνεται το μήνυμα λάθους που έχει ορίσει ο προγραμματιστής. Το ανάποδο ισχύει για τα test που περιέχουν σημασιολογικά σφάλματα. Με κωδικό εξόδου 1 θεωρούμε πως το πρόγραμμα αναγνώρισε σωστά το λάθος, ενώ για κωδικό εξόδου 0 η εκτέλεση τερμάτισε χωρίς να βρεθεί το σφάλμα οπότε σημειώνεται ότι σε αυτό το case παρουσιάζεται λανθασμένη συμπεριφορά.

Σε αυτό το σημείο πρέπει να επισημανθεί μια παραδοχή του συστήματος ελέγχου που χρησιμοποιούμε. Κατά τον έλεγχο των test cases που περιέχουν λάθη όταν παρατηρούμε τον κωδικό 1 που επιστρέφει ο compiler κατά την έξοδο, δεν γνωρίζουμε ποιο ακριβώς ήταν το σφάλμα που αντιμετώπισε και τερμάτισε την εκτέλεση. Για να είμαστε σίγουροι ότι κάνουμε σωστό έλεγχο φροντίζουμε να έχουμε επαρκές αριθμό απο tests, το καθένα από τα οποία θα ελέγχει για ένα συγκεκριμένο τύπο λάθους. Εφόσον δεν υπάρχει κάποιο επαναλαμβανόμενο λάθος μπορεί να εντοπιστούν τα μεμονωμένα λάθη από το ποια tests τερματίζουν με λάθος exit code. Με αυτόν τον τρόπο θα είμαστε βέβαιοι

ότι αν ο compiler περάσει αυτό το στάδιο ελέγχου, θα αναγνωρίζει επιτυχώς σημασιολογικά λάθη σε προγράμματα αυτής της γλώσσας.

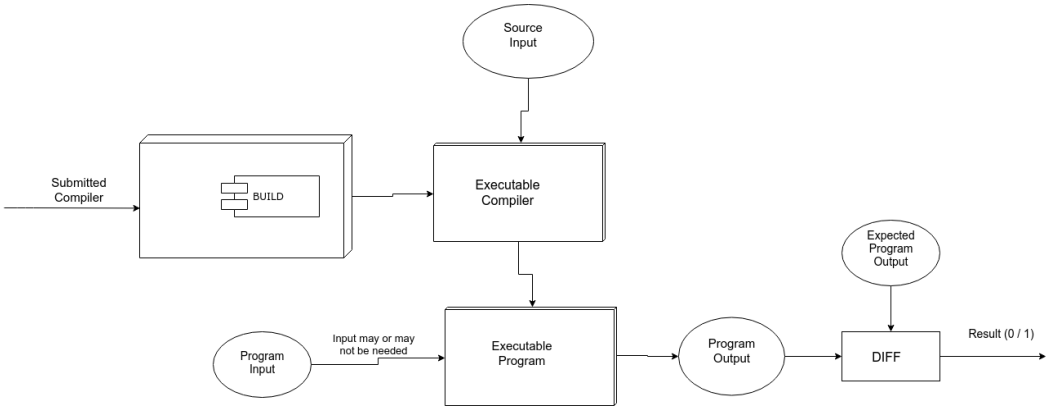
```
STATISTICS

Correct cases: 1 - 42
Cases with errors: 43 - 64

Number of correct test cases: 40/64
Number of failed test cases: 24/64 : [1, 5, 11, 12, 13, 17, 18, 19, 21, 23, 24, 26, 28, 29, 37, 40, 41, 43, 46, 48, 53, 56, 58, 61]
SEMANTIC: failed
```

Σχήμα 3.2: Αποτελέσματα ελέγχου της φάσης της σημασιολογικής ανάλυσης

3.5 Έλεγχος ολοκληρωμένου μεταγλωττιστή

Έχοντας ελέγξει τα στάδια του ”frontend” του compiler προχωράμε τέλος στον συνολικό του έλεγχο. Αυτό σημαίνει πως τρέχουμε το εκτελέσιμο αρχείο που έχει ορίσει ο προγραμματιστής το οποίο περιέχει τον πλήρη compiler και γίνεται η μετάφραση του αρχείου εισόδου από την αρχική γλώσσα σε ένα ισοδύναμο στην γλώσσα στόχο (target language). Στο σύστημα που έχει υλοποιηθεί, σε αυτό το στάδιο γίνεται η υπόθεση πως ο compiler μεταγλωττίζει το πρόγραμμα εισόδου και παράγει ένα εκτελέσιμο από τον υπολογιστή πρόγραμμα. Όπως αναφέρθηκε παραπάνω, αυτή είναι και η πιο συχνή χρήση ενός compiler στην πράξη, αφού επιτρέπει σε αυτόν που το γράφει να διαπιστώσει έμπρακτα τα αποτελέσματα του προγράμματος του.

Παρατίθεται το σχήμα που απεικονίζει πορεία εκτέλεσής του ελέγχου σε αυτό το στάδιο. Η διαφορά με τα προηγούμενα στάδια είναι πως τώρα, πέρα από το να ελέγχουμε αν ο compiler τερματίζει με είσοδο το δεδομένο test, το σύστημα συνεχίζει με τον έλεγχο του εκτελέσιμου.



Σχήμα 3.3: Σχήμα συστήματος για τον έλεγχο του πλήρη compiler

Ξεκινάμε όπως και στα προηγούμενα στάδια τρέχοντας τον compiler με είσοδο τα tests που έχουμε στη διάθεση μας. Οι διαφορές στο παρόν στάδιο είναι ότι ο χρήστης μπορεί να διευκρινίσει ένα διαφορετικό όνομα για το αρχείο που εκτελείται από ότι στα προηγούμενα στάδια. Γράφοντας στο αρχείο με το όνομα EXECUTABLE το όνομα του εκτελέσιμου, ο προγραμματιστής υποδηλώνει πιο αρχείο επιθυμεί να εκτελεστεί για να γίνει ο έλεγχος. Διαχωρίζοντας το executable του compiler από αυτό που θα εκτελεστεί στην αρχή αυτού του σταδίου ελέγχου δίνεται μεγαλύτερη ελευθερία στον προγραμματιστή να διαχειριστεί πράγματα όπως τα arguments που θα δωθούν σαν είσοδο στον compiler ή περαιτέρω μηνύματα θέλει να εμφανίζονται στην οθόνη.

Για τα tests που περιέχουν λάθη ο compiler θεωρείται ότι έχει σωστή συμπεριφορά αν επιστρέψει exit code 1 και δεν παράξει εκτελέσιμο αρχείο. Στα σωστά tests ο compiler πρέπει να επιστρέψει κωδικό 0 και να δημιουργήσει ένα εκτελέσιμο αρχείο, διαφορετικά θεωρείται ότι αυτό το case παρουσιάζει λάθος. Στη συνέχεια γίνεται έλεγχος του εκτελέσιμου που παράχθηκε, καθώς θα πρέπει να

εκτελεί τις λειτουργίες με τη λογική που έχουν διατυπωθεί στη γλώσσα που γράφτηκε. Ελέγχουμε δηλαδή την αρχή της μεταγλώττισης ενός προγράμματος που διατυπώνει ότι το τελικό πρόγραμμα πρέπει να είναι ισοδύναμο του αρχικού. Για ευκολία θεωρείται ότι το εκτελέσιμο ονομάζεται **a.out**. Τρέχουμε το αρχείο αυτό δίνοντας του μια κατάλληλη είσοδο (αν χρειάζεται) και ελέγχουμε το αποτέλεσμα που τυπώνεται στην οθόνη. Αν το αποτέλεσμα αυτό συμφωνεί με το αναμενόμενο αποτέλεσμα που έχουμε αποθηκευμένο στο Test suite (στον φάκελο με τα Outputs), τότε ο compiler θεωρείται ότι είναι σωστός για το συγκεκριμένο case. Διαφορετικά σημειώνεται ότι υπάρχει λάθος λόγω διαφορών στην έξοδο. Ο έλεγχος του αποτελέσματος γίνεται μέσω της εντολής diff του Unix, η οποία συγκρίνει το περιεχόμενο δύο αρχείων και επιστρέφει αν είναι ίδια ή όχι. Έχοντας φροντίσει να γράψουμε πρώτα την εξοδο του εκτελέσιμου a.out σε ένα αρχείο κειμένου με όνομα *user_output*, το συγκρίνουμε με την αναμενόμενη έξοδο που έχουμε στο test suite. Το τμήμα του κώδικα που διεκπεραιώνει τον έλεγχο του πλήρη compiler παρατίθεται στο παράρτημα Α.

```
36. Running correct test: 53char.dna
--------------------------------
Compiling /home/george/Documents/Diplomatiki/Diploma/Testsuite/Correct/53char.dna
/home/george/Documents/Compilers/LLVM/danac: line 60: 17921 Segmentation fault      (core dumped) ./dana
< $file > 1.ll
Error in compiling

37. Running correct test: 32byte.dna
--------------------------------
Compiling /home/george/Documents/Diplomatiki/Diploma/Testsuite/Correct/32byte.dna
(null):0: Error, type mismatch in & operation
The dana compiler is lazy and aborts...
Error in compiling

38. Running correct test: 01empty.dna
--------------------------------
Compiling /home/george/Documents/Diplomatiki/Diploma/Testsuite/Correct/01empty.dna
Compiling finished
Files user_output and 01empty.out are identical

39. Running correct test: 60hanoi.dna
--------------------------------
Compiling /home/george/Documents/Diplomatiki/Diploma/Testsuite/Correct/60hanoi.dna
Compiling finished
Files user_output and 60hanoi.out are identical
```

Σχήμα 3.4: Παράδειγμα ελέγχου πλήρη μεταγλωττιστή για τυχαία test cases

Ο έλεγχος του compiler στο σύνολο του μπορεί να γίνει και με ενεργά ένα ή περισσότερα optimizations που έχει σχεδιάσει ο προγραμματιστής. Το παραγόμενο εκτελέσιμο θα πρέπει να έχει την ίδια έξοδο με πριν οπότε η διαδικασία ελέγχου που χρησιμοποιούμε είναι ίδια. Υπάρχουν τρόποι για εκτεταμένο έλεγχο πάνω στα optimizations στον compiler, αλλά σε αυτό το σύστημα δεν έχουμε υλοποιήσει προς το παρόν κάποιον από αυτούς. Περισσότερα για σκέψεις πάνω στα optimizations στο επόμενο κεφάλαιο.

Μετά το στάδιο αυτό έχουμε αποκτήσει μια συνολική εικόνα για το αν ο μεταγλωττιστής είναι σε θέση να επιτελέσει τη λειτουργία για την οποία σχεδιάστηκε, δηλαδή να κάνει μετατροπή κώδικα από μια γλώσσα σε μια άλλη διατηρώντας τις λειτουργίες που είχαν οριστεί στη γλώσσα εισόδου. Αν η έκδοση του μεταγλωττιστή είναι συνεπής με αυτή που έτρεξε για τα προηγούμενα στάδια τότε ο προγραμματιστής έχει επιβεβαιώσει ότι το πρόγραμμα που έγραψε έχει περάσει επιτυχώς και τα πρώτα στάδια ελέγχου. Έτσι υπάρχει η επιβεβαίωση ότι ο compiler λειτουργεί ορθά, τουλάχιστον για το σύνολο των περιπτώσεων του test suite που χρησιμοποιήθηκε.

Κεφάλαιο 4

Σύνοψη

4.1 Αποτελέσματα - Συμπεράσματα

Έχοντας μελετήσει τη βιβλιογραφία γίνεται εμφανές πόσο σημαντικό είναι να μπορούμε να επιβεβαιώσουμε την ορθότητα ενός μεταγλωττιστή. Για το σκοπό αυτό έχουν μελετηθεί διάφορες προσεγγίσεις προσπαθώντας να φτάσουν σε ένα εργαλείο το οποίο θα είναι αποδοτικό. Ξεκινώντας αυτήν την εργασία είχαμε ως στόχο τη δημιουργία ενός συστήματος που θα αποτελεί χρήσιμο εργαλείο για τον προγραμματιστή που θέλει να αναπτύξει έναν compiler. Μέσα από τη δουλειά αυτή πετύχαμε την αυτοματοποίηση της διαδικασίας του testing του compiler έχοντας ως είσοδο ένα test suite με κατάλληλα test cases. Έτσι παρέχεται μια εγγύηση στον προγραμματιστή ότι το εργαλείο που χρησιμοποιεί είναι αξιόπιστο για μια πληθώρα περιπτώσεων από προγράμματα της γλώσσας.

Επιπλέον μέσα από την εκτέλεση αυτής της εργασίας καταλήξαμε σε ορισμένα συμπεράσματα. Ένα από αυτά είναι ότι το testing χρησιμοποιώντας τυχαία παραδείγματα είναι απαραίτητο για να εξακριβώσουμε την ορθότητα ενός compiler, καθώς όπως κάθε λογισμικό, έτσι κι ο compiler πρέπει να ελέγχεται με είσοδο προγράμματα παρόμοια με αυτά που θα κλειθεί να μεταφράσει όταν γίνει διαθέσιμος στο κοινό. Αν καταφέρει να περάσει αυτές τις περιπτώσεις μπορεί να μην είναι γνωστό αν είναι bug-free, αλλά θα έχουμε πετύχει σε μικρό χρονικό διάστημα την εγγύηση ότι δουλεύει σωστά στις περισσότερες περιπτώσεις. Παρά την εγγύηση αυτή όμως, όπως είναι γνωστό, το testing είναι η διαδικασία στην οποία ο προγραμματιστής προσπαθεί να βρει λάθη στο σύστημα. Έτσι θα πρέπει να έχουμε υπόψη ότι πρέπει να ελέγχεται ο μεταγλωττιστής και σε corner cases τα οποία είναι πιθανό να εμφανίσουν κάποιο σφάλμα. Στόχος μας είναι πάντα να ανακαλύπτονται τα σφάλματα στο στάδιο του ελέγχου ώστε να είναι το τελικό προιόν ασφαλές και αξιόπιστο.

Ένα άλλο συμπέρασμα στο οποίο καταλήξαμε είναι η δυσκολία που παρουσιάζεται στον έλεγχο της ορθότητας του ενδιάμεσου κώδικα του μεταγλωττιστή. Καθώς κάθε μεταγλωττιστής μπορεί να χρησιμοποιεί ξεχωριστό intermediate representation (IR), γίνεται δύσκολο να δημιουργηθεί ένα σύστημα που να λειτουργεί γενικά και να ελέγχει το στάδιο του ενδιάμεσου κώδικα. Μια πρόταση θα ήταν η επιλογή ενός αξιόπιστου IR σαν πρότυπο για κάθε compiler με στόχο να επικεντρωθούν οι προσπάθειες ελέγχου ορθότητας συγκεκριμένα σε αυτό. Ένα παράδειγμα τέτοιου IR είναι αυτό που χρησιμοποιεί το LLVM project, το οποίο διαθέτει code generation για αρκετές αρχιτεκτονικές επεξεργαστών.

4.2 Μελλοντικοί στόχοι

Παρακάτω παρουσιάζονται μερικές σκέψεις για κατευθύνσεις που μπορεί να ακολουθήσει μελλοντικά η έρευνα πάνω στο αντικείμενο της ορθότητας των μεταγλωττιστών.

Στο κεφάλαιο για την υλοποίηση του συστήματος ελέγχου αναφέρθηκε η σημασία του test suite που χρησιμοποιείται προκειμένου να ελεγχθεί ο compiler. Στην παρούσα εργασία θεωρήσαμε τα test cases που χρησιμοποιήθηκαν ως δεδομένα και δεν ασχοληθήκαμε με τον τρόπο δημιουργία τους. Μια

κατεύθυνση για μελλοντική δουλειά θα ήταν η μελέτη για τη δημιουργία μια γεννήτριας αυτόματης παραγωγής παραδειγμάτων (automatic test generation) για τον έλεγχο του μεταγλωττιστή. Ένα τέτοιο πρόγραμμα θα μπορούσε να χρησιμοποιεί αλγορίθμους όπως ορισμένοι που αναφέρονται στη βιβλιογραφία [Mall01] ή να ακολουθεί κάποια καινούργια μέθοδο. Η ύπαρξη μιας τέτοιας γεννήτριας θα έκανε την προσπάθεια του ελέγχου σημαντικά πιο εύκολη και σύντομη για τον προγραμματιστή.

Ένα άλλο πιθανό αντικείμενο για μελέτη είναι ο έλεγχος της αποδοτικότητας των optimizations που μπορεί να εφαρμόσει ο μεταγλωττιστής. Οι σύγχρονοι μεταγλωττιστές χρησιμοποιούν μια πληθώρα από optimizations προκειμένου να βελτιώσουν το παραγόμενο πρόγραμμα. Αυτό που θα θέλαμε είναι ένας τρόπος να επιβεβαιώσουμε ότι τα optimizations αυτά λειτουργούν σωστά χωρίς να παράγουν κάποιο ανεπιθύμητο αποτέλεσμα, ότι δηλαδή είχαμε σαν σκοπό και για τη λειτουργία του compiler. Μια μέθοδος για έναν τέτοιο έλεγχο περιγράφεται στο [Lern03]. Μελλοντικές έρευνες θα μπορούσαν να εστιάσουν στο να γενικευτεί ο έλεγχος αυτός για κάθε γλώσσα προγραμματισμού.

Μια άλλη ενδιαφέρουσα προσέγγιση της εγγύησης της ορθότητας ενός compiler θα ήταν να δημιουργηθεί ένα σύστημα που θα κάνει τον έλεγχο ακολουθώντας την λογική του white box testing. Σε αντίθεση με το σύστημα που υλοποιήσαμε σε αυτή την εργασία το οποίο ακολουθεί την λογική του black box testing, ένα σύστημα που θα χρησιμοποιούσε τεχνικές white box testing θα έκανε τον έλεγχο έχοντας γνώση της υλοποίησης του μεταγλωττιστή τον οποίο θα εξέταζε. Ένα τέτοιο σύστημα δε θα μπορούσε να δουλεύει με τον ίδιο τρόπο για κάθε μεταγλωττιστή, όπως αυτό που παρουσιάσαμε σε αυτήν την εργασία, αλλά θα μπορούσε να τροποποιείται κατάλληλα ώστε να εξετάζει συγκεκριμένες λειτουργικότητες ενός μεταγλωττιστή. Αυτή η προσέγγιση θα επέφερε περισσότερα οφέλη για τον έλεγχο των ενδιάμεσων σταδίων του μεταγλωττιστή καθώς εκεί ένας εκτενής έλεγχος της υλοποίησης θα έδινε μια καλύτερη εικόνα σχετικά με το αν το συγκεκριμένο κομμάτι δουλεύει σωστά από τον έλεγχο απλώς της εξόδου. Το γεγονός αυτό είναι εύκολα κατανοητό αν αναλογιστούμε ότι η έξοδος σε μερικές από τις φάσεις του μεταγλωττιστή δεν δίνει μια αρκετά περιγραφική εικόνα σχετικά με τη λειτουργία του συστήματος. Για παράδειγμα στο σύστημα που υλοποιήσαμε στα στάδια των Parser και Semantic analysis η έξοδος είναι απλώς ο exit code και κάποιο μήνυμα λάθους, αν αυτό υπάρχει, πράγμα που δεν επιτρέπει στον προγραμματιστή του compiler να καταλάβει τι συμβαίνει εσωτερικά π.χ ποιοι κανόνες της γραμματικής χρησιμοποιήθηκαν. Επομένως μια στρατηγική που θα χρησιμοποιούσε μικτό testing, black box για τον συνολικό compiler και white-box για τα επιμέρους στάδια του, θα μπορούσε να επιφέρει καλύτερα αποτελέσματα.

Βιβλιογραφία

[Bouj97] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment", *Information and Software Technology*, vol. 39, no. 9, pp. 617–625, 1997.

[Chen16] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang and Bing Xie, "An Empirical Comparison of Compiler Testing Techniques", in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 180–190, New York, NY, USA, 2016, ACM.

[Chli09] Adam Chlipala, "Syntactic Proofs of Compositional Compiler Correctness", 01 2009.

[Este18] Marvin Heinrich Esterhuizen, "Test Case Generation for Context Free Grammars", 03 2018.

[Grun12] D. Grune, K. van Reeuwijk, H.E. Bal, C.J.H. Jacobs and K. Langendoen, *Modern Compiler Design*, Springer New York, 2012.

[Koss05] A. S. Kossatchev and M. A. Posypkin, "Survey of Compiler Testing Methods", *Programming and Computer Software*, vol. 31, no. 1, pp. 10–19, 2005.

[Lern03] Sorin Lerner, Todd Milstein and Craig Chambers, "Automatically Proving the Correctness of Compiler Optimizations", in *PLDI '03 Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pp. 220–231, ACM New York, NY, USA, 2003.

[Lero08] Xavier Leroy, "Formal verification of a realistic compiler", 2008.

[Mall01] Brian A. Malloy and James F. Power, "An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases", in *1st ACIS Annual International Conference on Computer and Information Science (ICIS '01)*, October 2001.

[Moge10] Torben Ægidius Mogensen, *Basics of Compiler Design*, Department of Computer Science, University of Copenhagen, 2010.

[Myer04] Glenford J. Myers, *The Art of Software Testing, Second Edition*, John Wiley & Sons, Inc, Hoboken, New Jersey, 2004.

[Nidh12] S. Nidhra and J. Dondeti, "Black Box and White Box Testing Techniques - A Literature Review", *International Journal of Embedded Systems and Applications*, vol. 2, no. 2, 2012.

[Norv05] Theodore S. Norvell, "So You Want to Test Your Compiler?", Technical report, Memorial Univerity, Electrical and Computer Engineering, 2005.

[Rege] John Regehr, "The Future of Compiler Correctness".

[Tang19] Yixuan Tang, Zhilei Ren, Weiqiang Kong and He Jiang, "Compiler testing: a systematic literature analysis", *Frontiers of Computer Science*, Mar 2019.

[Tatl10] Zachary Tatlock and Sorin Lerner, "Bringing Extensibility to Verified Compilers", in *PLDI '10 Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 111–121, ACM New York, NY, USA, 2010.

[vKar90] B. v. Karger, "Aspects of Proving Compiler Correctness", *Christian-Albert-Universität*, 1990.

Παράρτημα Α

Τμήματα κώδικα

Έλεγχος πλήρη compiler (αρχείο compiler_tests.py) :

```
import sys
import io
import random
from os import listdir, system, access, X_OK
from os.path import isfile, join, basename

for test in correct_tests:
inputname = path_to_correct + test
expected_output = path_to_output + test[:-4] + ".out"
print( str(count) + ". Running correct test: " + test)
print ("--------------------------------------")

x = system(compiler_to_test + " " + inputname)
val = x >> 8
if (val == 1):
   print(bcolors.BOLD + "Error in compiling\n" + bcolors.ENDC)
   wrong_cases += 1
   wrong_list.append(count)
   count += 1
   continue

print("Compiling finished")
inname = path_to_input + "/" + test[:-4] + ".in"
if (isfile(inname)):
   system(path_to_correct + "a.out < " + inname + " > user_output")
else:
   system(path_to_correct + "a.out > user_output")

x = system("diff -s user_output " + expected_output)
system("rm user_output")
#print("")
val = x >> 8
if (val == 1):
   print(bcolors.BOLD + "Error, output mismatch in " + test + bcolors.ENDC)
   wrong_cases += 1
   wrong_list.append(count)
else:
   correct_cases += 1
count += 1
#we delete the a.out executable produced from the compiler
system("rm " + path_to_correct + "a.out 2> /dev/null")
print("")
```

Εκτέλεση όλων των ελέγχων

Listing A.1: doall.sh

```
#!/bin/bash

# run every step of the compiler check

# some flags that are used when we want to test a specific phase of the compiler
runlexer=false
runparser=false
runsemantic=false
runfull=false

while [ $# -gt 0 ]
do
   case "$1" in
      -l) runlexer=true;;
      -p) runparser=true;;
      -s) runsemantic=true;;
      -full) runfull=true;;
      -c) runfull=true;;
      -*) ;;
   esac
   shift
done

# path_to_here: path to the current directory
# path_to_uc: path to the directory that contains the source files of the compiler
    to-be-tested (uc: abbreviation of user's compiler)
# to run every step of the compiler separate directories need to be specified
# the script contains an example of the names those directories may have

echo "Give path to current directory "
read -r path_to_here
echo "Give path to the compiler under test "
read -r path_to_uc



if [ "$runlexer" = true ];
then
   ./run_lexer.sh $path_to_here "$path_to_uc/Lexer"
   exitl=$?
   if [ $exitl -eq 0 ];
   then
      echo "LEXER: passed"
   else
      echo "LEXER: failed"
   fi
   exit 0
fi

if [ "$runparser" = true ];
then
   ./run_lexer.sh $path_to_here "$path_to_uc/Lexer"
   exitl=$?
   if [ $exitp -eq 0 ];
   then
      echo "PARSER: passed"
   else
```

```
      echo ”PARSER: failed”
   fi
   exit 0
fi

if [ ”$runsemantic” = true ];
then
   ./run_semantic.sh $path_to_here ”$path_to_uc/Sem_analysis”
   exits=$?
   if [ $exits —eq 0 ];
   then
      echo ”SEMANTIC: passed”
   else
      echo ”SEMANTIC: failed”
   fi
   exit 0
fi

if [ ”$runfull” = true ];
then
   ./run_full_compiler.sh $path_to_here ”$path_to_uc/LLVM”
   exitf=$?
   if [ $exitf —eq 0 ];
   then
      echo ”EXECUTION: passed”
   else
      echo ”EXECUTION: failed”
   fi
   exit 0
fi

./run_lexer.sh $path_to_here ”$path_to_uc/Lexer”
exitl=$?

./run_parser.sh $path_to_here ”$path_to_uc/Parser”
exitp=$?

./run_semantic.sh $path_to_here ”$path_to_uc/Sem_analysis”
exits=$?

./run_full_compiler.sh $path_to_here ”$path_to_uc/LLVM”
exitf=$?


echo ””
echo ”RESULTS”
echo ”——————————————————————————————”
if [ $exitl —eq 0 ];
then
   echo ”LEXER: passed”
else
   echo ”LEXER: failed”
fi

if [ $exitp —eq 0 ];
then
   echo ”PARSER: passed”
else
```

```
    echo "PARSER: failed"
fi

if [ $exits -eq 0 ];
then
    echo "SEMANTIC: passed"
else
    echo "SEMANTIC: failed"
fi

if [ $exitf -eq 0 ];
then
    echo "EXECUTION: passed"
else
    echo "EXECUTION: failed"
fi

echo "-----------------------------------"

exit 0
```