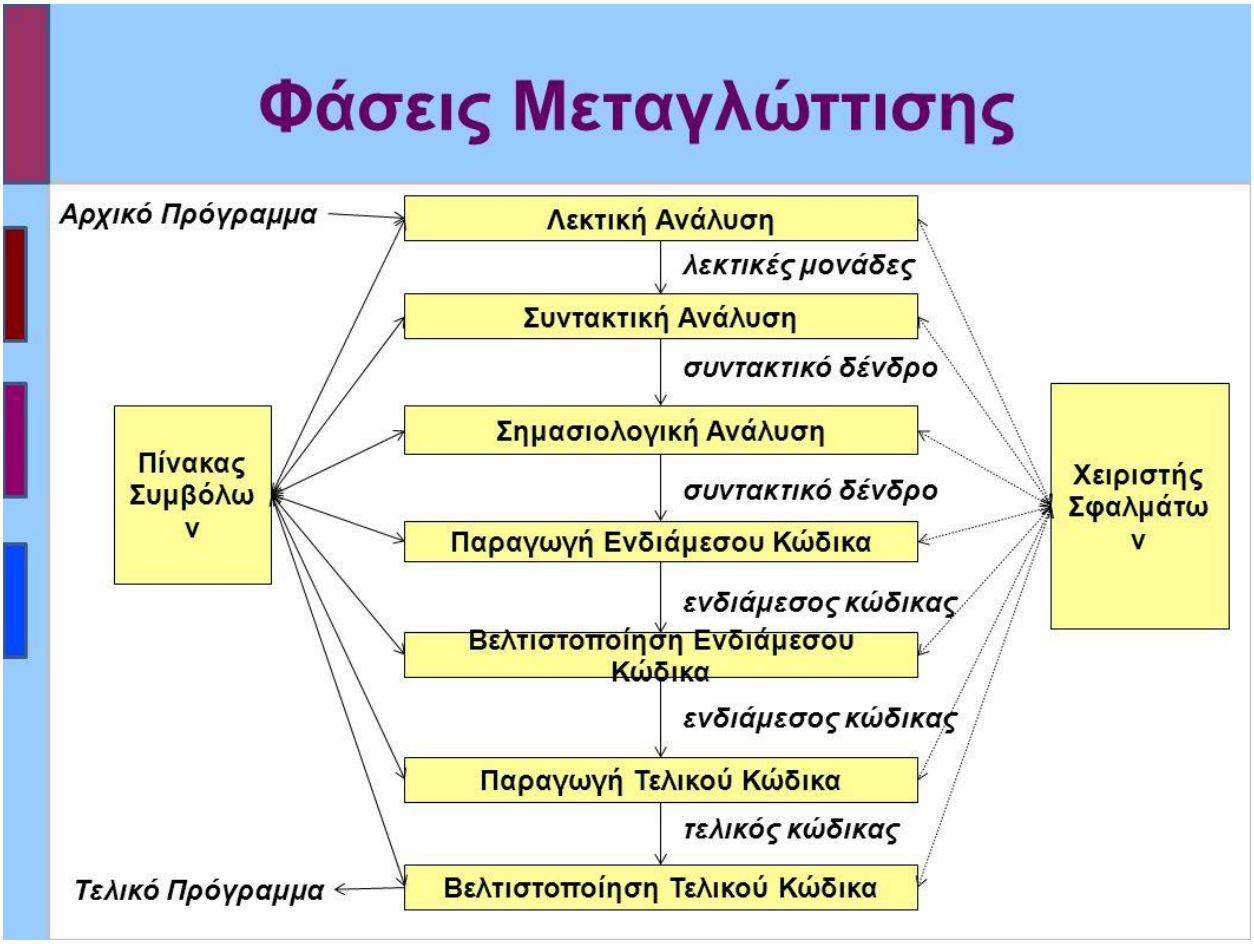
Βασίλης Γεωργούλας
ΑΜ 2954
Μεταφραστές 2021

Στόχος της άσκησης αυτής είναι η κατασκευή ενός μεταγλωττιστή για τη γλώσσα Cimple. Η Cimple είναι μια μικρή γλώσσα προγραμματισμού η οποία αντλεί στοιχειά από άλλες γλώσσες όπως η C και η Pascal. Ο μεταγλωττιστής θα παίρνει ως είσοδο το πηγαίο πρόγραμμα και θα παράγει το ισοδύναμο σε γλώσσα Assembly. Στην ασκηση αυτη θα αγνοησουμε το κομματι της βελτιστοποιησης.

Η φάσεις της μεταγλώττισης που θα ακολουθήσουμε είναι οι έξης:

•Λεκτική Ανάλυση
•Συντακτική Ανάλυση
•Παράγωγη Ενδιάμεσου Κωδικά
•Πινάκας Συμβόλων/Σημασιολογική ανάλυση
•Παράγωγη τελικού Κωδικά

Η δομη ενος μεταγλωττιστή φαίνεται και παρακατω:



Φαση 1

Λεκτικος αναλύτης.

Ο λεκτικός αναλυτής αποτελεί τον πρώτο μηχανισμό μετάφρασης του προγράμματος. Δέχεται ως είσοδο το αρχικό πηγαίο πρόγραμμα με τη μορφή μιας συμβολοσειράς χαρακτήρων και κάθε φορά επιστρέφει στην επομένη φάση την επομένη λεκτική μονάδα. Οι λεκτικές αυτές μονάδες είναι τερματικά σύμβολα της γραμματικής που περιγραφεί τη σύνταξη της Cimple.

Λεπτομέρειες λεκτικου αναλυτή.

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 (παράρτημα), 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 επόμενος χαρακτήρας είναι = επιστρέφει >= η αν είναι αλφαριθμητικό η καινό η πρόσημο η group symbol 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.

Συντακτικος αναλύτης .

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. Παράγει μία δομή που ονομάζεται συντακτικό δένδρο (παραρτημα).

Για κάθε κάνονα της γραμματικής που περιγραφει τη συνταξη της Cimple έχει υλοποιηθεί και μια συνάρτηση και κάθε φορά με βάση το τρέχον τόκεν αποφασίζει ποιον κάνονα της γραμματικης να ακολουθήσει . 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 τόκεν 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 με τις παράμετρους του υποπρογραμματος. Καθε παράμετρος θα πρέπει να συνοδεύεται από in h inout αν είναι με τιμή η αναφορά.
Actualparlist: Φτιάχνει μια λίστα με τις παραμέτρους κλησης καλώντας επαναληπτικά την actualparitem και τις επιστρέφει στην idtail η στην callstat. Καθε στοιχειο της λιστας ειναι της μορφης (parId , parType).

Idtail: Καλείται στη περίπτωση που εντοπίσουμε id σε expression. Τότε είναι πιθανό να έχουμε κλήση συνάρτησης. Λαμβάνει τη λίστα παραμέτρων από την actualparlist αν έχουμε κλήση αλλιώς ενημερώνει την factor ότι δεν έχουμε κλήση.
Expression: Ορίζει μια αριθμητική παράσταση απο αριθμητικες παραστασεις χωρισμενες απο +, - μέσα στην οποία μπορεί να έχουμε και κλήσεις υποπρογραμματων. Ενα expression δημιουργείται καλώντας επαναληπτικά τους κάνονες factor, term.
Condition: Ορίζει μια λογική παρασταση αποτελούμενη από λογικούς, σχεσιακούς τελεστές και expressions που τελικά θα αποτιμηθεί σε αληθές η ψευδές. Η λογικη παρασταση που απαρτιζει ενα condition παραγεται καλωντας επαναληπτικα τους κανονες boolterm, boolfactor.
AssignStat: αποτελει τον κανονα εκχωρισης τιμης ή αποτελεσμα εκφρασης σε μεταβλητη.
PrintStat: αποτελει τον κανονα τυπωσης τιμης ή αποτελεσμα εκφρασης.
InputStat: εκχωρει μια τιμη απο το πληκτρολογιο σε μια μεταβλητη του προγραμματος.
ReturnStat: κανονας επιστροφης τιμης συναρτησης. Μπορει να εχει και αποτελεσμα expression.
CallStat: κανονας για κληση διαδηκασιων.
Factor: εντοπιζει εναν παραγοντα μιας αριθμητίκης εκφρασης. Με βαση το τοκεν θα ειναι ενα id, ενας ακεραιος η και μια εμφολευμενη εκφραση σε παρενθεσεις. Αν βρει id τοτε μπορει να εχουμε κληση συναρτησης οποτε καλει την idtail.
Term: παραγει μια εκφραση απο παραγοντες χωρισμενους απο * και / καλωντας την Factor.
Boolfactor: με βαση το τοκεν παραγει μια σχεσιακη εκφραση ή εντοπιζει μια εμφολευμενη συνθηκη μεσα σε αγκυλες που μπορει να συνοδευεται απο τον λογικο τελεστη not.
Boolterm: φτιαχνει μια λογικη παρασταση απο λογικους παραγοντες χωρισμενους απο τον τελεστη and.
ConditionalLoops: Τέλος οι κάνονες **switchcase**, **incase**, **forcase**, **while**, **if** ορίζουν τους λογικούς επαναλήπτες της Cimple. Εκτελούν τα αντίστοιχα statement με βάση κάποια συνθήκη.

Φάση 2

Παράγωγη ενδιάμεσου κωδικά.

Στη φάση αυτή το πηγαίο πρόγραμμα μεταφράζεται σε ένα ισοδύναμο πρόγραμμα , γραμμένο σε ενδιάμεση γλώσσα, η οποια αποτελει μια γραμμική αναπαράσταση του συντακτικού δένδρου .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.

Βοηθητικές συναρτήσεις ενδιάμεσου κωδικά.

- **newTemp():** δημιουργεί και επιστρέφει μία νέα προσωρινή μεταβλητή
- **genquad(op, x, y, z):** δημιουργεί την επόμενη τετράδα (op, x, y, z)
- **nextquad():** επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί
- **emptylist():** δημιουργεί μία κενή λίστα ετικετών τετράδων
- **makelist(x):** δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x
- **merge(list1, list2):** δημιουργεί μία λίστα ετικετών τετράδων από τη συνένωση των λιστών list1, list2
- **backpatch(list,z):** η λίστα list αποτελείται από δείκτες σε τετράδες των οποίων το τελευταίο τελούμενο δεν είναι συμπληρωμένο. Η backpatch επισκέπτεται μία μία τις τετράδες αυτές και τις συμπληρώνει με την ετικέτα z

Λεπτομέρειες παράγωγης ενδιάμεσου κωδικά.

Εκφρασεις :

Κάθε αριθμητική έκφραση παράγεται με βάση τους κάνονες:

expression -> sign term (ADD_OP term)∗
term -> factor (MUL_OP factor)∗
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 η 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(), id η int 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).

Κατα την κληση ενος υποπρογραμματος 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 αναλογα και θα εχουμε ως παραμετρο τη μεταβλητη που κρατάει την τιμη επιστροφης.

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)∗
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, εκφραση στα δεξια του and. 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 για την επομενη επαναληψη η επιστροφη.

Τέλος οι τροποποιημένες αυτές λίστες (Qtrue, Qfalse) που περιέχουν τις τετράδες για την αποτίμηση του συγκεκριμένου boolterm της λογικής έκφρασης επιστρέφονται 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 δηλαδη την εκφραση στα δεξια του or, και να συγχωνεύσουμε όλες τις αληθής αποτιμήσεις. Πρέπει επίσης να αποθηκεύσουμε τη λίστα των ψευδών αποτιμήσεων της condition για την επομενη επαναληψη η επιστροφη.

Πλέον κάθε φορά που ένα statement καλεί ένα condition θα λαμβάνει τις λίστες (Btrue, Bfalse) οι οποίες περιέχουν 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.

Δομες:

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() Αποθηκευση 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 βοηθητικης τετραδας
{p2}: (Btrue , Bfalse) = condition() ληψη τετραδων condition για καθε case
{p3}: backpatch(Btrue , nextQuad()) συμληρωση αληθων αποτημισεων
{p4}: genQuad(':=' , '1' , '_' , temp[0]) αν μπουμε σε statement τοτε temp[0] = 1 (αλλαγη σε run time)

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() ληψη τετραδων condition για καθε case

{p3}: backpatch(Btrue, nextQuad()) σημπληρωση αληθων αποτημισεων με την πρωτη τετραδα στα statements

{p4}: exit.append(nextQuad()) αποθηκευση 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() ληψη 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({p1}actualparlist{p2}) (ομοια με idtail στην περιπτωση που εχουμε κληση συναρτησης σε expression)

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

{p2}: new_temp = newTemp() περιεχει ολλες τις τυπικες παραμετρους της συγκεκρημενης κλησης

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

idTail -> ({p1}actualparlist{p2})| ε

{p1}: parlist = actualparlist()
{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(block) -> declarations subprograms {p1} statements {p2}

{p1}: genQuad('begin_block' , name , '_' , '_')
{p2}: if block[1]:
genquad('halt' , '_' , 'name' , '_')
genquad('end_block' , name , '_' , '_')

Φάση 3

<u>Πινακας συμβολων.</u>

Κάθε μεταγλωττιστής χρειάζεται να συγκεντρώνει πληροφορίες για τα ονόματα που εμφανίζονται στο αρχικό πρόγραμμα, και στη συνεχεία να τις χρησιμοποιεί κατά τη διάρκεια της μεταγλώττισης. Στη Cimple τα ονόματα που εμφανίζονται είναι το ίδιο το πρόγραμμα, ονόματα υποπρογραμματων, μεταβλητές και παράμετροι.

Περιεχόμενα και ιδιότητες του πινάκα συμβολών.

Στον πινάκα συμβολών το κύριο στοιχειό που αποθηκεύεται είναι τα scopes η αλλιώς εμβέλειες του αρχικού προγράμματος. Μια εμβέλεια είναι μια δομική μονάδα του προγράμματος που περιέχει δηλώσεις μια η περισσότερων μεταβλητών εκτός από τις εντολές. Στη Cimple οι δομικές μονάδες είναι το κύριο πρόγραμμα, οι συναρτήσεις και οι διαδικασίες. Επίσης η Cimple μας επιτρέπει να έχουμε και αιμφολευμενες εμβέλειες η μια μέσα στην αλλη. Με αυτόν τον τρόπο είμαστε σε θέση να ορίσουμε και την ορατότητα των μεταβλητών και συναρτήσεων.

Στη Cimple μια μεταβλητή είναι ορατή στην εμβέλεια που δηλώνεται και σε κάθε εμβολιασμένη εμβέλεια με την προϋπόθεση ότι δεν υπάρχει ξαναδηλωμενη κάπου εσωτερικά. Τότε η εσωτερική

δήλωση θα επισκίαση την εξωτερική της δήλωση. Επίσης με βάση τον πινάκα συμβολών ένα scope μπορεί να καλέσει όλους τους πρόγονους του, κάθε scope αδερφό εφόσον προηγείται από αυτό και όλα τα scopes δηλωμένα σε αυτό.

Δομικά στοιχειά κάθε scope

Σε κάθε scope μετά τη δημιουργία του και κατά τη μετάφραση καταγραφούμε σε μια λίστα όλα τα entities που εντοπίζουμε. Κάθε entity μπορεί να αντιπροσωπεύει μια παράμετρο με τιμη η αναφορα, τοπική μεταβλητή, προσωρινή μεταβλητή, καθολική μεταβλητή η και ένα αποτύπωμα ενός υποπρογραμματος που ορίζεται στο scope. Για τις μεταβλητές επίσης αποθηκεύουμε και τη θέση στο εγγράφημα δραστηριοποίησης οπού βρίσκονται, ένω για τα υποπρογραμματα έχουμε το συνολικό frameLength του ΕΔ που καταλαμβάνει, τον τύπο παραμέτρων και την ετικέτα της τετράδας του begin_block.

Οργάνωση του πινάκα συμβολών

Οι βασικές λειτουργιές που χρειάζεται ο πινάκας είναι η αναζήτηση, πρόσθεση ενός scope η entity, διαγραφή scope. Δημιουργούμε ένα scope καθώς εντοπίζουμε ένα υποπρογραμμα, προσθετουμε τα entities των ορισματων και δηλωσεων και έπειτα αν εντοπίσουμε κάποιο φωλιασμένο scope τότε προχωράμε σε επόμενο επίπεδο, δημιουργούμε το νέο scope και προσθέτουμε στον πάτερα το αντίστοιχο αποτύπωμα. Όταν φτάσουμε και στο πιο εσωτερικό τότε μετά την πλήρη μετάφραση αρχίζουμε να τα διαγραφούμε από μέσα προς τα έξω. Οι συναρτήσεις για την υλοποίηση των λειτουργιών του πινάκα είναι οι έξης:

- **newScope(name):** δημιουργεί το νέο scope και αυξάνει το επίπεδο που βρισκόμαστε.
- **delScope():** διαγραφεί το τελευταίο ολοκλήρωνο scope άλλα πριν απο αυτό το τυπώνει τον πίνακα και μειώνει το τρέχον επιπεδο. Τέλος προσθέτει στο αποτύπωμα του πάτερα το συνολικό frameLength.
- **searchByName(name):** κάνει αναζήτηση ενός entity από το τρέχον και προς τα έξω και επιστρέφει το πρωτο που βρηκε καθώς και το επίπεδο στο οποίο το βρήκε. Αν είναι υποπρογραμμα τότε θα πάρουμε το επιπεδο του γωνεα, ενω αν δεν το βρει έχουμε σφάλμα.
- **patchStart(nextQuad()):** ενημερώνει το αποτύπωμα στον πάτερα με την πρώτη τετράδα του scope begin_block (στον κωδηκα που παραδωθηκε δειχνει στην πρωτη εκτελεσημη εντολη)
- **newArg(par)** ενημερώνει το αποτύπωμα στον πάτερα με τους τύπους ορισμάτων του scope.
- **search(name, type):** κάνει αναζήτηση ενός entity από το τρέχον και προς τα έξω με βαση το type.
- **newEntity(e)**: προσθετει ενα entity στο τρεχον scope.

Σημασιολογική ανάλυση

Στο σημείο αυτό μπορούμε να εκμεταλλευτούμε τον πινάκα συμβολών για να κάνουμε επιπλέον έλεγχους για την ορθότητα του προγράμματος. Ποιο συγκεκριμένα θα εξασφαλίσουμε τα εξης:

- μια μεταβλητή που χρησιμοποιείται από ένα υποπρογραμμα είναι ορατή δηλαδή να έχει δηλωθεί η περαστει ως παραμτρος εντός εμβέλειας

- στο ίδιο επίπεδο δεν μπορούμε να έχουμε δηλώσει 2 μεταβλητές/παραμέτρους η 2 υποπρογραμματα η 1 υποπρογραμμα και μια μεταβλητή/παράμετρος με το ίδιο όνομα
- κάθε κλίση υποπρογραμματος να είναι εντός εμβέλειας, δηλαδή η κληθησα θα πρέπει να είναι ορατή στην καλούσα
- όταν έχουμε κλήση υποπρογραμματος οι τυπικές και οι παράμετροι κλήσης θα πρέπει να συμφωνούν
- κάθε συνάρτηση έχει τουλάχιστον ένα return και κάποιο return δεν μπορεί να βρίσκεται πουθενά άλλου

Τα παραπάνω υλοποιηθηκαν ως έξης:

- κάθε φορά που εντοπίζουμε ένα όνομα μεταβλητής είτε σε expression είτε αριστερά εντολής εκχώρησης είτε σε παραμέτρους καλούμε την συνάρτηση validatePars(lst), και κάνουμε search στον πινάκα να βρούμε κάποιο entity με τύπο μεταβλητής και όνομα της παραμέτρου.
- όλη η δουλειά γίνεται στην συνάρτηση new Entity(entity). Πριν την εισαγωγή του entity κοιτάμε στο scope αν υπάρχει entity με το ίδιο όνομα. Έπειτα διακρίνουμε τις περιπτώσεις υποπρογραμμα/μεταβλητή, μεταβλητή/μεταβλητή, υποπρογραμμα/υποπρογραμμα.
- στις συναρτήσεις idtail και call όταν εντοπίζουμε κλήσεις υποπρογραματος τότε καλούμε τις isFunc(name), isProc(name) και κάνουμε έλεγχο για το αν υπάρχει το αντίστοιχο υποπρογραμα εντός εμβέλειας.
- αυτό γίνεται στον τελικό κωδικά Αρχικά καθώς η subPars() διατρέχει τις τετράδες με τις παραμέτρους ενός υποπρογραμματος αποθηκεύουμε τον τύπο τους. Έπειτα κάνουμε εύρεση του αποτυπώματος του υποπρογραμματος στον πινάκα συμβολών και καλούμε την checkArgs() που ελέγχει αυτές τις 2 λίστες.
- καθε φορά πριν ξεκινήσουν τα statements ενός block θέτουμε στη σφαιρική μεταβλητή ret Check False. Έπειτα αν η statements() βρει return θέτει στη retCheck = (True, line). Μόλις τελειώσουν τα statements καλούμε την hasReturn(block). Στη μεταβλητή block έχουμε τον τύπο και όνομα του υποπρογραμματος. Η hasReturn με βάση αυτές τις πληροφορίες κάνει τους αντίστοιχους έλεγχους.

Φάση 4

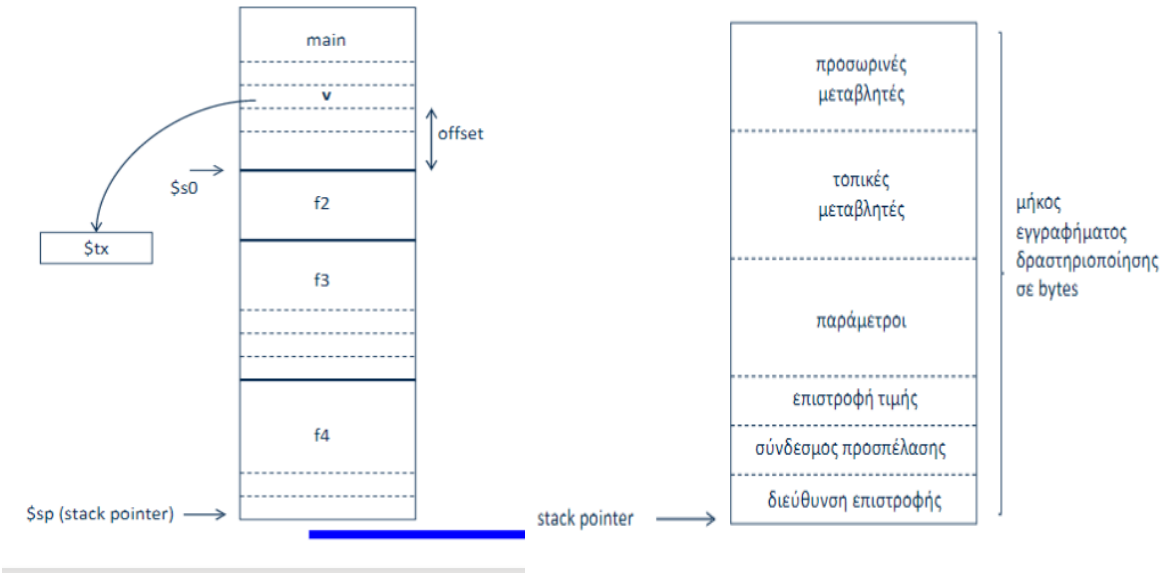
Εγγράφημα δραστηριοποίησης

Πριν προχωρήσουμε στην παράγωγη του τελικού κωδικά θα πρέπει να καθορίσουμε τον τρόπο με τον οποίο θα αποθηκεύσουμε τα δεδομένα για κάθε υποπρογραμμα. Επίσης θα πρέπει να είναι αποθηκευμένα με τέτοιο τρόπο ώστε να έχουμε εύκολη πρόσβαση σε διάφορες περιπτώσεις όπως πέρασμα παραμέτρων, επιστροφή τιμής, η και χρήση μη τοπικών μεταβλητών.

Τα τοπικά δεδομένα ενός υποπρογραμματος αποθηκεύονται κατά τη διάρκεια της εκτέλεσης του προγράμματος σε ένα τμήμα της μνήμης που ονομάζεται εγγράφημα δραστηριοποίησης. Το ΕΔ έχει θέσεις για την αποθήκευση των παραμέτρων, αποτελεσμάτων, τοπικών, σφαιρικών και προσωρινών μεταβλητών. Το μέγεθος και η μορφή του καθορίζεται από τις πληροφορίες για τα ονόματα που βρίσκονται στον πινάκα συμβολών. Έχουμε ένα ΕΔ για κάθε υποπρογραμμα το οποίο αποθηκεύεται στην στοίβα και ανανεώνεται σε κάθε κλήση. Για να προσπελάσουμε τη πληροφορία ενός entity στο ΕΔ άπλα περνούμε το offset από τον πινάκα συμβολών και αναφερόμαστε στη θέση [$sp - offset] της στοίβας, με το $sp να δείχνει στην αρχη του ΕΔ. Επίσης κάθε κελί του ΕΔ έχει μέγεθος 4 bytes.

Οι θέσεις στις οποίες θα αποθηκευτούν οι μεταβλητές του υποπρογαμματος στο ΕΔ καθορίζονται καθώς προσθέτουμε Entities σε ένα scope. Αν για παραδειγμα σε καποια στιγμη το μεγεθος ειναι 4Bx5 = 20 συνολικο μηκος του ΕΔ τοτε η επομενη μεταβλητη θα μπει στη επομενη θεση με offset = 24 απο την αρχη. Αρα στον πινακα η μεταβλιτη αυτη θα πρεπει να εχει offset = 24 και το αντιστοιχο scope θα εχει frameLength = 28 δηλαδη 28x4 Bytes. Βέβαια η αποθήκευση της πληροφορίας θα γίνει από τον τελικό κωδικά.

Σε κάθε ΕΔ πάντα δεσμεύουμε της πρώτες 3 αρχικές θέσεις. Στην 1 θέση αποθηκεύουμε την διεύθυνση επιστροφής την οποία έχει τοποθετήσει στον $ran jal, για να ξέρουμε που θα επιστρέψουμε. Στη δεύτερη θέση έχουμε τον σύνδεσμο προσπέλασης. Ο pointer αυτός δείχνει στην αρχή του Εδ του πάτερα της κλειθησας δηλαδή ποιο πάνω στη στοίβα. Έτσι αν χρειαστεί κάποια μεταβλητή που δεν είναι τοπική τον χρησιμοποιεί. Τέλος στην τρίτη θέση έχουμε την τιμή επιστροφής. Η θέση αυτή γεμίζει με έναν δείκτη πριν τη κλήση μιας συνάρτησης, ο οποίος θα δείχνει στη θέση της προσωρινής μεταβλητής που κρατάει την τιμή επιστροφής στο ΕΔ της καλούσας. Κατά την επιστροφή μια συνάρτηση χρησιμοποιεί αυτόν τον δείκτη για να βάλει την τιμή επιστροφής.



Παράγωγη τελικού κωδικά.

Έχοντας πλέον όλες τις πληροφορίες για κάθε scope του πηγαίου προγράμματος είμαστε σε θέση να παράξουμε τον αντίστοιχο τελικό κωδικά σε συμβολική γλώσσα, για κάθε τετράδα ενδιάμεσου κωδικά που το απαρτίζει.

Πριν διαγράψουμε το scope από τον πινάκα συμβολών καλούμε την συνάρτηση genInstructions() η οποία διατρέχει τις τετράδες του scope και ανάλογα με τον τύπο τετράδας παράγει τις εντολές τελικού κωδικά

Για να κάνουμε τη ζωή μας πιό εύκολη έχουμε ορίσει της έξης βοηθητικές συναρτήσεις:

- **gnvlcode(e):** μεταφέρει στον $t0 την διεύθυνση μιας μη τοπικής μεταβλητής. Ξεκινώντας από τον σύνδεσμο προσπέλασης του τρέχον scope ανεβαίνει τόσο επίπεδα όσα χρειαστούν για να φτάσει στο επίπεδο που ορίζεται η μεταβλητή, μέσω των συνδέσμων προσπέλασης. Έπειτα σε συνδυασμό με το offset βρίσκει τον pointer και τον φορτώνει στον $t0.
- **loadvr(v, r):** μεταφέρει δεδομένα στον καταχωριτη r. Ποιο συγκεκριμένα διακρίνει περιπτώσεις οπού η v είναι τοπική, καθολική, τυπική παράμετρος με αναφορά η τιμή, σταθερα η προσωρινή μεταβλητή καθώς και στο επίπεδο που βρίσκεται σε σχέση με το επίπεδο της τρέχουσας. Αν η τρέχουσα είναι σε διαφορετικό επίπεδο και η μεταβλητή είναι εκει παράμετρος η τοπική μεταβλητή τότε αναλαμβάνει δράση η gnvlcode(e).
- **storerv(r, v):** έχει παρόμοια λειτουργιά με την loadvr άλλα κάνει μεταφορά δεδομένων από τον καταχωρητη r στη μνήμη (μεταβλητή v).

Λεπτομέρειες παράγωγης τελικού κωδικά.

Η κύρια συνάρτηση που παράγει τελικό κωδικά για κάθε τετράδα ενός scope είναι η συνάρτηση quadsToFinal(). Διατρέχει τι τετράδες μια μια μέχρι να βρει την end_block και ανάλογα με την εντολή που εκτελεί καλεί την αντίστοιχη συνάρτηση για την παράγωγη, όπως και με τον λεκτικό αναλυτή. Ποιο συγκεκριμένα έχουμε τις έξης ρουτίνες:

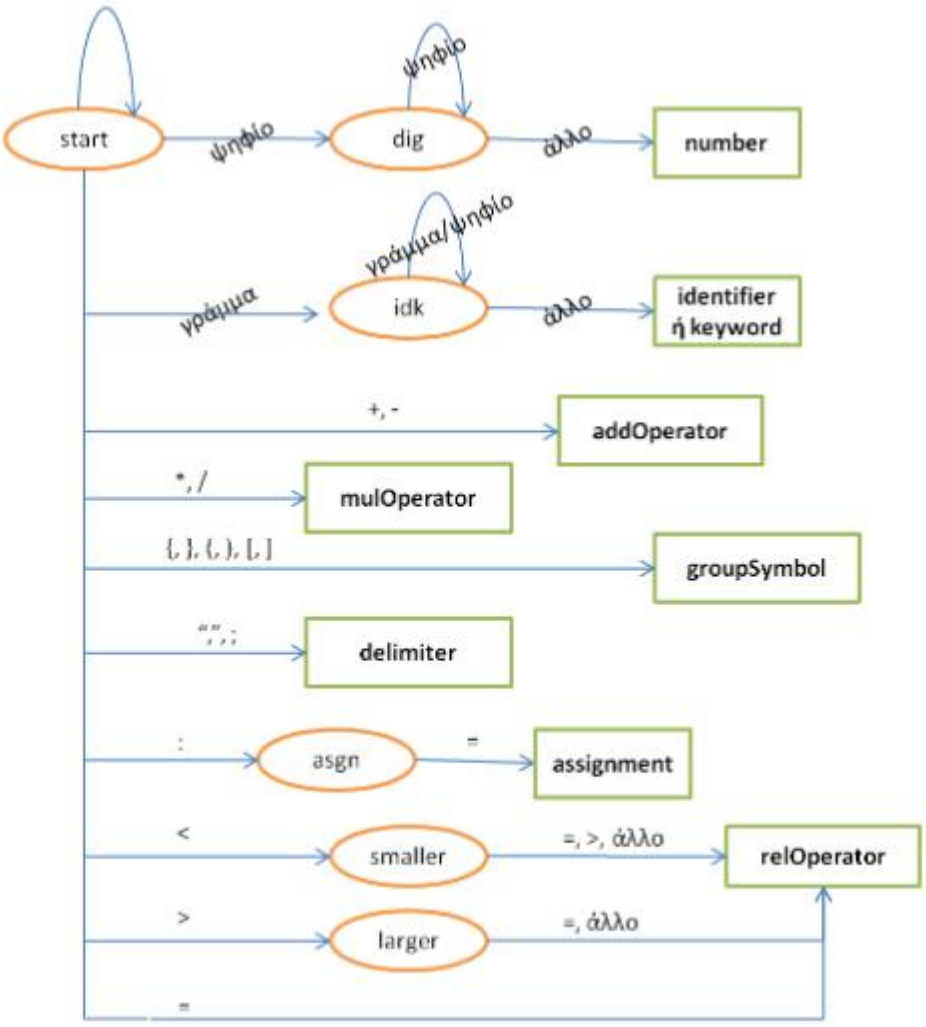
- **numExp(op, x, y, z):** φορτώνει στους t1, t2 τα χ, y εκτελεί την πράξη t1 = t1 + t2 και φορτώνει το αποτέλεσμα στη μνήμη (στο offset του z οπού και αν βρίσκεται)
- **assignV(x, z):** φορτώνει τα δεδομένα στον t1 και έπειτα τα αποθηκεύει στη μνήμη
- **relop(op, x, y, z):** φορτώνει τις τιμές των χ, y από τη μνήμη και παράγει την αντίστοιχη εντολή Άλματος.
- **retv(x):** αποθηκεύεται ο x στη διεύθυνση που είναι αποθηκευμένη στην 3η θέση του εγγραφήματος δραστηριοποίησης (επιστροφή τιμής). Έπειτα αφου έχουμε τέλος συνάρτησης τότε παίρνουμε από την πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της συνάρτησης και την βάζουμε πάλι στον $ra. Μέσω του $ra επιστρέφουμε στην καλούσα.
- **jump(l):** παράγουμε την αντίστοιχη εντολή Άλματος.
- **outP(x):** εκτελούμε syscall με επιλογή $v0 = 1 και εκτυπώνουμε την τιμή του $a0 που είναι η χ

- **inP(x):** εκτελούμε syscall με επιλογή $v0 = 5 και μεταφέρουμε το περιεχόμενο του $v0 στη μνήμη
- **subPars():** εδω χειριζομαστε το περασμα παραμετρων. Οσο βρίσκουμε τετράδες τύπου par τις διατρέχουμε και αρχίζουμε να γεμίζουμε τη στοίβα της κλειθισας με pointers η τιμές ανάλογα με τα αν η παράμετρος είναι cv h ref. Κάθε φορά τοποθετούμε κάτι στη στοίβα με βάση το $fp, που αρχικά τοποθετείται στην αρχή της στοίβας της κληθείσας. Βέβαια μια παράμετρος μπορεί να ορίζεται σε ανώτερο επίπεδο απο την καλουσα οποτε εδω θα θελαμε τη βοηθεια της gnvlcode. Αν βρούμε τετράδα τύπου RET τότε έχουμε κλήση συνάρτησης. Αυτό σημαίνει ότι περνούμε τον pointer της tempV από την καλούσα και τον περνάμε στην τρίτη θέση του εδ της κλειθησας. Έπειτα καλούμε το υποπρογραμμα.
- **callSub():** αρχικά θα πρέπει να βάλουμε στη δεύτερη θέση του εδ (-4($fp)) της κλειθησας την διεύθυνση του πάτερα για να μπορεί να προσπελάσει μεταβλητές που δεν της ανήκουν. Αν και οι δυο βρίσκονται στο ίδιο επίπεδο τότε έχουν το ίδιο πάτερα οπότε μεταφέρουμε στη θέση -4($fp) το σύνδεσμο προσπέλασης του πάτερα δηλαδή τον pointer στη θέση -4($sp). Αλλιώς άπλα περνάμε το $sp. Έπειτα κατεβάζουμε τον $sp στην αρχή της κληθείσας κάνουμε jal και μετά την επιστροφή ανεβάζουμε πάλι τον $sp στην αρχή της καλούσας.
- **callSubNoPars(name):** ίδια με την callSub άλλα θα κληθεί όταν έχουμε κλήση διαδικασίας χωρίς ορίσματα.
- **main():** καλείται κατά τη μετάφραση του κυρίως προγράμματος. Κατεβάζει τον $sp κατά framelength της main και σημειώνει στον $s0 το εγγράφημα δραστηριοποίησης της main ώστε να έχουμε εύκολη πρόσβαση στις global μεταβλητές.

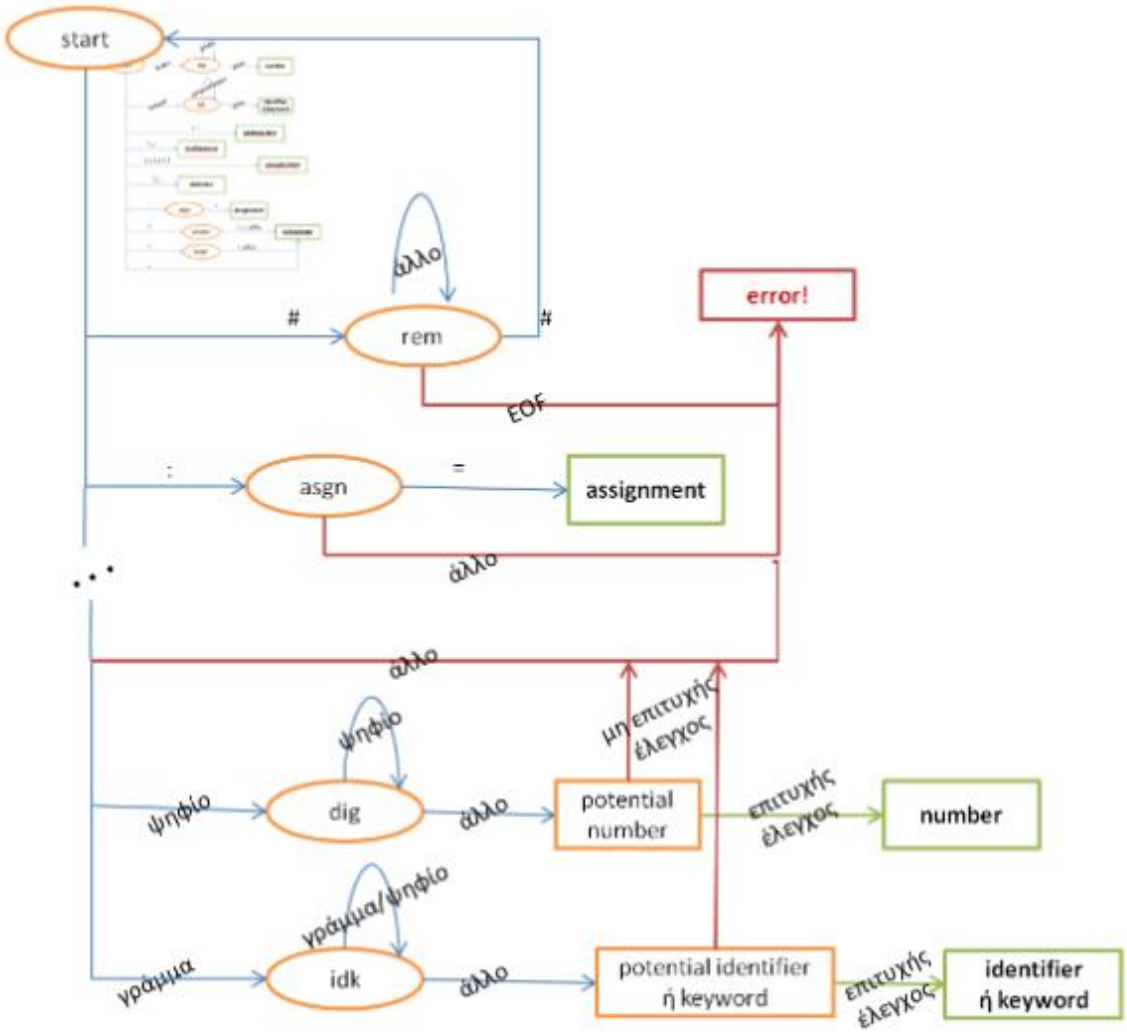
Επιπλέον στην αρχή κάθε συνάρτησης αποθηκεύουμε την διεύθυνση επιστροφής στη πρώτη θέση του εγγραφήματος δραστηριοποίησης της. Τέλος για να λειτουργήσουν όλα ομαλά προσθέτουμε εντολή Άλματος στη main ως αρχική εντολή του τελικού κωδικά. Στο σημείο αυτο η μεταφραση του πηγαιου προγράμματος εχει ολοκληρωθεί.

Παραρτημα

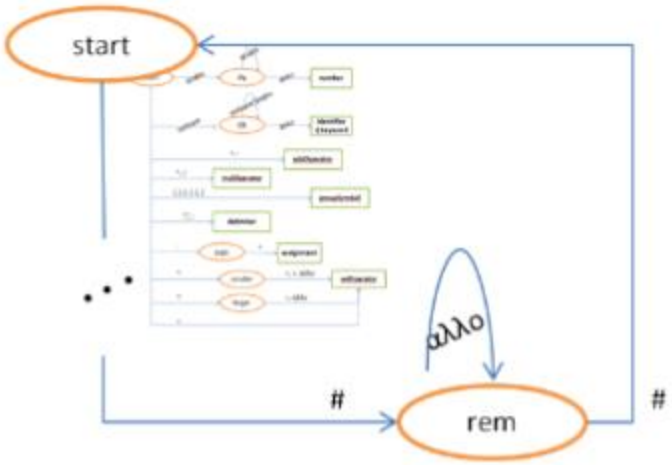
Αυτοματο Λειτουργιας



Σχήμα 2: Το αυτόματο λειτουργίας του λεκτικού αναλυτή

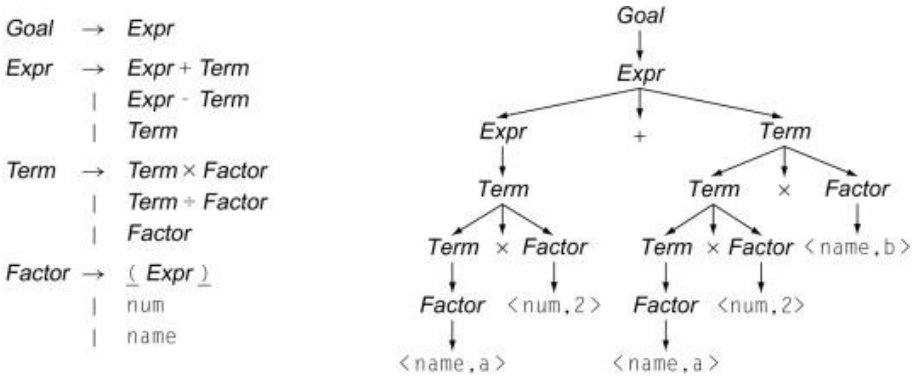


Σχήμα 4: Διαχείριση σφαλμάτων από τον λεκτικό αναλυτή



Σχήμα 3: Διαχείριση σχολίων από τον λεκτικό αναλυτή

Παραδειγμα Συντακτικου Δεντρου



Γραμματικη της Cimple

program : program ID block .
block : declarations subprograms statements
declarations : (declare varlist ;)∗
varlist : ID (, ID)∗ | ε
subprograms : (subprogram)∗
subprogram : function ID (formalparlist) block | procedure ID (formalparlist) block
formalparlist : formalparitem (, formalparitem)∗ | ε
formalparitem : in ID | inout ID
statements : statement ; | { statement (; statement)∗ }

statement : assignStat | ifStat | whileStat | switchcaseStat | forcaseStat | incaseStat | callStat | returnStat | inputStat | printStat | ε
assignStat : ID := expression
ifStat : if (condition) statements elsepart
elsepart : else statements | ε

whileStat : while (condition) statements
switchcaseStat: switchcase (case (condition) statements)∗ default statements
forcaseStat : forcase (case (condition) statements)∗ default statements
incaseStat : incase (case (condition) statements)∗
returnStat : return(expression)
callStat : call ID(actualparlist)
printStat : print(expression)
inputStat : input(ID)
actualparlist : actualparitem (, actualparitem)∗ | ε
actualparitem : in expression | inout ID
condition : boolterm (or boolterm)∗

boolterm : boolfactor (and boolfactor)∗
boolfactor : not [condition] | [condition] | expression REL_OP expression
expression : optionalSign term (ADD_OP term)∗
term : factor (MUL_OP factor)∗
factor : INTEGER | (expression) | ID idtail
idtail : (actualparlist) | ε
optionalSign : ADD_OP | ε
REL_OP : = | <= | >= | > | < | <> ; ADD_OP : + |
MUL_OP : * | /
INTEGER : [0-9]+
ID : [a-zA-Z][a-zA-Z0-9]*

Παραδειγμα Μεταφρασης

Πηγαιο προγραμμα

```
program blocks2
   declare x;
   procedure f1(in a)
      declare z;
      procedure f2()
         function f3(inout b)
```

```
            {
                return(a + x + b);
            }
        {
            x:=f3(inout x);
        }
    {
        z:=2;
        call f2();
    }
    {
        x:=1;
        call f1(in x);
        print(x);
    }.
```

Λεκτικες Μοναδες

{'tokenType': 'program_tk', 'tokenString': 'program', 'lineNo': 1}
{'tokenType': 'id_tk', 'tokenString': 'blocks2', 'lineNo': 2}
{'tokenType': 'declare_tk', 'tokenString': 'declare', 'lineNo': 2}
{'tokenType': 'id_tk', 'tokenString': 'x', 'lineNo': 2}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 3}
{'tokenType': 'procedure_tk', 'tokenString': 'procedure', 'lineNo': 3}
{'tokenType': 'id_tk', 'tokenString': 'f1', 'lineNo': 3}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 3}
{'tokenType': 'in_tk', 'tokenString': 'in', 'lineNo': 3}
{'tokenType': 'id_tk', 'tokenString': 'a', 'lineNo': 3}
{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 4}
{'tokenType': 'declare_tk', 'tokenString': 'declare', 'lineNo': 4}
{'tokenType': 'id_tk', 'tokenString': 'z', 'lineNo': 4}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 5}
{'tokenType': 'procedure_tk', 'tokenString': 'procedure', 'lineNo': 5}
{'tokenType': 'id_tk', 'tokenString': 'f2', 'lineNo': 5}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 5}
{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 6}
{'tokenType': 'function_tk', 'tokenString': 'function', 'lineNo': 6}
{'tokenType': 'id_tk', 'tokenString': 'f3', 'lineNo': 6}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 6}
{'tokenType': 'inout_tk', 'tokenString': 'inout', 'lineNo': 6}
{'tokenType': 'id_tk', 'tokenString': 'b', 'lineNo': 6}
{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 7}

{'tokenType': 'lbrace_tk', 'tokenString': '{', 'lineNo': 8}
{'tokenType': 'return_tk', 'tokenString': 'return', 'lineNo': 8}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 8}
{'tokenType': 'id_tk', 'tokenString': 'a', 'lineNo': 8}
{'tokenType': 'plus_tk', 'tokenString': '+', 'lineNo': 8}
{'tokenType': 'id_tk', 'tokenString': 'x', 'lineNo': 8}
{'tokenType': 'plus_tk', 'tokenString': '+', 'lineNo': 8}
{'tokenType': 'id_tk', 'tokenString': 'b', 'lineNo': 8}
{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 8}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 9}
{'tokenType': 'rbrace_tk', 'tokenString': '}', 'lineNo': 10}
{'tokenType': 'lbrace_tk', 'tokenString': '{', 'lineNo': 11}
{'tokenType': 'id_tk', 'tokenString': 'x', 'lineNo': 11}
{'tokenType': 'assign_tk', 'tokenString': ':=', 'lineNo': 11}
{'tokenType': 'id_tk', 'tokenString': 'f3', 'lineNo': 11}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 11}
{'tokenType': 'inout_tk', 'tokenString': 'inout', 'lineNo': 11}
{'tokenType': 'id_tk', 'tokenString': 'x', 'lineNo': 11}
{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 11}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 12}
{'tokenType': 'rbrace_tk', 'tokenString': '}', 'lineNo': 13}
{'tokenType': 'lbrace_tk', 'tokenString': '{', 'lineNo': 14}
{'tokenType': 'id_tk', 'tokenString': 'z', 'lineNo': 14}
{'tokenType': 'assign_tk', 'tokenString': ':=', 'lineNo': 14}
{'tokenType': 'int_tk', 'tokenString': '2', 'lineNo': 14}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 15}
{'tokenType': 'call_tk', 'tokenString': 'call', 'lineNo': 15}
{'tokenType': 'id_tk', 'tokenString': 'f2', 'lineNo': 15}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 15}
{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 15}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 16}
{'tokenType': 'rbrace_tk', 'tokenString': '}', 'lineNo': 17}
{'tokenType': 'lbrace_tk', 'tokenString': '{', 'lineNo': 18}
{'tokenType': 'id_tk', 'tokenString': 'x', 'lineNo': 18}
{'tokenType': 'assign_tk', 'tokenString': ':=', 'lineNo': 18}
{'tokenType': 'int_tk', 'tokenString': '1', 'lineNo': 18}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 19}
{'tokenType': 'call_tk', 'tokenString': 'call', 'lineNo': 19}
{'tokenType': 'id_tk', 'tokenString': 'f1', 'lineNo': 19}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 19}
{'tokenType': 'in_tk', 'tokenString': 'in', 'lineNo': 19}
{'tokenType': 'id_tk', 'tokenString': 'x', 'lineNo': 19}

{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 19}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 20}
{'tokenType': 'print_tk', 'tokenString': 'print', 'lineNo': 20}
{'tokenType': 'lpar_tk', 'tokenString': '(', 'lineNo': 20}
{'tokenType': 'id_tk', 'tokenString': 'x', 'lineNo': 20}
{'tokenType': 'rpar_tk', 'tokenString': ')', 'lineNo': 20}
{'tokenType': 'semcol_tk', 'tokenString': ';', 'lineNo': 21}
{'tokenType': 'rbrace_tk', 'tokenString': '}', 'lineNo': 21}
{'tokenType': 'end_tk', 'tokenString': '.', 'lineNo': 21}

Ενδιάμεσος Κώδικας

0: begin_block , f3 , _ , _
1: + , a , x , T_0
2: + , T_0 , b , T_1
3: retv , T_1 , _ , _
4: end_block , f3 , _ , _
5: begin_block , f2 , _ , _
6: par , x , ref , _
7: par , T_2 , RET , _
8: call , _ , _ , f3
9: := , T_2 , _ , x
10: end_block , f2 , _ , _
11: begin_block , f1 , _ , _
12: := , 2 , _ , z
13: call , _ , _ , f2
14: end_block , f1 , _ , _
15: begin_block , blocks2 , _ , _
16: := , 1 , _ , x
17: par , x , cv , _
18: call , _ , _ , f1
19: out , x , _ , _
20: halt , _ , _ , _
21: end_block , blocks2 , _ , _

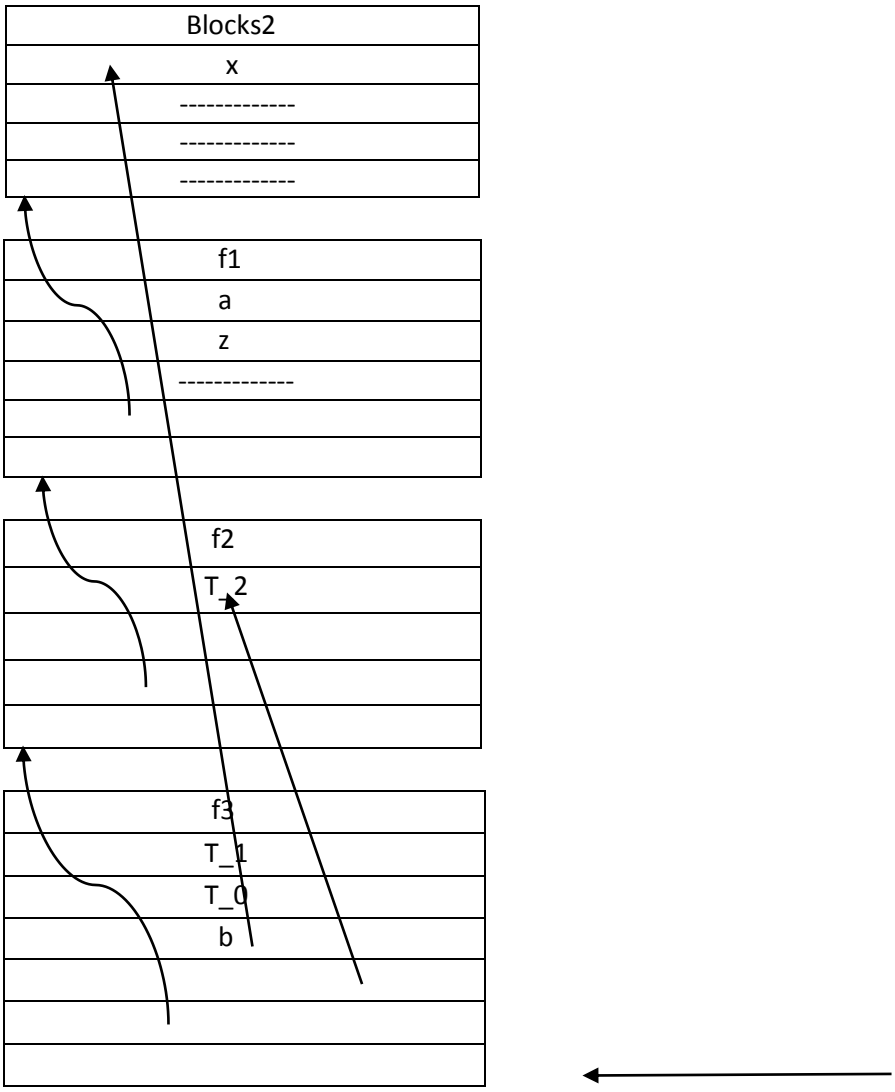
Πινακας συμβολων και ΕΔ για καθε υποπρογραμμα

{'name': 'b', 'offset': 12, 'parMode': 'ref', 'type': 'par'}
{'name': 'T_0', 'offset': 16, 'type': 'temp'}
{'name': 'T_1', 'offset': 20, 'type': 'temp'}
scope name: f3
scope fLength: 24
scope level: 3
==========

{'name': 'f3', 'type': 'function', 'args': ['ref'], 'sQuad': 0, 'fLength': 24}
{'name': 'T_2', 'offset': 12, 'type': 'temp'}
scope name: f2
scope fLength: 16
scope level: 2
==========

{'name': 'a', 'offset': 12, 'parMode': 'cv', 'type': 'par'}
{'name': 'z', 'type': 'var', 'offset': 16}
{'name': 'f2', 'type': 'procedure', 'args': [], 'sQuad': 5, 'fLength': 16}
scope name: f1
scope fLength: 20
scope level: 1
==========

{'name': 'x', 'type': 'global', 'offset': 12}
{'name': 'f1', 'type': 'procedure', 'args': ['cv'], 'sQuad': 11, 'fLength': 20}
scope name: blocks2
scope fLength: 16
scope level: 0



Τελικος Κωδηκας

```
Lbegin:
 j Lmain
L0:     #[0, 'begin_block', 'f3', '_', '_']#
 sw $ra, ($sp)
L1:     #[1, '+', 'a', 'x', 'T_0']#
 lw $t0,-4($sp)    !!!gnvl looks for a
 lw $t0,-4($t0)    !!!climbs to f1
 addi $t0,$t0,-12 !!!finds the cv a param
 lw $t1, ($t0)      !!! gets the value
 lw $t2, -12($s0)
 add $t1, $t1, $t2
 sw $t1, -16($sp)
L2:     #[2, '+', 'T_0', 'b', 'T_1']#
 lw $t1, -16($sp)
 lw $t0, -12($sp)
 lw $t2, ($t0)
 add $t1, $t1, $t2
 sw $t1, -20($sp)
L3:     #[3, 'retv', 'T_1', '_', '_']#
 lw $t1, -20($sp)
 lw $t0, -8($sp)
 sw $t1, ($t0)
 lw $ra, ($sp)
 jr $ra
L4:    #[4, 'end_block', 'f3', '_', '_']#
 lw $ra, ($sp)
 jr $ra
L5:     #[5, 'begin_block', 'f2', '_', '_']#
 sw $ra, ($sp)
L6:     #[6, 'par', 'x', 'ref', '_']#
 addi $fp, $sp, 24
 lw $t0,-4($sp)        !!!gnvlcode looks for x
 lw $t0,-4($t0)
 addi $t0,$t0,-12     !!!reaches the global x
 sw $t0, -12($fp)     !!!stores the address
L7:     #[7, 'par', 'T_2', 'RET', '_']#
 addi $t0, $sp, -12
 sw $t0, -8($fp)
L8:     #[8, 'call', '_', '_', 'f3']#
 sw $sp, -4($fp)
 addi $sp, $sp, 24
 jal L0
 addi $sp, $sp, -24
L9:     #[9, ':=', 'T_2', '_', 'x']#
 lw $t1, -12($sp)
 sw $t1, -12($s0)
L10:    #[10, 'end_block', 'f2', '_', '_']#
 lw $ra, ($sp)
 jr $ra
L11:    #[11, 'begin_block', 'f1', '_', '_']#
 sw $ra, ($sp)
L12:    #[12, ':=', '2', '_', 'z']#
 li $t1, 2
 sw $t1, -16($sp)
L13:    #[13, 'call', '_', '_', 'f2']#
 addi $fp, $sp, 16
 sw $sp, -4($fp)
 addi $sp, $sp, 16
 jal L5
 addi $sp, $sp, -16
L14:    #[14, 'end_block', 'f1', '_', '_']#
 lw $ra, ($sp)
 jr $ra
Lmain:    #MAIN#
 addi $sp, $sp, 16
 move $s0, $sp
L15:    #[16, ':=', '1', '_', 'x']#
 li $t1, 1
 sw $t1, -12($s0)
L16:    #[17, 'par', 'x', 'cv', '_']#
 addi $fp, $sp, 20
 lw $t0, -12($s0)
 sw $t0, -12($fp)
L17:    #[18, 'call', '_', '_', 'f1']#
```

```
sw $sp, -4($fp)
addi $sp, $sp, 20
jal L11
addi $sp, $sp, -20
```

```
L18:      #[19, 'out', 'x', '_', '_']#
  li $v0, 1
  lw $a0, -12($s0)
  syscall
```