COMPILERS
2017

Βασίλειος Δρέττας – 1115201300042
Δημήτριος Κωνσταντάκης – 1115201300079

Η εργασία έχει υλοποιηθεί σε Java.
Υλοποιήθηκαν όλα τα ερωτήματα και το bonus ερώτημα για optimization. Αναλυτικές πληροφορίες παρακάτω.

Parser.grammar
Το αρχείο αυτό χρησιμοποιείται για την λεκτική και συντακτική ανάλυση ενός κώδικα σε γλώσσα Grace. Πιο αναλυτικά έχουν υλοποιηθεί τα :

Helpers : Για τα comment έχουν υλοποιηθεί δύο είδη, το παραδοσιακό (traditional) το οποίο μοιάζει με το /* */ , με τη διαφορά ότι θέλει υποχρεωτικά δύο δολάρια στην αρχή και δύο δολάρια στο τέλος. Οι ενδιάμεσοι χαρακτήρες μπορεί να είναι οτιδήποτε , με τον περιορισμό όμως ότι αν υπάρχει ένα δολάριο εσωτερικά ο επόμενος χαρακτήρας πρέπει να είναι μη δολάριο(έτσι αποφεύγεται η χρήση εμφωλευμένων σχολίων). Το άλλο είδος comment είναι το end_of_line_comment , το οποίο έχει ένα δολάριο στην αρχή και μετά παίρνει έναν οποιοδήποτε χαρακτήρα εκτός από eol. Οι περιορισμοί είναι ότι επόμενως χαρακτήρας από το δολάριο δεν πρέπει να είναι δολάριο και ο τελευταίος χαρακτήρας είναι υποχρεωτικά eol. Πχ. Αν δοθεί $$comment$$$ αυτό διαβάζεται ως traditional comment $$comment$$ και απλό comment $.
Το helper input_string μπορεί να έχει οποιοδήποτε χαρακτήρα εκτός από ‘’ , \ καθώς αυτά τα σύμβολα χρησιμοποιούνται με τη βοήθεια των escape sequence. Το helper input_char ορίζεται με τον ίδιο τρόπο όπως και το input_string με τη διαφορά ότι δεν δέχεται επιπλέον τον χαρακτήρα ‘ .

Tokens : Έχουν οριστεί όλες οι λέξεις κλειδιά και οι συμβολικοί τελεστές. Επίσης ορίζονται όλες οι σταθερές , τα white spaces και το όνομα μιας μεταβλητής (identifier) . Τα comments και τα white spaces γίνονται **ignore**. Οι const char σταθερές μπορούν να πάρουν οποιοδήποτε κοινό χαρακτήρα εκτός από ‘’, ‘, \ ή ένα escape sequence, μέσα σε ‘ ‘.

Productions : Τα productions έχουν υλοποιηθεί όπως περιγράφονται στην εκφώνηση της άσκησης (σελίδα -11-). Έχουν χρησιμοποιηθεί και ενδιάμεσες καταστάσεις καθώς δεν επιτρέπονται οι παραγωγές που περιλαμβάνουν εκφράσεις με παρενθέσεις που έχουν παραπάνω από ένα στοιχεία. Ειδικότερα για την αποφυγή των Shift/Reduce Conflicts χρησιμοποιούνται production για if without else και if with else. Επίσης, για το condition και για τις απλές πράξεις, η σειρά με την οποία γίνονται οι παραγωγές είναι ίδια με την προτεραιότητα των (λογικών και μαθηματικών) τελεστέων αντίστοιχα.

Print.java
Χρησιμοποιεί το concrete syntax tree για την εκτύπωση των σημαντικότερων κόμβων του δέντρου. Από το αρχείο Analysis.java δημιουργήθηκαν οι συναρτήσεις οι οποίες :
-Εκτυπώνουν το όνομα, τις παραμέτρους και τον τύπο μιας συνάρτησης όταν αυτή δηλώνεται. Αν δεν υπάρχουν παράμετροι εκτυπώνει null. Αν η συνάρτηση δηλώνεται εσψτερικά σε μία άλλη εμφανίζεται μήνυμα “local function definition”.
-Εκτυπώνουν το όνομα και τον τύπο μιας μεταβλητής όταν αυτή δηλώνεται (variable definition).
-Όταν μπαίνουμε ή βγαίνουμε από ένα block τότε εκτυπώνεται αντίστοιχο μήνυμα (block in / block out).
-Όταν έχουμε ανάθεση εκτυπώνεται το όνομα της μεταβλητής και η τιμή που ανατίθεται στην μεταβλητή.
-Όταν έχουμε κάποιο statement εμφανίζεται το όνομα του και συγκεκριμένα ποια περίπτωση έχουμε (while, if with else, if without else) . Ακολούθως εκτυπώνεται το condition και το μήνυμα “—Enter “…” Statement. Όταν βγει από το statement εμφανίζεται μήνυμα “_Exit “…” Statement.
-Όταν γίνεται κλήση μιας συνάρτησης τότε εκτυπώνεται το όνομα της και οι παράμετροι που παίρνει

Main.java
Παίρνει ως όρισμα ένα αρχείο , φτιάχνει το δέντρο και το εκτυπώνει μέσω της Print.

Abstract Syntax Tree

‘Εχει υλοποιηθεί η μετατροπή του Concrete Syntax Tree σε Abstract Syntax Tree . Για την δημιουργία του Abstract Syntax Tree έχουμε διατηρήσει σχεδόν την ίδια γραμματική της πρώτης φάσης με ελάχιστες αλλαγές. Τα στοιχεία που έχουμε προσθέσει είναι μία ακόμη παραγωγή **expr_list** (λίστα από expressions), καθώς και το token **erroneous_number** για τον εντοπισμό λάθους σε πράξεις (mod, div) όταν ο αριθμός δεν χωρίζεται με κενό με το token της πράξης.Τέλος, έχουμε επεκτείνει τις πράξεις pos και neg (εφαρμογή προσήμου), οι οποίες τώρα εφραμόζονται σε παραγωγή expr_par και όχι term (προκειμένου να δέχεται την πράξη ---3). Το traverse εφαρμόζεται στις ρουτίνες in και out του απλοποιημένου δέντρου AST. Γενικά για το AST, οι παραγωγές expressions παράγονται κατευθείαν από τις παραγωγές expr, expr_par, factor, term του CST και δημιουργούνται οι κόμβοι της αντίστοιχης πράξης. Έχουν προστεθεί ακόμη στην παραγωγή expression, οι παραγωγές function call και lvalue, προκειμένου να υπάρχει μια γενικότερη απλοποιημένη μορφή στο δέντρο. Το ίδιο συμβάνει και με τα conditions και τις αντίστοιχες παραγωγές τους. Από τις παραγωγές έχουμε αφαιρέσει τα tokens που δεν χρησιμοποιούνται (σύμβολα πράξεων, brackets, τελεστές, …) .

Semantic Analysis

Ο επιθυμητός στόχος είναι ο καλύτερος έλεγχος του προγράμματος .Η σημασιολογική ανάλυση πρέπει να παρέχει στο οπίσθιο μέρος του μεταγλωττιστή ότι πληροφορία χρειάζεται για να παράγει τον τελικό κώδικα. Για την υλοποίηση του Πίνακα Συμβόλων (SymbolTable) δημιουργούμε μία λίστα από πίνακες της δομής Hashtable. Κάθε κόμβος της λίστας αναφέρεται σε μία διαφορετική εμβέλεια.

Έχουν υλοποιηθεί οι ρουτίνες :

enter(): δημιουργία καινούργιου πίνακα. Καλείται στην caseAFunDefinition, μετά την κλήση της in.
insert(): προσθήκη στον τρέχον πίνακα. Καλείται κάθε φορά που θέλουμε να προσθέσουμε ένα αντικείμενο τύπου Record στον πίνακα.
lookup(): αναζήτηση σε κάθε πίνακα και επιστρέφει την πρώτη εμφάνιση του αντικειμένου.
exit(): καταστροφή του τρέχοντος πίνακα. Καλείται επίσης στην caseAFunDefinition, μετά την κλήση της out και την δημιουργία του ενδιάμεσου κώδικα.

Record Class

Τα αντικείμενα που δημιουργούμε και προσθέτουμε στους πίνακες είναι τύπου Record. Ως Record έχουμε ορίσει μία γενική κλάση η οποία περιλαμβάνει ένα όνομα και ένα τύπο. Για κάθε αντικείμενο που θέλουμε να προσθέσουμε καποιές ακόμη ιδιότητες χρησιμοποιούμε subclasses .Για παράδειγμα αν το αντικείμενο αυτό είναι μία συνάρτηση τότε δημιουργούμε μία δομή RecordFunction και την αρχικοποιούμε με τα επιπρόσθετα χαρακτηριστικά όπως την λίστα με τις παραμέτρους. Με την ίδια λογική έχουμε υλοποιήσει τις δομές RecordArray, RecordParam, RecordParamArray. Με casting μπορούμε το αποτέλεσμα του lookup που είναι τύπου Record να το μετατρέψουμε στον τύπο της υποκλάσης που είναι instance of και να έχουμε πρόσβαση σε όλα τα δεδομένα της.

Error Class

Για κάθε τύπο λάθους έχουμε υλοποιήσει μία αντίστοιχη συνάρτηση στην κλάση Error η οποία εκτυπώνει το λάθος που εντοπίστηκε κατά την σημασιολογική ανάλυση καθως και τη γραμμή που βρίσκεται στον κώδικα.

RecType Class

Είναι βοηθητική κλάση για τον σημασιολογικό έλεγχο. Για κάθε μεταβλητή ή σταθερά που χρησιμοποιείται στον κώδικα φτιάχνουμε ένα αντικείμενο τύπου RecType και το τοποθετούμε στη στοίβα typeStack. Αυτό περιλαμβάνει το όνομα, τον τύπο και τις διαστάσεις του. Όταν έχουμε μία αριθμητική ή λογική πράξη , assignment,… και χρειάζεται να ελέγξουμε αν οι τύποι των μεταβλητών είναι οι σωστοί και οι αναμενόμενοι ,τότε ανάλογα με την περίπτωση κάνουμε pop όσα αντικείμενα χρειάζονται από την στοίβα και συγκρίνουμε τους τύπους τους. Αφού γίνει η σύγκριση εμείς επιλέξαμε να βάζουμε στη στοίβα ανεξαρτήτως αποτελέσματος τον αναμενόμενο παραγόμενο τύπο αφενός για να μην εμφανιστούν αλυσιδωτά errors, αφετέρου πράξεις μεταξύ διαφορετικών τύπων (πχ. int + char) δεν υφίστανται στην γλώσσα. Με τον ίδιο τρόπο έχουμε υλοποιήσει και μία στοίβα με τους τύπους επιστροφής των συναρτήσεων για να τσεκάρουμε αν κάθε φορά που χρησιμοποιείται τον statement return μέσα στην συνάρτηση

επιστρέφει αντικείμενο ίδιου τύπου με τον τύπο ορισμού της συνάρτησης. Τις συναρτήσεις της standard library της Γλώσσας Grace τις εισάγωγουμε χειροκίνητα στο SymbolTable κατά την δημιουργία του. Επίσης, έχουμε ορίσει unique fixed names για τις συναρτήσεις. Πιο συγκεκριμένα για τις συναρτήσεις της βιβλιοθήκης και την συνάρτηση main το fixed name παράγεται από το όνομα της συνάρτησης που ακολουθείται απο το επίθεμα "_grace", ενώ για τι συναρτήσεις που ορίζονται στον κώδικα δημιουργείται μοναδικο id με τη βοήθεια ενός counter και χρησιμοποιείται ως επίθεμα μετά το όνομα της συνάρτησης.

Intermidiate Code
Για την παραγωγή του ενδιάμεσου Κώδικα, θα πρέπει να μην έχει εντοπιστεί κάποιο Error στον σημασιολογικό έλεγχο. Μόλις ολοκληρωθεί ο σημασιολογικός έλεγχος και δεν υπάρχει error τότε καλούμε την caseAFunDefinition του VisitorIR για τον κόμβο του δέντρου στον οποίο μόλις καλέσαμε την outAFunDefintion της κλάσης SymbolTable. Ο ενδιάμεσος κώδικας παράγεται δηλαδή πριν βγούμε από το caseAFunDefintion που βρισκόμαστε. Πρακτικά αυτό που γίνεται είναι να ξανακάνουμε traverse στον υπόδεντρο που ήμασταν, αυτή τη φορά όμως γνωρίζουμε ότι δεν υπάρχει σημασιολογικό λάθος και προχωράμε στην υλοποίηση των quads.

Quad Class
Αποτελείται από ένα label, τον operator και τα opt1, opt2 και opt3.

QuadList Class
Περιλαμβάνει τη λίστα που αποθηκεύονται τα quads και τη λίστα στην οποία αποθηκεύουμε τις προσωρινές μεταβλητές που χρησιμοποιούμε (TmpVar). Οι προσωρινές μεταβλητές έχουν όνομα και τύπο. Οι βασικότερες ρουτίνες που έχουμε υλοποιήσει είναι οι NextQuad() που επιστρέφει τον αύξοντα αριθμό για το νέο quad και GenQuad() που δημιουργεί ένα νέο quad και το βάζει στην λίστα (quadlist).

Condition Class
Περιλαμβάνει τις δύο λίστες true και false. Σε αυτές εισάγουμε τα labels των quads. Σε μια λογική συνθήκη δημιουργούμε ένα προσωρινό αντικείμενο extrCond τύπου Contition και στη συνέχεια ανάλογα τη συνθήκη εφαρμόζουμε την κατάλληλη ρουτίνα mergeLists ή swapLists . Αφού ολοκληρωθεί η συνθήκη, καλείται η συνάρτηση backPatch η οποία τοποθετεί στα "*" των quads της αντίστοιχης λίστα (true,false) στο πεδίο opt3 το κατάλληλο label.

RecordLValue/ StringLiteral
Χρησιμοποιείται με τον ίδιο τρόπο όπως και η κλάση RecType, σε συνδυασμό με τη στοίβα stackLValue.Την δομή StringLiteral την χρησιμοποιούμε καθώς δεν ορίζεται εγγραφή για τα string literal. Τα string literal αρχικά τα κάνουμε ανάθεση σε tmpVar. Την δομή RecordLValue την χρησιμοποιούμε όταν αναγνωρίζεται ένα lvalue.

Για την εκπόνηση της 3ης φάσης έχουμε δημιουργήσει αρχικά την κλάση FunctionOffsets η οποία κρατάει το offset των παραμέτρων και των τοπικών και προσωρινών μεταβλητών. Στο symbolTable έχουμε προσθέσει και μία λίστα με FunctionOffsets αντικείμενα για κάθε συνάρτηση γίνεται defined.

Activation Record
Η δομή του Activation Record ξεκινά με την πρώτη παράμετρο στη θέση ebp+16 και με την τελευταία στη υψηλότερη θέση της στοίβας. Το allignment που εφαρμόζουμε στη στοίβα του AR είναι 4 bytes ανά παράμετρο ανεξαρτήτου τύπου. Σε περίπτωση που ο τύπος είναι array τότε αποθηκεύεται η διεύθυνση του, δηλαδή σαν παράμετρος by referenced . Έχουμε προσθέσει κώδικα που τσεκάρει αν μια παράμετρος τύπου array περνιέται κατά αναφορά. Οι τοπικές μεταβλητές κάτω απο τον ebp σε μικρότερες θέσεις στην στοίβα. Μία μεταβλητή int δεσμεύει 4 bytes και τα χρησιμοποιεί όλα και μια μεταβλητή τύπου char δεσμεύει 4 bytes, αλλά χρησιμοποιεί μόνο ένα byte, για να έχουμε σωστό allignment. Μία μεταβλητή τύπου int array με μέγεθος n, δεσμεύει 4*n bytes , ενώ μια μεταβλητή τύπου char array με μέγεθος n, δεσμεύει n+k bytes , όπου k<4 είναι ο αριθμός των bytes που χρειάζονται ακόμη για να είανι ο αριθμός n+k πολλαπλάσιο του 4, λόγω allignment. Το αντίστοιχο και για τις προσωρινές μεταβλητές. Τα StringLiterals τα αποθηκεύουμε στο segment .data .Για κάθε μεταβλητή (τοπική/προσωρινή/παράμετρος) δημιουργείται ένα αντικείμενο τύπου

Record το οποίο αποθηκεύει το offset της. Το offset υπολογίζεται και αποθηκεύεται κατά της εισαγωγή της εγγραφής Record στο symbolTable. Χρησιμοποιούμε access links (θέση ebp+8) για να έχουμε πρόσβαση σε μεταβλητές που δηλωθεί σε διαφορετικές συναρτήσεις μικρότερου scope. Για να επιστρέψουμε το αποτέλεσμα της συνάρτησης αποθηκεύουμε στη θέση ebp+12 την διεύθυνση του αποτελέσματος.

Standard Grace Libraries

Υλοποιήσαμε όλες τις συναρτήσεις της βιβλιοθήκης της Grace. Οι συναρτήσεις puti, putc, geti, getc, chr και ord υλοποιήθηκαν σε assembly και χρησιμοποιούν τις συναρτήσεις printf και scanf της c. Οι υπόλοιπες υλοποιήθηκαν σε grace και μεταγλωττίστηκαν με τον μεταγλωττιστή μας. Βρίσκονται στο αρχείο libraryFunctions.s .

Optimization in Intermediate Code

Οι κλάσεις Optimise, Graph και node χρησιμοποιούνται για την βελτιστοποίηση του ενδιάμεσου κώδικα. Έχουμε εφαρμόσει βελτιστοποιήσεις τόσο σε επίπεδο block όσο και σε global δηλαδή σε επίπεδο συνάρτησης. Η κλάση Optimise δέχεται τον intermediate code που παράχθηκε για κάθε function και στην συνεχεία τον χωρίζει σε blocks και προσθέτει ακμές από ένα block σε άλλο στην περίπτωση που υπάρχουν jumps ή η ροη παρεμβάλετε από κάποιο label στο οποίο μπορεί να γίνει jump. Κάθε block αποτελεί ένα Node στον γράφο, επίσης σε κάθε node σημειώνουμε την εισερχόμενες και εξερχόμενες ακμές.

Ένα πρώτο optimization που κάνουμε είναι να αφαιρούμε τα περιττά jump στο condition. Όταν εντοπίζαμε ένα condition δημιουργούσαμε δυο quads ένα που είχε το relop, oper1, oper2, jumpLabel σε περίπτωση επιτυχίας να κάνει jump και ένα απλό jump σε άλλο label σε περίπτωση αποτυχίας. Αυτό το αντικαταστήσαμε με ένα quad που χρησιμοποιεί το αντίστροφο relation operator και κάνει jump μόνο σε περίπτωση αποτυχίας. Σε περίπτωση επιτυχίας άπλα συνεχίσουμε την ροή του προγράμματος.

Σε επίπεδο block κάναμε τρία optimization που εφαρμόζονται εναλλάξ σε κάθε quad της συνάρτησης μέχρι να μην προκύψει κάποια βελτιστοποίηση ή να ξεπεραστούν να επιτρεπτά loops βελτιστοποίησης.

- Algebra simplification: Δηλαδή μερικές άπλες απλοποιήσεις που μπορούν να γίνουν σε περίπτωση που γίνεται π.χ πρόσθεση/πολλαπλασιασμός με το μηδέν κ.α
- Constant folding: Δηλαδή να εκτελείται η πράξη μεταξύ δυο σταθερών τιμών. Οι πράξεις μπορεί να είναι αριθμητικές +,-,* κτλ ή μπορεί να είναι λογικές πράξεις όπως >,<, = κτλ. Στις λογικές πράξεις άμα η συνθήκη είναι πάντα true τότε το jump θα γίνεται πάντα οπότε αντικαταστήσουμε το quad σε ένα απλό jump. Επίσης άμα δν είναι ποτέ true το σβήνουμε. Για να σβήσουμε ενα quad το αντικαθιστούμε σε ένα nop quad που κατά την μετατροπή από ενδιάμεσο κώδικα σε τελικό αγνοείται.

- Constant propagation: Όταν γίνεται μια ανάθεση μιας σταθεράς (αριθμητική ή χαρακτήρας) σε μεταβλητή την κρατάμε σε ένα hashmap. Όταν στην συνέχεια δούμε αυτή την μεταβλητή σε ένα άλλο quad τότε κάνουμε την αντικατάσταση με την σταθερά. Δεν το εφαρμόζουμε στην περίπτωση του string literal λόγω του τρόπου που δημιουργούμε τον τελικό κώδικα από τα quads.

Σε global επίπεδο εφαρμόσαμε δυο optimization που εφαρμόζονται μετά τα block optimizations

- Constant propagation: Όπως και πριν έτσι και εδώ όταν δούμε κάποια μεταβλητή που είναι σταθερά τότε αντικαθιστούμε την τιμή της με την σταθερά. Η διάφορα είναι οτι τώρα δεν κοιτάμε μόνο το block αλλά όλη την συνάρτηση. Η μέθοδος που ακολουθήσαμε είναι αυτή που παρουσιάστηκε στο μάθημα. Όταν δούμε κάποιο function call τότε αγνοούμε οτι έχουμε βρει μέχρι τώρα και θεωρούμε ότι όλες οι μεταβλητές έχουν άγνωστη τιμή μέχρι να ξαναρχικοποιηθουν. Αυτό το κάνουμε για λόγους ασφάλειας καθώς δεν γνωρίζουμε αν η συνάρτηση αλλάζει την τιμή σε κάποια από τις μεταβλητές μας. Η grace επιτρέπει στις συναρτήσεις να αλλάζουν πειράζουν μεταβλητές άλλης συνάρτησής. Άμα είχαμε υλοποιήσει l-lifting (αντί για access links) θα μπορούσαμε να θέσουμε άγνωστη τιμή μόνο σε όσες μεταβλητές παίρνει η συνάρτηση που καλείται ως παράμετρο και έτσι

να μην ακυρώσουμε όλες τις τιμές μας άλλα λόγω χρόνου δεν το κάναμε. Επίσης οι μεταβλητές άλλου scope δεν γίνονται πάντα propagate στο επόμενο node.

- Dead code elimination: Μετα την εφαρμογή των προηγούμενων optimizations μπορεί να προκύψουν πολλές αναθέσεις χωρίς λόγο, σε μεταβλητές που απλά χρησιμοποιήθηκαν σαν βοηθητικές και δεν χρησιμοποιούνται στην συνεχεία του προγράμματος. Αυτές τις αναθέσεις τις διαγράφουμε. Αυτό το κάνουμε εφαρμόζοντας τον αλγόριθμο που προτάθηκε στην τάξη.
- Unreachable code: Επίσης μετά το constant folding σε relation operation μπορεί να προκύψουν blocks που δεν εκτελούνται ποτέ δηλαδή δεν έχουν εισερχόμενες ακμές. Αυτά τα block τα διαγράφουμε.

Όλες οι συναρτήσεις καλούνται απο την συνάρτηση run της Optimise. Άμα θέλετε να τρέξετε κάθε optimization ξεχωριστά για λόγους testing απλά βάλτε σε comment την αντίστοιχη συνάρτηση.

Το optimization δεν γίνεται πάντα αλλά πρέπει να δοθεί το κατάλληλο keyword από τον χρήστη κατά την εκτέλεση. Περισσότερες λεπτομερείς στο τμήμα Execution.

Execution

Το script grc.sh παίρνει δύο arguments, το input file που είναι ένα αρχείο .grace και το output file που είναι το εκτελέσιμο αρχείο που θα παραχθεί. Δημιουργεί αρχικά τον κώδικα x86 που αποθηκεύεται στο assembly.s και στη συνέχεια με τη χρήση του gcc ως assembler και linker των βασικών συναρτήσεων της c, παράγουμε τον εκτελέσιμο κώδικα.

./grc.sh examples/hello2.grace ot optimize 10

Το πρώτο argument ειναι το inputFile, το δεύτερο είναι το εκτελέσιμο αρχείο και προαιρετικά μπορεί να μπει το optimize και το πλήθος των loops για την βελτιστοποίηση του ενδιάμεσου κώδικα.

Επίσης στο αρχείο OutputFile.s βρίσκεται ο κώδικας x86.

Compile

Με την εντολη mvn clean package

Παρατηρήσεις

Τρέξαμε όλα τα παραδείγματα (hello.grace , bsort.grace, …) και η μεταγλώττιση γινόταν σωστά. Μόνο στο bsort μας πέταγε error , καθώς έλειπε ένα semi , αλλά το προσθέσαμε εκεί που μας έλεγε ότι το περίμενε και μετά λειτουργούσε κανονικά.

(Το script bash το δημιουργήσαμε για να τρέξουμε τα test cases του λεκτικού-σημασιολογικού ελέγχου)

Η εργασία βρίσκεται στο παρακάτω link : https://github.com/billDrett/GraceCompiler