Βασίλης Γεωργούλας
ΑΜ 2954
Μεταφραστές 2021

Φαση 1

Υλοποίηση λεκτικού αναλύτη.

Η κύρια συνάρτηση η οποία υλοποιεί τον λεκτικό αναλυτή είναι η tokenResolver η οποία τελικά επιστρέφει ένα αντικείμενο τύπου token το οποίο περιέχει τον τύπο , την τιμή και την γραμμή στην οποία αναγνωρίστηκε μια λεκτική μονάδα .Η λειτουργιά της ακολουθεί το αυτόματο λειτουργιάς του λεκτικού αναλυτή , και πιο συγκεκριμένα για κάθε εισερχόμενο χαρακτήρα ακολουθεί την αντίστοιχη 'μετάβαση' που θα το οδηγήσει στην κατάλληλη τελική η και ενδιάμεση κατασταση.Βεβαια αν το εισερχόμενο σύμβολο δεν ανήκει στην γλώσσα θα οδηγηθούμε σε σφάλμα και η μετάφραση θα διακοπεί.

Για κάθε τελική η ενδιάμεση κατάσταση έχει δημιουργηθεί και μια συνάρτηση η οποία θα καθορίσει τον τύπο της λεκτικής μονάδας η και την απόρριψη της κάνοντας τους κατάλληλους έλεγχους. Επίσης για τους έλεγχους έχουν δημιουργηθεί λεξικά που αποθηκεύουν τις λέξεις/σύμβολα κλειδιά της Cimple για εύκολους και γρήγορους έλεγχους.

Παραδείγματα κλήσης λεκτικού αναλύτη.

Με εισερχόμενο χαρακτήρα > συνάρτηση tokenResolver κάνοντας έλεγχο του με κάθε πιθανό εισερχόμενο σύμβολο αποφασίζει ότι το > ανήκει στους σχεσιακούς τελεστές (κάνοντας έλεγχο στο αντίστοιχο λεξικό) και οδηγείται στη ενδιάμεση κατάσταση relOp. Για να αποφασίσει η relOp τελικά ποια θα είναι η λεκτική μονάδα αποθηκεύει το > σε μια προσωρινή μεταβλητή και κάνει ένα ακόμα βήμα χωρίς όμως να καταναλώσει τον επόμενο χαρακτηρα. Επειτα για παράδειγμα αν ο ο επόμενος χαρακτήρας είναι = επιστρέφει >= η αν είναι αλφαριθμητικό η καινό η πρόσημο επιστρέφει >.

Αν για παράδειγμα μεταβούμε στην relOp με = τότε κάνουμε ένα βήμα και το επιστρέφουμε αμέσως.

Όμοια λειτουργούμε με είσοδο χαρακτήρα string. Μεταβαίνουμε στην κατάσταση identOrKey και όσο συνεχίζουμε να διαβάζουμε χαρακτήρες χτιζουμε παράλληλα th λέξη όσο όμως οι χαρακτήρες είναι αλφαριθμητικά . Τελικά αποφασίζουμε αν θα επιστρέψουμε keyword η id κοιτάζοντας στο λεξικό των keywords.

Υλοποίηση συντακτικού αναλύτη .

Ο συντακτικός αναλύτης είναι αυτός που θα καθορίσει αν το πρόγραμμα μας είναι τελικά μέσα στη γλώσσα Για να το επιτύχει αυτό σε κάθε βήμα καλεί τον λεκτικό αναλύτη για να λάβει την επομένη λεκτική μονάδα και για κάθε μη τερματικό σύμβολο που λαμβάνει καλεί το αντίστοιχο υποπρογραμμα . Όταν αναγνωριστεί και η τελευταία λέξη του πηγαίου προγράμματος έχουμε επιτυχία.

Για κάθε κάνονα της γραμματικής έχει υλοποιηθεί και μια συνάρτηση και κάθε φορά με βάση το τρέχον τόκεν αποφασίζει ποιον κάνονα να ακολουθήσει ακολουθώντας πάντα τη λογική της αναδρομικής κατάβασης.Στην περίπτωση που είναι διαφορετικό από αυτό που περίμενε ο συντακτικός αναλύτης τότε αναλαμβάνει δράση ο χειρίστης σφάλματος. Το τοκεν κάθε φόρα λαμβάνεται καλώντας τη συνάρτηση lex() μετά την κατανάλωση . Με τη σειρά της η lex() καλεί την συνάρτηση tokenResolver και όσο βρίσκει comment_tk το αγνοεί αλλιώς επιστρέφει το tokenType.

Λεπτομέρειες συντακτικού αναλυτή.

Program: Είναι η εναρκτήρια συνάρτηση από την οποία αρχίζουμε να χτίζουμε το δένδρο της αναδρομικής κατάβασης . Γεμίζει το τόκων για πρώτη φόρα και καλεί την block.Τελος αν όλα πάνε κάλα θα επιστρέψουμε μετά την κλήση της block και θα τερματίσουμε ομαλά , αν βέβαια το τόκεν περιέχει το σύμβολο τερματισμού.

Block: Υλοποιεί την κύρια δομή του προγράμματος η οποία αποτελείται από τα declarations , subprograms statements.

Declarations : Μάζι με τον κάνονα var list ορίζει μια λίστα με τα declarations , αποτελούμενα από identifiers.

Subprograms: Κάθε πρόγραμμα μετά τα declarations μπορεί να έχει ένα η περισσότερα υποπρογραμμτα.Καθε υποπρογραμμα μπορεί να είναι μια διαδικασία η μια συνάρτηση που με τη σειρά τους μπορεί να έχουν declarations και έπειτα φωλιασμένα ένα η περισσότερα blocks. Επίσης μπορεί να δέχονται και παραμέτρους.

Statements: Τέλος ένα πρόγραμμα η και υποπρογραμμα μπορεί να έχει ένα η περισσότερα statements. Κάθε statement είναι πρακτικά μια από τις δομές της Cimple (if , switchcase , while , print , assign...) . Κάποια από τα statements μπορεί , με βάση τον κάνονα , να έχουν ένα η περισσότερα statements και να βρίσκονται η όχι μέσα σε {} .

Formalparlist: Η συνάρτηση αυτή έχει την αρμοδιότητα του εντοπισμού των παραμέτρων μια συνάρτησης η διαδικασίας. Καλώντας επαναληπτικά την formalparitem δημιουργεί μια λίστα μεταβλητών που θα περαστούν ως παράμετροι στο υποπρογραμμα . Καθε παράμετρος θα πρέπει να συνοδεύεται από in h inout αν είναι με τιμή η αναφορά.

Expression: Ορίζει μια αριθμητική παράσταση μέσα στην οποία μπορεί να έχουμε και κλήσεις υποπρογραμματων.

Condition: Ορίζει μια λογική παρασταση αποτελούμενη από λογικούς , σχεσιακούς τελεστές και expressions που τελικά θα αποτιμηθεί σε αληθές η ψευδές.

Ο έλεγχος του λεκτικού/συντακτικού αναλυτή έγινε με τα αρχεία cimple1.ci , cimple2.ci , cimple3.ci.

Φάση 2

Παράγωγη ενδιάμεσου κωδικά.

Στη φάση αυτή το πηγαίο πρόγραμμα μεταφράζεται σε ένα ισοδύναμο πρόγραμμα , γραμμένο σε ενδιάμεση γλώσσα .Η ενδιαμεση γλώσσα είναι χαμηλότερου επιπέδου από την αρχική άλλα υψηλότερη από την τελική και στην περίπτωση μας αποτελείται από τετραδες.Καθε τετράδα είναι και μια εντολή και εχεί τη μορφή op , x, y ,z . Η διαδικασία αυτή διευκολύνει το έργο της μετάφρασης και της βελτιστοποίησης του παραγόμενου κωδικά Η παράγωγη του ενδιάμεσου κωδικά γίνεται με τροποποίηση του συντακτικού αναλυτή ώστε να παράγονται οι τετράδες για κάθε δομή της Cimple.

Λεπτομέρειες παράγωγης ενδιάμεσου κωδικά.

Εκφρασεις :

Κάθε αριθμητική έκφραση παράγεται με βάση τους κάνονες:

expression -> sign tleft (ADD_OP tright)∗ (κωδηκας οπως στις διαφανειες)
term -> fleft (MUL_OP fright)∗
factor -> INT | (expression) | ID idtail

Όσο ο κάνονας expression συναντάει + η - κατασκευάζει μια έκφραση της μορφής tleft addop tright σε κάθε επανάληψη και παράγει την τετράδα (tleft , addop , tright , newTem()) Έπειτα h τιμή newTemp() η οποία κρατάει το αποτέλεσμα θα λειτουργήσει ως tleft στην επομένη επανάληψη για τη δημιουργία της επομένης τετράδας ,αν βέβαια συναντήσουμε και άλλον addOperator στην έκφραση. Με αυτόν τον τρόπο κατασκευάζουμε αλυσιδωτά μια αριθμητική έκφραση και τέλος επιστρέφουμε το τελικό αποτέλεσμα newTemp() της έκφρασης σε οποία δομή απαιτητέ.

Ο κάνονας expression σε κάθε βήμα τροφοδοτείται με τις τιμές tleft , tright καλώντας την term.Η λειτουργιά της term ακολουθεί την ίδια λογική με την διάφορα ότι τώρα παράγονται τετράδες (fleft ,

mulop , fright , newTem()) Δηλαδή κάθε term θα επιστρέφει το αποτέλεσμα του πολλαπλασιασμού/διαίρεσης μεταξύ παραγόντων για προσθαφαίρεση στην expression.

Τέλος οι παράγοντες στέλνονται προς την term από την factor . Κάθε παράγοντας μπορεί να είναι ενας ακέραιος , ένας identifier η και αποτέλεσμα κλήσης μιας συναρτησης.Στην περίπτωση που πράγματι έχουμε κλήση υποπρογραμματος τότε θα θέλαμε να στείλουμε ως factor το retv T_i στην term. Οπότε εδώ θα κληθει η id tail η οποία επίσης μας πληροφορεί αν όντως έχουμε κλήση και αν όντως έχουμε , θα παράξει τις τετράδες με τα ορίσματα και τελικά θα επιστρέψει το retv . Για την παράγωγη τον τετράδων αυτών καλεί την actualparlist η οποία όσο βρίσκει νέα ορίσματα τα αποθηκεύει για να τα στείλει πίσω στην idtail ώστε τελικά να παράξουμε της τετράδες των ορισμάτων με την σωστή σειρά Επίσης είναι πιθανό να εντοπίσουμε εμφολευμενο expression μέσα σε παρένθεσεις.Πάλι θα θέλαμε να στείλουμε προς τα πάνω την αποτίμηση αυτής της έκφρασης ως ένα id (T_i , η , id ,integer).

Κάθε όρισμα μπορεί να είναι CV h REF και επίσης μπορεί να είναι ένα expression η ακόμα και μια εμφολευμενη κληση.Αυτο εντοπίζεται στην actualparitem. Αν είναι expression , για την παράγωγη του τελικού T_ι της έκφρασης καλείται h expression πριν στείλουμε το par id στην id tail . Προφανώς αν έχουμε εμφολευμενη κλήση η παραπάνω κλήση της expression θα το χειριστεί με τον ίδιο τρόπο.

Η expression επίσης θα πρέπει να τσεκάρει αν έχουμε κάποιο πρόσημο στη αρχή της κλήσης . Αν έχουμε + το αγνοουμε τελείως , αν όμως έχουμε - παράγουμε μια επιπλέων τετράδα ('-', 0, tleft, temp) και έπειτα θέτουμε το πρώτο tleft = temp.

<u>Παραδείγματα έλεγχου αριθμητικών παραστάσεων</u>.

1) x := 3*x + f1(in f2(in x) , in -(f3(in z)));

1: * , 3 , x , T_0 πρώτο tleft που δημιουργείται από την term καλώντας την factor με fleft = 3 , fright = x , T_0 επιστρέφεται πρως τα πανω

2: par , x , CV , _ η επομένη κλήση για το tright εντοπίζει κλήση -> δημιουργία τετράδων με ορίσματα

3: par , T_1 , RET , _ η κλήση της expression για το πρώτο in όρισμα εντοπίζει εμφολευμένη κλήση

4: call , _ , _ , f2 κλήση εμφολευμενης , T_1 κρατάει το πρώτο input , το οποίο αποθηκεύεται σ την actualparlist

5: par , z , CV , _ επόμενο input ->κλήση expression απο pari tem-> δεύτερη εμφολευμενη

6: par , T_2 , RET , _ κλήση δεύτερη εμφολευμενης T_2 κρατάει το αποτέλεσμα της f3 επιστρέφει στην expression

7: call , _ , _ , f3 έχοντας πλέον το T_2 ως tleft η expression χειρίζεται το πρόσημο , το επιστρέφει στην paritem - > actualparlist

8: - , 0 , T_2 , T_3

9: par , T_1 , CV , _ πλέον η parlist επέστρεψε μια λίστα με ορίσματα στην ιdtail -> παράγωγη -> τετράδων par , RET

10: par , T_3 , CV , _

11: par , T_4 , RET , _ επιστροφή retv προς τα πάνω T_4 (factor)

12: call , _ , _ , f1

13: + , T_0 , T_4 , T_5 πλέον η expression έχει και το tright T_4 (αρχικο tright)

14: := , T_5 , _ , x το χ παίρνει την τελική τιμή T_5

Κατά την κλήση των f2 , f3 ακολουθείται η ίδια διαδικασία με την αποθήκευση και παράγωγη . τετράδων par/Ret/CALL.

2) x:= -(x + (-(3 + 10*y))) + z; Πολλαπλά εμφολευμενα expressions.

1: * , 10 , y , T_0
2: + , 3 , T_0 , T_1
3: - , 0 , T_1 , T_2
4: + , x , T_2 , T_3
5: - , 0 , T_3 , T_4
6: + , T_4 , z , T_5
7: := , T_5 , _ , x

Λογηκες Παραστασεις:

Οι λογικές παραστάσεις παράγονται με βάση τους κάνονες

condition -> boolterm (or boolterm)∗ (κωδηκας οπως στις διαφανειες)
boolterm -> boolfactor (and boolfactor)∗
boolfactor -> not [condition] | [condition] | expression REL_OP expression.

Μια λογική παράσταση στη Cimple αποτελείται από αριθμητικές εκφράσεις χωριζόμενες από σχεσιακούς τελεστές. Σε μια συνθήκη μπορούμε να έχουμε μια η περισσότερες λογικές παραστάσεις χωριζόμενες από τους λογικούς τελεστές της Cimple and , or , not.

Η παράγωγη των τετράδων που χρειαζόμαστε για την αναπαράσταση των λογικών παραστάσεων γίνεται στην boolfactor . Εκεί για κάθε έκφραση της μορφής leftExp relOp rightExp που απαρτίζει την λογική παράσταση δημιουργούμε 2 τετράδες . Για την αληθή αποτίμηση genQuad(op, left, right, '_') και genQuad('jump', '_', '_', '_') για την ψευδή Επίσης δημιουργούμε 2 λίστες Rtrue , Rfalse οι οποίες περιέχουν τις ετικέτες αυτών των 2 τετράδων.

Έπειτα ανεβαίνοντας προς την boolterm επιστρέφουμε αυτές τις 2 λίστες . Εφόσον η boolterm δημιουργεί παραστάσεις χωριζόμενες από το and μπορούμε στο σημείο αυτό να συμπληρώσουμε τις τετράδες με αληθείς αποτιμήσεις με την επομένη τετράδα. Επίσης για την μη αληθή αποτίμηση γνωρίζουμε ότι αν το boolfactor που ήρθε είναι ψευδής τότε όλη η boolterm θα είναι ψευδής οπότε μπορούμε να συγχωνεύσουμε όλες τις ψευδής αποτιμήσεις σε μια λίστα (ετικέτες τετράδων) και αυτές θα κάνουν jump στο ίδιο σημείο (άγνωστο στο σημείο αυτό)Πρέπει επίσης να αποθηκεύσουμε τη λίστα των αληθών αποτιμήσεων της boolterm.

Τέλος οι τροποποιημένες αυτές λίστες (Qtrue , Qfalse) που περιέχουν τις τετράδες για την αποτίμηση του συγκεκριμένου boolterm της λογικής έκφρασης επιστρέφεται στην condition . Η condition παράγει παραστάσεις χωρισμένες από or . Έτσι σε αντίθεση με την boolterm μπορούμε να συμπληρωσουμε τις τετράδες με την ψευδή αποτίμηση με την επομένη τετράδα , και να συγχωνεύσουμε όλες τις αληθής αποτιμήσεις . Πρέπει επίσης να αποθηκεύσουμε τη λίστα των ψευδών αποτιμήσεων της condition.

Πλέον κάθε φορά που ένα statement καλεί ένα condition θα λαμβάνει τις λίστες (Btrue , Bfalse) οι οποίες περιέχουν τι ετικέτες σε τετράδες των αληθών/ψευδών που απαρτίζουν την condition και αναμένουν να συμπληρωθούν.

Η γραμματική της Cimple επιτρέπει να έχουμε και φωλιασμένα conditions μέσα σε [] που μπορεί να συνοδεύονται από τον τελεστή not . Στην περίπτωση αυτή η κλήση της condition άπλα θα αντιστρέψει τις λίστες που έλαβε από την δεύτερη κλήση και θα τις επιστρέψει.

<u>Δομες</u>

IfStat->if (condition{p1}) {p2} statements{p3} elsepart{p4}

{p1}: (Btrue, Bfalse) = condition() Λήψη ληστών με της ετικέτες τετράδων της condition
{p2}: backpatch(Btrue, nextQuad()) Συμπλήρωση αληθών τετράδων με το label της πρώτης εντολής statements.
{p3}: ifList = makeList(nextQuad()) Δημιουργία τετράδας για έξοδο στην περίπτωση της αληθής αποτίμησης.
genQuad('jump', '_', '_', '_')
backpatch(Bfalse , nextQuad()) Συμπλήρωση ψευδών τετράδων με το label της πρώτης εντολής statements στο else.
{p4}: backpatch(ifList , nextQuad()) Συμπλήρωση τετράδας έξοδου με την πρώτη εντολή μετά το else

whileStat -> while ({p1} condition){p2} statements{p3}

{p1}: Bquad = nextQuad() Αποθικευση ετηκετας πρωτης εντολης τη condition
(Btrue , Bfalse) = condition() Κληση condition καθε φορα κατα τον ελεγχο
{p2}: backpatch(Btrue , nextQuad()) Συμπλήρωση αληθών τετράδων με το label της πρώτης εντολής statements.
{p3}: genQuad('jump', '_', '_', Bquad) Δημιουργία τετράδας για αλμα στην condition
backpatch(Bfalse , nextQuad()) Συμπληρωση τετραδων ψευδης αποτιμησης με την τετραδα εξω απο while

incaseStat -> incase {p1} (case ({p2} condition){p3} statements {p4})∗{p5}

{p1}: temp = (newTemp() , nextQuad())
genQuad(‘:=’ , ‘0’ , ‘_’ , temp[0]) δημιουργια βοηθιτηκης τετραδας
{p2}: (Btrue , Bfalse) = condition() λυψη τετραδων condition για καθε case
{p3}: backpatch(Btrue , nextQuad()) συμληρωση αληθων αποτημισεων
{p4}: genQuad(':=' , '1' , '_' , temp[0]) αλλαγη προσωρηνης μεταβλητης με την ολοκληρωση των statements του case
backpatch(Bfalse , nextQuad()) συμπηρωση ψευδων αποτημισεων με τετραδα εξω απο το case
{p5}:genQuad('=' , '1' , temp[0] , temp[1]) τελος αν εχουμε μπει σε καποιο case επιστροφη στην αρχη

Switchcase -> {p1}(case (condition{p2}){p3} statements1{p4})∗ default statements2{p5}

{p1}: exit = emptyList() λιστα για αποθυκευση αλματων εξοδου
{p2}: (Btrue , Bfalse) = condition() λυψη τετραδων condition για καθε case
{p3}: backpatch(Btrue, nextQuad()) συμληρωση αληθων αποτημισεων με την πρωτη τετραδα στα statements
{p4}: exit.append(nextQuad()) αποθυκευση τετραδων εξοδου
genQuad('jump', '_', '_', '_') θα μας οδηγηση εξω απο την default αν εκτελεστει καποιο case
backpatch(Bfalse , nextQuad()) συμπηρωση ψευδων αποτημισεων με πρωτη τετραδα εξω απο το case
{p5}: backpatch(exit , nextQuad()) συμπληρωση τετραδων εξοδου
*αν ακολοθησουμε τα false jumps συνεχεια θα φτασουμε στην default

Forcase ->{p1} (case (condition{p2}){p3} statements1{p4})∗ default statements2

{p1}: Bquad = newTemp() αποθυκευση της αρχης
{p2}: (Btrue,Bfalse) = condition() λυψη τετραδων condition για καθε case
{p3}: backpatch(Btrue , nextQuad()) συμληρωση αληθων αποτημισεων με την πρωτη τετραδα στα statements
{p4}: genQuad(‘jump’ , ‘_’ , ‘_’ , Bquad) επιστροφη στην αρχη οταν τελιωσουμε με ενα statement
backpatch(Bfalse , nextQuad()) συμπηρωση ψευδων αποτημισεων με πρωτη τετραδα εξω απο το case
*αν ακολοθησουμε τα false jumps συνεχεια θα φτασουμε στην default

AssignStat(name) -> ID := expression{p1}

{p1}: genquad(‘:=’ , expression() , ‘_’ , name)

PrintStat -> print(expression{p1})

{p1}: genQuad('out' , expression() , '_' , '_')

InputStat -> input(ID {p1})

{p1}: genQuad('inp' , ID , '_' , '_')

ReturnStat -> return(expression{p1})

{p1}: genQuad('retv' , expression() , '_' , '_')

callStat -> call ID(actualparlist {p1}){p2} (ομοια με idtail στην περιπτωση που εχουμε κληση συναρτησης σε expression)

{p1}: parlist = actualparlist() λιστα παραμετρους με καθε στοιχιο (parId , parType)

{p2}: new_temp = newTemp()

```
for par in parlist:
   genQuad('par', par[0], par[1], '_')
genQuad('par', new_temp, 'RET', '_')
genQuad('call', '_', '_', name)
return new_temp
```

Block(name) -> declarations{p1} subprograms statements {p2}

{p1}: genQuad(‘begin_block’ , name , ‘_’ , ‘_’)

{p2}: if name == programName:

```
    genquad(‘halt’ , ‘_’ , ‘name’ , ‘_’)
genquad(‘end_block’ , name , ‘_’ , ‘_’)
```