ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αναφορά Εξαμηνιαίας Εργασίας

στο μάθημα 8ου εξαμήνου (ροής Λ)

Μεταγλωττιστές

Θέμα εργασίας: “*Ανάπτυξη ενός μεταγλωττιστή για τη γλώσσα προγραμματισμού Alan*”

Στοιχεία ομάδας

Ονοματεπώνυμο:	ΜΑΘΙΟΥΛΑΚΗ ΕΛΕΝΗ
Α.Μ.:	03114040
E-mail:	el.mathioulaki@gmail.com

Ονοματεπώνυμο:	ΝΙΑΡΧΟΣ ΣΩΤΗΡΙΟΣ
Α.Μ.:	03114076
E-mail:	sot.niarchos@gmail.com

Εισαγωγή

Αντικείμενο της εργασίας μας είναι η ανάπτυξη ενός μεταγλωττιστή για τη γλώσσα προγραμματισμού Alan, μία απλή προστακτική γλώσσα προγραμματισμού που περιγράφεται αναλυτικά στην εκφώνηση της εργασίας.

Η παρούσα αναφορά χωρίζεται σε αρκετά μέρη, ένα για κάθε τμήμα του μεταγλωττιστή. Συγκεκριμένα, η παρουσίαση θα γίνει ως εξής:

1. Εποπτεία της συνολικής αρχιτεκτονικής του μεταγλωττιστή
2. Λεκτικός αναλυτής (lexer)
3. Συντακτικός αναλυτής (parser) και κατασκευή αφαιρετικού συντακτικού δέντρου (AST)
4. Σημασιολογικός αναλυτής
5. Παραγωγή ενδιάμεσου κώδικα και βελτιστοποίηση
6. Παραγωγή τελικού κώδικα και βελτιστοποίηση
7. alanc – ο τελικός μεταγλωττιστής

Οι τεχνολογίες που χρησιμοποιήθηκαν σε κάθε τμήμα του μεταγλωττιστή είναι οι εξής:

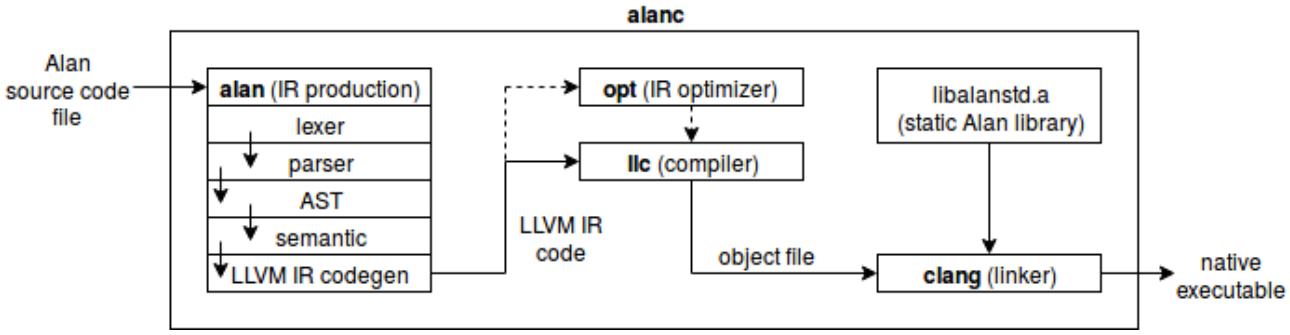
- Για τον λεκτικό αναλυτή χρησιμοποιήθηκε το εργαλείο flex (έκδοση 2.6.0)
- Για τον συντακτικό αναλυτή χρησιμοποιήθηκε το εργαλείο bison (έκδοση 3.0.4)
- Ο σημασιολογικός αναλυτής και η κατασκευή του AST γράφτηκαν σε C++11
- Η παραγωγή ενδιάμεσου κώδικα έγινε σε C++11 με χρήση των αντίστοιχων LLVM bindings (ο μεταγλωττιστής ελέγχθηκε με τις εκδόσεις 3.8 και 6.0 του LLVM, ανάλογα την έκδοση μπορεί να δημιουργηθούν κάποια warnings κατά το make, τα οποία μπορούν με ασφάλεια να αγνοηθούν)
- Η βελτιστοποίηση του ενδιάμεσου κώδικα, καθώς και η παραγωγή του τελικού κώδικα και η βελτιστοποίησή του έγιναν με χρήση κάποιων command line utilities που προσφέρονται από το LLVM (αναλυτικότερη αναφορά σε αυτά θα γίνει στη συνέχεια)
- Η standard βιβλιοθήκη της Alan γράφτηκε από την αρχή σε C, ακολουθώντας όσο πιο πιστά γινόταν αφενός την περιγραφή των συναρτήσεων στην εκφώνηση της εργασίας, αφετέρου τη συμπεριφορά των αντίστοιχων συναρτήσεων της C. Αυτό επιλέξαμε να το κάνουμε στα πλαίσια της προσπάθειας για μία πιο cross-platform προσέγγιση
- Το τελικό script που αλληλεπιδρά με τον χρήστη και συνενώνει όλα τα παραπάνω γράφτηκε σε Python 3.7. Η Python προτιμήθηκε ως “γλώσσα συνένωσης” (glue language) έναντι της αρχικής μας υλοποίησης σε bash με στόχο το τελικό script να είναι όσο το δυνατόν πιο εύχρηστο, ευανάγνωστο, επεκτάσιμο και cross-platform

Για το build του project αρκεί η εκτέλεση της εντολής make στο root directory (αυτό αποτελεί και το μόνο σημείο της εργασίας μας που, δυστυχώς, δεν συμμορφώνεται με την ιδέα ενός πλήρως cross-platform project).

1. Εποπτεία της συνολικής αρχιτεκτονικής του μεταγλωττιστή

Αρχικά, θα αναφερθούμε στην συνολική αρχιτεκτονική του μεταγλωττιστή, ώστε να είναι ξεκάθαρα η θέση και ο ρόλος του κάθε τμήματος το οποίο θα αναλύουμε στη συνέχεια της αναφοράς.

Μία αφαιρετική οπτική της αρχιτεκτονικής του μεταγλωττιστή που υλοποιήσαμε φαίνεται στο διάγραμμα που ακολουθεί. Πριν προχωρήσουμε στην επιμέρους ανάλυση των τμημάτων του μεταγλωττιστή, θα αναφερθούμε συνοπτικά στον τρόπο που αυτά αλληλεπιδρούν και ποιες είναι οι αρμοδιότητες του κάθε τμήματος, όπως αυτές προκύπτουν από τον σχεδιασμό μας.



Η καρδιά του μεταγλωττιστή μας είναι το πρόγραμμα **alan**. Αυτό είναι υπεύθυνο για την ανάγνωση του πηγαίου κώδικα Alan, την επεξεργασία του και, εν τέλει, την παραγωγή του αντίστοιχου ενδιάμεσου κώδικα, σε LLVM IR. Για να το επιτύχει αυτό, αποτελείται μεταξύ άλλων από τον λεκτικό, τον συντακτικό και τον σημασιολογικό αναλυτή. Ο λεκτικός αναλυτής αναγνωρίζει και προωθεί στο pipeline του alan τα lexemes, τα οποία στη συνέχεια ο συντακτικός αναλυτής χρησιμοποιεί για να κατασκευάσει το abstract syntax tree (AST) που αντιπροσωπεύει το αρχικό πρόγραμμα. Αυτό επιτυγχάνεται, προφανώς, στην περίπτωση που δεν εντοπιστούν ούτε λεκτικά ούτε συντακτικά λάθη. Στη συνέχεια, ο σημασιολογικός αναλυτής ελέγχει ολόκληρο το AST για σημασιολογικά λάθη. Εάν τέτοια δεν εντοπιστούν, ένα ακόμα τελευταίο πέρασμα του AST λαμβάνει χώρα, κατά το οποίο παράγεται ενδιάμεσος κώδικας στη γλώσσα του LLVM (LLVM IR), με χρήση των κατάλληλων C++ bindings.

Σε αυτό το σημείο, έχει χαθεί οτιδήποτε σχετίζεται με την Alan: έχουμε αυτή τη στιγμή στα χέρια μας το ισοδύναμο του αρχικού προγράμματος σε LLVM IR, το οποίο σημαίνει πως τώρα μπορούμε να το αντιμετωπίσουμε ως τέτοιο, απολύτως ανεξάρτητα από την αρχική (πηγαία) γλώσσα. Συνεπώς, η πορεία από εδώ και πέρα καθορίζεται σχεδόν αποκλειστικά από το LLVM.

Πιο συγκεκριμένα, εάν ο χρήστης έχει ζητήσει να γίνουν βελτιστοποιήσεις, ο ενδιάμεσος κώδικας περνάει από το **opt**, ένα command line utility του LLVM το οποίο δέχεται ως είσοδο LLVM IR κώδικα και παράγει (πιθανώς) βελτιστοποιημένο LLVM IR κώδικα. Δεν θελήσαμε να κάνουμε υποχρεωτικό το στάδιο της βελτιστοποίησης του ενδιάμεσου κώδικα, ώστε να μπορούμε να πειραματιστούμε περισσότερο με τις διαφορές που προκύπτουν (με ή χωρίς βελτιστοποίηση) στη συνέχεια του pipeline, αλλά και για να μπορεί ο χρήστης να δει τον ενδιάμεσο κώδικα ακριβώς όπως τον παράγουμε εμείς μέσω των C++ bindings, χωρίς καμία αλλαγή (με χρήση του flag -i, όπως περιγράφεται στην εκφώνηση της εργασίας).

Στη συνέχεια, το **llc**, ο compiler του LLVM, αναλαμβάνει να μετατρέψει τον ενδιάμεσο κώδικα σε object file, να το μεταγλωττίσει, δηλαδή, στη native assembly.

Τέλος, ο **clang** αναλαμβάνει να συνδέσει (link) το object file που έχει παραχθεί με τη standard βιβλιοθήκη της Alan (libalanstd.a), παράγοντας έτσι το τελικό native εκτελέσιμο πρόγραμμα.

2. Λεκτικός αναλυτής (lexer)

Ο λεκτικός αναλυτής (lexer) είναι το πρώτο τμήμα του μεταγλωττιστή μας. Η υλοποίησή του έγινε με τη βοήθεια του εργαλείου **flex**. Ο lexer είναι υπεύθυνος για την ανάγνωση του source code του προγράμματος εισόδου και της παραγωγής των κατάλληλων lexemes, τα οποία και στη συνέχεια προωθούνται στον συντακτικό αναλυτή (parser). Ο lexer υλοποιήθηκε αποκλειστικά στο αρχείο **src/lexer.l**.

Η υλοποίηση του lexer αποτελεί μία αρκετά standard διαδικασία και δεν έχουμε να επισημάνουμε κάποια ιδιαίτερη σχεδιαστική επιλογή.

Ακολουθώντας την εκφώνηση, γράψαμε τα regexes που ορίζουν τις μεταβλητές, τις σταθερές (ακέραιες και δεκαεξαδικές), τους χαρακτήρες διαφυγής, και ούτω καθεξής.

Για τα keywords της Alan χρησιμοποιήσαμε τον parser (βλ. επόμενο μέρος) όπου και τα ορίσαμε και στη συνέχεια τα χρησιμοποιούμε από τον lexer μέσω του header parser.hpp που παράγεται κατά την εκτέλεση του bison.

Για κάθε κανόνα που αντιστοιχεί σε κάποιο lexeme, αυτό προωθείται στον parser μέσω του struct **yylval**, για το οποίο θα μιλήσουμε στο επόμενο μέρος.

Επιπλέον, για τα σχόλια (και κυρίως επειδή αυτά επιτρεπόταν να είναι εμφωλευμένα), χρησιμοποιήσαμε τον μηχανισμό των καταστάσεων (states) που προσφέρει το flex, δημιουργώντας μία ξεχωριστή κατάσταση για την περίπτωση των σχολίων (<MULLINE_COMMENT>). Από αυτήν την κατάσταση φεύγουμε μόνο στην περίπτωση που συναντήσουμε τόσες αρχές σχολίων (“(*”) όσα και τέλη (“*)”). Ενδιαμέσως, δεν αναγνωρίζουμε κανένα άλλο lexeme.

Τέλος, ο lexer χειρίζεται τον μετρητή linecount προσαυξάνοντάς τον κάθε φορά που συναντάει χαρακτήρα αλλαγής γραμμής (“\n”), ο οποίος έχει οριστεί ως εξωτερική (extern) μεταβλητή στον parser.

3. Συντακτικός αναλυτής (parser) και κατασκευή abstract syntax tree (AST)

Ο συντακτικός αναλυτής (parser) αποτελεί την καρδιά του προγράμματος **alan** (βλ. διάγραμμα 1ου μέρους), αφού είναι αυτός στον οποίο ορίζεται η συνάρτηση main και ενορχηστρώνει τόσο την ανάγνωση του προγράμματος εισόδου (χρησιμοποιώντας τον lexer), όσο και την κατασκευή του AST, τον σημασιολογικό έλεγχο αλλά και την παραγωγή του ενδιάμεσου κώδικα σε LLVM IR. Ο parser υλοποιήθηκε αποκλειστικά στο αρχείο **src/parser.ypp**, με χρήση του εργαλείου **bison**.

3.1 Η υλοποίηση του συντακτικού αναλυτή

Αρχικά, θα αναφερθούμε στην υλοποίηση του parser. Όπως και στον lexer, η διαδικασία κατασκευής του parser είναι σε μεγάλο βαθμό standard. Το κύριο μέλημα εδώ είναι η μετάφραση της γραμματικής της Alan σε κατάλληλους κανόνες που θα οδηγήσουν στην κατασκευή ενός σαφούς και εύχρηστου AST (θα αναφερθούμε εκτενώς σε αυτό στη συνέχεια).

Πρώτα απ' όλα, στον parser ορίζουμε ποια lexemes αντιστοιχούν σε keywords της Alan, πληροφορία που γίνεται exported στον lexer μέσω του header parser.hpp, που δημιουργείται αυτόματα.

Στη συνέχεια, ορίζουμε το struct **yylval**, το οποίο λειτουργεί ως ένας "buffer" μεταξύ lexer και parser, όπου ο πρώτος γράφει τα lexemes που βρίσκει στη κατάλληλη θέση ώστε να τα καταναλώσει ο δεύτερος. Εξαίρεση αποτελούν τα keywords, για τα οποία ο lexer επιστρέφει στον parser απλά τον αντίστοιχο αριθμό, όπως ορίστηκε αυτόματα στο parser.hpp. Το yylval δεν είναι παρά ένα union, στο οποίο ορίζονται όλοι οι δυνατοί τύποι δεδομένων για τα αντικείμενα που είτε έρχονται από τον lexer, είτε δημιουργούνται από τον parser:

- κόμβοι του AST (ASTNodes, δημιουργούνται από τον parser)
- χαρακτήρες (από τον lexer)
- strings (από τον lexer)
- ακέραιοι (από τον lexer)
- τύποι (Types, ορισμένοι στο δοσμένο πίνακα συμβόλων, δημιουργούνται από τον parser)

Έπειτα, ορίζεται η προτεραιότητα και η προσεταιριστικότητα των πράξεων/τελεστών με βάση την εκφώνηση της άσκησης, καθώς και οι τύποι (με βάση το yylval) των δεδομένων που δημιουργούνται όταν κάποιος από τους κανόνες του parser ενεργοποιείται.

Σε αυτό το σημείο, ορίζονται οι κανόνες της γραμματικής της Alan. Θεωρούμε ότι δεν υπάρχει λόγος να μπούμε σε εκτενείς λεπτομέρειες, καθώς ακολουθήσαμε πιστά την γραμματική όπως αυτή παρουσιάζεται στην εκφώνηση. Για κάθε κανόνα που ενεργοποιείται, δημιουργείται μία αντίστοιχη δομή (είτε ένα καινούργιο ASTNode, είτε ένα καινούργιο Type, ανάλογα με τον κανόνα), και συνδέεται με την δομή – γονιό, με τη βοήθεια των γνωστών τελεστών $$ και $1, $2, ... του bison. Αφού αναλυθεί συντακτικά το σύνολο του προγράμματος εισόδου, έχουμε στα χέρια μας το συνολικό AST που το αναπαριστά.

3.2 Το αφαιρετικό συντακτικό δέντρο (abstract syntax tree – AST)

Σε αυτό το σημείο θα μιλήσουμε πιο αναλυτικά για τον τρόπο που υλοποιήσαμε το AST που θα αναπαριστά το εκάστοτε πρόγραμμα εισόδου, μετά την επιτυχή συντακτική ανάλυσή του. Το AST υλοποιήθηκε στο αρχείο **include/ast.hpp**, σε C++11.

Στο ast.hpp ορίζεται ο κόμβος (ASTNode) του AST, ως μία κλάση. Τα private πεδία της κλάσης είναι όλες οι πληροφορίες που μπορεί να χρειάζεται να συγκρατήσει *οποιοδήποτε είδος κόμβου του AST*. Με άλλα λόγια, έχουμε ένα και μόνο είδος ASTNode κλάσης. Για παραπάνω πληροφορίες για το κάθε πεδίο της ASTNode παραπέμπουμε στο αντίστοιχο αρχείο, όπου περιγράφεται αναλυτικά σε σχόλια ό,τι χρειάζεται να γνωρίζει κανείς για την κλάση.

Η διαφοροποίηση μεταξύ ASTNodes που επιτελούν άλλη λειτουργία (για παράδειγμα, μεταξύ ενός κόμβου που αναπαριστά δήλωση συνάρτησης και ενός που αναπαριστά μία άθροιση) επιτυγχάνεται με τη χρήση διαφορετικών **constructors**, οι οποίοι ορίζονται ως protected μέθοδοι της κλάσης ASTNode. Στη συνέχεια, ορίζονται πολλές υποκλάσεις της ASTNode, τόσες όσα και τα διαφορετικά είδη κόμβων που χρειαζόμαστε για το AST. Κάθε μία από αυτές τις υποκλάσεις αποτελείται από τις εξής τρεις μεθόδους, και μόνον αυτές (μαζί, προφανώς, με τα private πεδία που κληρονόμησαν από την ASTNode):

- Έναν constructor, ο οποίος καλεί τον αντίστοιχο constructor της γονεϊκής κλάσης ASTNode, που σημαίνει πως γεμίζει μόνο τα private πεδία που αυτός ο συγκεκριμένος κόμβος χρειάζεται,
- την void sem(), η οποία χρησιμοποιείται κατά τον σημασιολογικό έλεγχο (βλ. επόμενο μέρος) και ορίζεται εκ νέου από κάθε κλάση – παιδί (είναι virtual μέθοδος της ASTNode), και
- την llvm::Value * codegen(), η οποία χρησιμοποιείται κατά την παραγωγή ενδιάμεσου κώδικα σε LLVM IR (βλ. Μέρος 5) και ορίζεται εκ νέου από κάθε κλάση – παιδί (είναι virtual μέθοδος της ASTNode).

Τα παραπάνω ορίζουν ένα πολύ απλό interface του AST για τον parser, το οποίο και χρησιμοποιεί εάν χρειαστεί να δημιουργηθεί ένας καινούργιος κόμβος του AST ως αποτέλεσμα ενεργοποίησης κάποιου κανόνα. Για παράδειγμα, εάν ο parser συναντήσει την εντολή “return 42;”, δεν δημιουργεί ένα αντικείμενο ASTNode, αλλά ένα αντικείμενο ASTRet, δίνοντας στον κατασκευαστή του ως όρισμα μόνο την έκφραση “42” (η οποία θα έχει γίνει και αυτή αναδρομικά ένας κόμβος του AST τύπου ASTInt). Έπειτα, ο κατασκευαστής της κλάσης ASTRet θα καλέσει τον κατασκευαστή της ASTNode που του αντιστοιχεί, δηλαδή εκείνον που δέχεται ως όρισμα μόνο μία έκφραση (και αφήνει τα υπόλοιπα πεδία κενά).

3.3 Η λειτουργία του συντακτικού αναλυτή (η συνάρτηση main)

Όλα τα παραπάνω εκκινούν από την main συνάρτηση, η οποία επίσης ορίζεται στο parser.ypp και αποτελεί τη main του προγράμματος alan γενικότερα.

Συγκεκριμένα, πρώτα αρχικοποιείται το όνομα του αρχείου εισόδου (filename) και ο μετρητής γραμμής (linecount). Έπειτα καλείται η **yyparse()**, η οποία κατασκευάζει το συνολικό AST και αποθηκεύει τον ριζικό κόμβο σε μία μεταβλητή t. Εάν η yyparse() αποτύχει, το πρόγραμμα τερματίζει με κατάλληλο μήνυμα και κωδικό εξόδου (1). Σε αντίθετη περίπτωση, δημιουργείται ο πίνακας συμβόλων (βλ. επόμενο μέρος) στον οποίο προστίθενται οι συναρτήσεις βιβλιοθήκης της Alan με μία δική μας συνάρτηση (initLibFunctions()), ώστε η ύπαρξη και το signature τους να είναι γνωστά κατά τον σημασιολογικό έλεγχο του AST. Σε αυτό το σημείο, καλείται η συνάρτηση sem() του ριζικού κόμβου του AST (μεταβλητή t), η οποία εκκινεί τον αναδρομικό σημασιολογικό έλεγχο του AST. Εάν αυτός αποτύχει σε κάποιον κόμβο, το πρόγραμμα *δεν τερματίζει ακαριαία*, με στόχο να βρεθούν όσο περισσότερα λάθη γίνεται. Αυτό επιλέξαμε να το κάνουμε μιμούμενοι τη συμπεριφορά άλλων compilers, όπως του gcc/g++. Παρόλα αυτά, το γεγονός πως υπήρξε λάθος καταχωρείται σε μία μεταβλητή και, όταν ο σημασιολογικός έλεγχος διατρέξει όλο το AST, ο πίνακας συμβόλων καταστρέφεται και η μεταβλητή αυτή ελέγχεται και, εάν έχει όντως προκύψει σημασιολογικό σφάλμα, το πρόγραμμα τερματίζει επιστρέφοντας κωδικό εξόδου 1. Σε αντίθετη περίπτωση, δεν μπορούν να υπάρξουν λάθη από εδώ και πέρα – έχουμε στα χέρια μας ένα ορθό, τόσο συντακτικά όσο και σημασιολογικά, AST. Έτσι, το τελευταίο βήμα είναι η κλήση της συνάρτησης codegen(), η οποία διατρέχει και πάλι όλο το AST και αναδρομικά παράγει τον αντίστοιχο LLVM IR κώδικα, τον οποίο και τυπώνει απευθείας στο stdin όταν ολοκληρωθεί. Αυτός, έπειτα, συλλέγεται από τα επόμενα τμήματα του συνολικού μεταγλωττιστή προς τη πιθανή βελτιστοποίηση και, εν τέλει, την παραγωγή τελικού κώδικα και ενός native εκτελέσιμου.

4. Σημασιολογικός αναλυτής

Σε αυτό το μέρος της αναφοράς θα αναφερθούμε σύντομα στον σημασιολογικό αναλυτή του μεταγλωττιστή μας. Ο σημασιολογικός αναλυτής υλοποιείται στο αρχείο **include/ast.cpp**, σε C++11, και επί της ουσίας δεν είναι τίποτα περισσότερο από τον ορισμό της συνάρτησης void sem(), για κάθε κλάση – παιδί της ASTNode.

Η διαδικασία είναι αναδρομική και ξεκινά από την κλήση της sem() στον ριζικό κόμβο του AST, από την main του parser. Σε κάθε κόμβο που φτάνει ο σημασιολογικός έλεγχος, ελέγχεται πλήθος συνθηκών που σχετίζεται με τις ιδιότητες και τη λειτουργία του συγκεκριμένου κόμβου, συναρτήσει του τι αυτός αντιπροσωπεύει στο αρχικό πρόγραμμα.

Λόγω του πλήθους συνθηκών που ελέγχονται σε κάθε κόμβο, αλλά και των μεγάλων διαφορών μεταξύ των κόμβων, κρίναμε πως δεν χρειάζεται να τα παραθέσουμε στην αναφορά. Παραπέμπουμε στον πηγαίο κώδικα, στον οποίο μπορεί κανείς να αποκτήσει μία πλήρη εικόνα των ελέγχων που λαμβάνουν χώρα, τόσο από τα σχόλια όσο και από τα αντίστοιχα μηνύματα λάθους που δίνονται ως ορίσματα στη συνάρτηση error().

Αξίζει παρόλα αυτά να αναφέρουμε δύο σχεδιαστικές μας επιλογές:

- Για τον πίνακα συμβόλων χρησιμοποιήθηκε η έτοιμη υλοποίηση όπως μας δόθηκε από τον διδάσκοντα (αρχεία **include/symbol.hpp** και **src/symbol.cpp**). Χρειάστηκε να γίνουν κάποιες μικρές αλλαγές ώστε ο κώδικας να μετατραπεί από C σε C++ (για παράδειγμα, τα enumerations δεν μπορούσαν πλέον παρά να ορίζονται globally).
- Χρησιμοποιήσαμε ένα stack από SymbolEntry* αντικείμενα για να μοντελοποιήσουμε τις συναρτήσεις του προγράμματος, ώστε να γνωρίζουμε ανά πάσα στιγμή μέσα σε ποια συνάρτηση βρισκόμαστε κατά τον σημασιολογικό έλεγχο, κάτι πολύ χρήσιμο, για παράδειγμα, στους ορισμούς επικεφαλίδων συναρτήσεων (για τη χρήση της endFunctionHeader() του πίνακα συμβόλων) ή για τον έλεγχο κατά την εντολή “return”.

5. Παραγωγή ενδιάμεσου κώδικα και βελτιστοποίηση

Αρχικά, θα

6. Παραγωγή τελικού κώδικα και βελτιστοποίηση

Βρισκόμαστε, πλέον, στο σημείο όπου ο ενδιάμεσος κώδικας έχει παραχθεί (και ενδεχομένως βελτιστοποιηθεί). Για την παραγωγή του τελικού κώδικα, δεδομένου του ότι ο ενδιάμεσος κώδικας είναι σε LLVM IR, αρκεί να χρησιμοποιήσουμε το command line utility **llc**, τον compiler του LLVM. Με την εντολή “llc -filetype=obj” και είσοδο τον ενδιάμεσο κώδικα από το προηγούμενο τμήμα του μεταγλωττιστή (συγκεκριμένα, το stdout του) παράγουμε ένα object file σε native assembly. Εάν ζητήθηκε βελτιστοποίηση, τότε προσθέτουμε και το flag -O3 στην παραπάνω εντολή.

Για να έχουμε στα χέρια μας ένα τελικό εκτελέσιμο, απαιτείται ένα ακόμα βήμα – η σύνδεση (linking) του παραπάνω object file με τη βιβλιοθήκη της Alan, ώστε να γίνουν resolve τα ονόματα συναρτήσεων που έχουν δηλωθεί μέσα στο object file, αλλά που το τελευταίο δεν γνωρίζει πού βρίσκονται οι υλοποιήσεις τους. Αυτό επιτυγχάνεται με την εντολή “clang <object file> libalanstd.a -o <outname>”, οπότε και παράγεται απευθείας το τελικό εκτελέσιμο πρόγραμμα.

7. alanc – ο τελικός μεταγλωττιστής

Σε αυτό το μέρος της παρούσας αναφοράς θα περιγράψουμε τον τρόπο χρήσης και λειτουργίας του τελικού εκτελέσιμου του μεταγλωττιστή, του **alanc**. Στο πρώτο μέρος παρουσιάσαμε συνοπτικά την γενική αρχιτεκτονική του script, ενώ σε αυτό το μέρος θα αναφερθούμε πιο αναλυτικά στην υλοποίησή του καθώς και στις σχεδιαστικές επιλογές που αποφασίσαμε να κάνουμε.

Αρχικά, παρατίθεται το μήνυμα που λαμβάνει ο χρήστης με την εκτέλεση της εντολής **./alanc -h**:

```
usage: alanc [-h] [-O] [-a] [-i] [-f] [-c] [-o OUTNAME] [infile]

ALANC - the Alan Limitless and Amazingly Neat Compiler

positional arguments:
  infile      (if -f or -i not given) the Alan source code to compile

optional arguments:
  -h, --help  show this help message and exit
  -O          optimize IR and final (assembly) code
  -a, --all   emit IR and assembly code in two seperate files (switched off by
              default)
  -i          read source code from stdin, print IR code to stdout, then exit
              (no executable is produced)
  -f          read source code from stdin, print final code to stdout, then
              exit (no executable is produced)
  -c          create object file and stop, skipping the linking phase
  -o OUTNAME  (if -f or -i not given) the name of the produced executable
              (default: a.out)
```

Όπως φαίνεται παραπάνω, υπάρχει μία σειρά από flags που προσφέρονται στον χρήση με σκοπό την παραμετροποίηση της λειτουργίας του μεταγλωττιστή.

Το flag **-O** ενεργοποιεί την βελτιστοποίηση τόσο του ενδιάμεσου (LLVM IR) όσο και του τελικού κώδικα (native assembly). Εξηγήσαμε στο πρώτο μέρος της αναφοράς την απόφασή μας να μην είναι υποχρεωτική η βελτιστοποίηση του ενδιάμεσου κώδικα.

Με χρήση του flag **-a (--all)** δημιουργούνται δύο επιπλέον αρχεία, ένα που περιέχει τον ενδιάμεσο και ένα τον τελικό κώδικα που παράγεται από τον μεταγλωττιστή, σε human-readable μορφή, κατά τις προδιαγραφές της εκφώνησης. Αποφασίσαμε να μην ορίσουμε ως default συμπεριφορά του μεταγλωττιστή την παραγωγή αυτών των δύο αρχείων (όπως φαίνεται, βέβαια, να υπονοείται από την εκφώνηση της εργασίας) ώστε να αποφύγουμε να δημιουργούνται δύο επιπλέον αρχεία κάθε φορά που ο χρήστης θέλει να μεταγλωττίσει ένα πρόγραμμα, τη στιγμή που απλά περιμένει να δημιουργηθεί ένα εκτελέσιμο. Με άλλα λόγια, δώσαμε σε αυτό το flag (και σε αυτήν την λειτουργικότητα του μεταγλωττιστή) μία διάσταση debugging.

Τα flags **-i** και **-f** ακολουθούν τις προδιαγραφές της εκφώνησης, όπως φαίνεται και στις περιγραφές τους στο μήνυμα παραπάνω. Αξίζει να σημειωθεί πως αποφασίσαμε να κάνουμε τη χρήση τους mutually exclusive, προς αποφυγή συγχύσεων. Επίσης, όπως φαίνεται και στα παραπάνω μηνύματα, αποφασίσαμε να μην παράγεται το τελικό εκτελέσιμο σε αυτές τις περιπτώσεις, δίνοντας την δυνατότητα στον μεταγλωττιστή μας να λειτουργήσει και ως μέρος κάποιου toolchain που θέλει να κατασκευάσει ο χρήστης, όπου και δεν περιμένει ή επιθυμεί τέτοιου είδους side effects.

Το flag **-c** έχει ακριβώς την ίδια σημασία με το αντίστοιχο flag του gcc: σταματάει στη φάση της μεταγλώττισης του πηγαίου προγράμματος, χωρίς να προχωρήσει στο στάδιο της σύνδεσης (linking) με τη standard βιβλιοθήκη της Alan και, τελικά, την παραγωγή του εκτελέσιμου προγράμματος. Τελικό προϊόν σε περίπτωση που δοθεί αυτό το flag είναι το αντίστοιχο object file, με κατάληξη “.o”. Σημειώνεται πως αυτό το αρχείο πάντα παράγεται, απλά σε περίπτωση που δεν

έχει δοθεί το εν λόγω flag, διαγράφεται μετά την παραγωγή του εκτελέσιμου προγράμματος.

Επιπλέον, δίνεται στο χρήστη η δυνατότητα να δώσει κάποιο όνομα που επιθυμεί στο τελικό εκτελέσιμο με τη χρήση του flag **-o** (όπως και στον gcc, το default όνομα για το παραγόμενο εκτελέσιμο πρόγραμμα είναι a.out).

Τέλος, ο χρήστης μπορεί να δώσει το όνομα ενός αρχείου προς μεταγλώττιση, ακριβώς όπως στον gcc και στον clang. Προφανώς, εάν δώσει κάποιο όνομα αρχείου, δεν μπορεί να έχει δώσει και κάποιο από τα flags **-i** ή **-f**.

Τα παραπάνω flags ορίστηκαν και υλοποιήθηκαν με βάση το module **argparse** της Python. Όταν το πρόγραμμα ξεκινά, γίνονται κάποιοι έλεγχοι σχετικοί με τα flags:

1. Ο χρήστης επιτρέπεται είτε να χρησιμοποιήσει ένα εκ των flags **-i** ή **-f** είτε να δώσει ένα όνομα αρχείου προς μεταγλώττιση.
2. Ένα από τα δύο παραπάνω πρέπει να έχουν δοθεί (δηλαδή, δεν επιτρέπεται να λείπουν και τα δύο)
3. Μόνο ένα εκ των flags **-i** ή **-f** επιτρέπεται να χρησιμοποιηθεί
4. Ο χρήστης επιτρέπεται είτε να χρησιμοποιήσει ένα εκ των flags **-i** ή **-f** είτε το flag **-a** (στα πλαίσια της σχεδιαστικής μας επιλογής να μην επιτρέπουμε side effects στην περίπτωση χρήσης των flags **-i** ή **-f**)
5. Δεδομένου του ότι η χρήση των flags **-i** ή **-f** συνεπάγεται μη παραγωγή τελικού εκτελέσιμου προγράμματος, δεν επιτρέπεται να χρησιμοποιούνται μαζί με το flag **-o**.

Σε περίπτωση που οποιαδήποτε από τις παραπάνω συνθήκες δεν πληρούται, η εκτέλεση του μεταγλωττιστή τερματίζεται με αντίστοιχο μήνυμα λάθους από το argparse.

Στη συνέχεια, ορίζεται το όνομα του προγράμματος (αποκόπτεται η κατάληξη εάν αυτή είναι “.alan”, αλλιώς μένει ως έχει), του object file που θα παραχθεί, καθώς και των αρχείων .imm και .asm κατά τις προδιαγραφές της εκφώνησης, σε περίπτωση που αυτά ζητήθηκαν από τον χρήστη.

Έπειτα, ορίζονται οι εντολές που εκτελούνται για τα 4 βήματα της μεταγλώττισης που ακολουθούν μαζί με τα αντίστοιχα flags τους. Τα 4 αυτά βήματα φαίνονται εποπτικά στο διάγραμμα του πρώτου μέρους της παρούσας αναφοράς:

1. Για την **παραγωγή του ενδιάμεσου κώδικα** χρησιμοποιείται το πρόγραμμα **alan** με μοναδικό όρισμα το όνομα του προγράμματος ώστε να χρησιμοποιηθεί εσωτερικά από το LLVM αλλά και για πιθανά μηνύματα λάθους. Δέχεται το πηγαίο κώδικα από το stdin και παράγει τον ενδιάμεσο κώδικα στο stdout.
2. Για την **βελτιστοποίηση του ενδιάμεσου κώδικα** χρησιμοποιείται το command line utility **opt** του LLVM, με το flag **-O3**, το οποίο καλύπτει τα ζητούμενα της πρώτης bonus μονάδας της εργασίας. Δέχεται τον ενδιάμεσο κώδικα από το stdin και παράγει (πιθανώς) βελτιστοποιημένο ενδιάμεσο κώδικα στο stdout. Αυτό το βήμα παραλείπεται σε περίπτωση που δεν έχει δοθεί από τον χρήστη το αντίστοιχο flag. Επιπλέον, μετά από αυτό το βήμα (είτε έχει εκτελεστεί είτε όχι) και εάν αυτό έχει ζητηθεί από τον χρήστη, τότε ο ενδιάμεσος κώδικας είτε αποθηκεύεται σε κατάλληλο .imm αρχείο και η εκτέλεση του μεταγλωττιστή συνεχίζεται είτε τυπώνεται στην οθόνη και το πρόγραμμα τερματίζει επιτυχώς.
3. Για την **παραγωγή του τελικού κώδικα** χρησιμοποιείται το command line utility **llc**, ο μεταγλωττιστής του LLVM, μαζί με το flag **-filetype=obj**. Αποφασίσαμε σε αυτό το στάδιο να παράγεται object file και όχι κάποια άλλη μορφή assembly ώστε να μπορούμε να υποστηρίξουμε ευκολότερα τη λειτουργία του flag **-c**, να είναι ευκολότερο το linking στη συνέχεια, αλλά και γιατί αποτελεί standard συμπεριφορά πολλών μεταγλωττιστών η παραγωγή ενός object file σε αυτή τη φάση, κάτι που μπορεί ο χρήστης να επιθυμεί. Σε περίπτωση που δόθηκε το flag βελτιστοποίησης, προστίθεται στα flags του llc το **-O3**, το οποίο καλύπτει τα ζητούμενα της δεύτερης bonus μονάδας της εργασίας. Η εντολή αυτή δέχεται τον (πιθανώς βελτιστοποιημένο) ενδιάμεσο κώδικα από το stdin και παράγει ένα

object file το οποίο αποθηκεύεται στο file system, στο τρέχον directory. Επιπλέον, μετά από αυτό το βήμα και εάν αυτό έχει ζητηθεί από τον χρήστη, τότε ο τελικός κώδικας είτε αποθηκεύεται σε κατάλληλο .asm αρχείο και η εκτέλεση του μεταγλωττιστή συνεχίζεται είτε τυπώνεται στην οθόνη, διαγράφεται το object file και το πρόγραμμα τερματίζει επιτυχώς. Για να συμβούν αυτά τα δύο εφόσον ζητηθούν, ξανακαλούμε τον llc, αλλά αυτή τη φορά χωρίς το flag **-filetype=obj**, ώστε να παραχθεί human-readable assembly αντί για ένα object file.

4. Το τελευταίο βήμα είναι **η σύνδεση του object file με τη standard βιβλιοθήκη της Alan (linking)**. Για αυτό το βήμα, εκτελείται απλώς η εντολή:
   `clang <objectfile> libalanstd.a –o <outname>`
   Μετά από αυτήν την εντολή, έχει παραχθεί το τελικό εκτελέσιμο πρόγραμμα με όνομα outname. Εάν έχουμε φτάσει σε αυτό το σημείο, διαγράφεται και το object file από το file system.

Σημειώνεται πως σε κάθε ένα από τα παραπάνω βήματα ελέγχεται στο stderr κάθε εντολής. Σε περίπτωση που δεν είναι κενό, το μήνυμα προωθείται στον χρήστη και ο μεταγλωττιστής τερματίζει, ανεπιτυχώς (με αρνητικό κωδικό εξόδου).

Για την εκτέλεση όλων των παραπάνω εντολών χρησιμοποιήθηκε το module **subprocess** ενώ για τον χειρισμό των directories, των paths και των αρχείων χρησιμοποιήθηκε το module **os**, και πάλι σε μία προσπάθεια για cross-platform υλοποίηση.