

<Μεταφραστές>

Φτιάχνοντας Mini-Pascal Compiler

<28/05/2014 Πανεπιστημιο Ιωαννίνων Τμήμα ΜΗΥ & Πληροφορικής>

Ομάδα εργασίας

<Νικόλαος Ζώης AM: 2054 & Μαρίνα Μπουρδάκη AM: 2111>

Περιεχόμενα

1. Εισαγωγή.....	3
1.1 Λίγα λόγια.....	3
1.2 Αντικείμενο Αναφοράς.....	3
1.3 Ερωτήματα που προκύπτουν.....	3
1.3 Θα επικεντρωθούμε στην δημιουργία.....	3
1.4 Τι θα χρησιμοποιήσουμε;.....	4
1.5 Απαιτήσεις από έναν Μεταφραστή.....	4
1.6 Βήματα υλοποίησης Μεταφραστή.....	5
2. Λεκτικός αναλυτής.....	5
2.1 Στόχος.....	5
2.2 Μεθόδοι και πεδία της κλάσης LexicalAnalyzer.....	6
2.3 parseToken() - Μετατροπής Αυτοματού NFA σε C++.....	7
2.3 getTokenId() - Υπολογισμός λετκικού κωδικού(TOKEN).....	8
3. Συντακτικός Αναλυτής.....	8
3.1 Στόχος.....	8
3.2 Υλοποίηση γραμματικής LL1 σε κώδικα C++.....	10
3.3 Μεθόδοι και Πεδία της κλάσης SyntaxAnalyzer.....	12
3.4 Ιδιαιτερότητες της C++.....	14
4. Ενδιαμεσος κώδικας.....	15
4.1 Στόχος.....	15
4.2 Μεθόδοι και πεδία του Ενδιάμεσου κώδικα.....	16
4.2.1 Κλάση LabelQuad (Βήμα).....	17
4.2.2 Κλάση LQList.....	17
4.3 Μετατροπή δομών σε <<Κώδικα Βημάτων>>.....	18
5.1 Στόχος.....	22
5.2 Μέθοδοι & Πεδία της κλάσης Entity.....	23
5.3 Μεθόδοι & Πεδία της Κλάσης Scope.....	24
5.5 Μέθοδοι & Πεδία του Πίνακα Συμβόλων στην κλάση SyntaxAnalyzer.....	25
6.1 Στόχος.....	27
6.2 Εισαγωγή στον Metasim.....	27
6.3 Εγγράφημα δραστηριοποιησης.....	28
6.4 Εντολές Metasim.....	29
6.5 Απο τετράδες σε τελικό κώδικα.....	30
6.5.1 Η συνάρτηση gnlvcode().....	30
6.5.2 Η συνάρτηση load(v,r).....	31
6.5.2 Η Συνάρτηση storev(r,v).....	31
6.6 Η κλάση EndCodeEngine.....	31
6.6.1 Μεθόδοι & Πεδία της κλάσης.....	32

1. Εισαγωγή

1.1 Λίγα λόγια

Πάντα αναρατιωμούν πως μπορώ να φτίαξω μια δική μου γλώσσα προγραμματισμού. Άλλα και πως ο υπολογίστης ή το κινητό μου καταλαβαίνουν την φυσική μας γλώσσα. Ευκαιρία απαντησης σε αυτό το ερώτημα έθεσε το μάθημα των Μεταφραστών ΠΛΥ602 στο Τμήμα ΜΗΥ & Πληροφορικής Ιωαννίνων.

1.2 Αντικείμενο Αναφοράς

Αντό που θα κάνουμε εμείς λοιπόν είναι να φτιάξουμε μια δική μας γλώσσα προγραμματισμού με κάποια standars που μας ικανοποιούν. Άλλα και πως θα την κάνουμε μετάφραση ώστε το μηχάνημα μας να την καταλάβει.

1.3 Ερωτήματα που προκύπτουν

Τι θέλουμε η προγραμματιστική μας γλώσσα να ικανοποιει : (for,while,..)

Σε ποια γλώσσα προγραμματισμού θα φτιάξουμε τον μεταφραστή μας ;

Ποια θα είναι η δομή του προγράμματος -μεταφραστή μας;

Πως θα κάνω το μηχανημά μου να καταλαβαίνει την καινουρια μας γλώσσα ;

Ποιες είναι οι απαιτήσεις μας από έναν Μεταφραστή ;

1.3 Θα επικεντρωθούμε στην δημιουργία

... της γλώσσας MiniPascal.

...του Λεκτικού Αναλυτή (parser).

...του Συντακτικού Αναλυτή.

...του Ενδιαμεσου Κώδικα.

...του Τελικού Κωδικα (Assembly).

...της Τελικής εφαρμογής.

1.4 Τι θα χρησημοποιήσουμε;

Την γλώσσα C++ για την υλοποιηση των μεθόδων του μεταφραστη.

Την κλασσική μέθοδο parsing με βάση ένα NFA ,ώστε να επιστρεψουμε μία αποδεκτή λέξη για την γλώσσα μας.

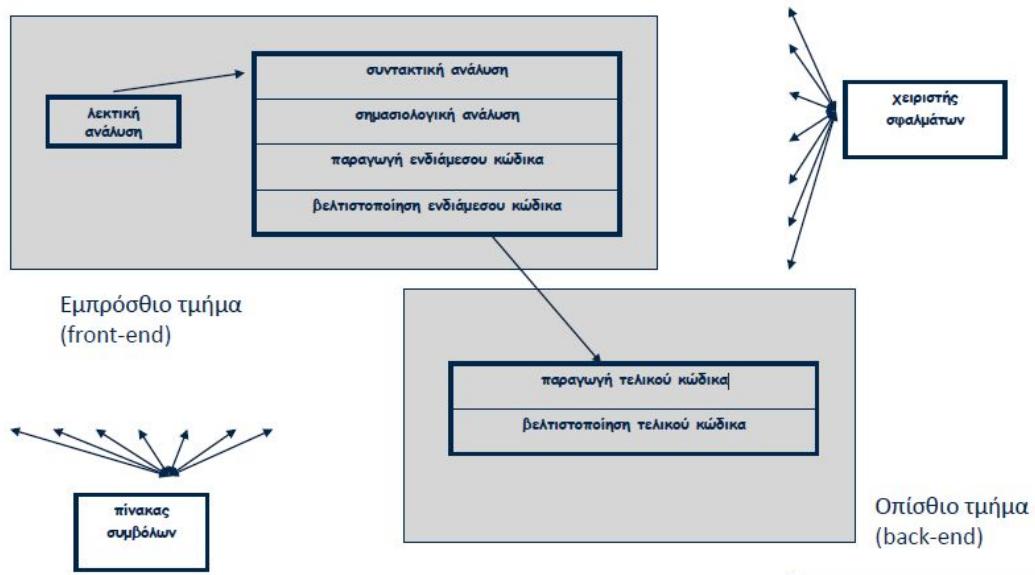
Γραμματική χωρίς συμφραζόμενα (Chomsky) για την αναπτυξη του συντακτικού μας δένδρου.

1.5 Απαιτήσεις από έναν Μεταφραστή

- Σωστή λειτουργία
- Να συμμορφώνεται με τις προδιαγραφές αρχικής και τελικής γλώσσας
- Να μεταφράζει προγράμματα αυθαίρετου μεγάλου μήκους
- Να παράγει αποδοτικό κώδικα
- Να έχει μικρό χρόνο εκτέλεσης
- Να έχει μικρές απαιτήσεις μνήμης κατά τη μεταγλώττιση
- Να δίνει καλά διαγνωστικά μηνύματα
- Να έχει τη δυνατότητα συνέχισης ύστερα από τον εντοπισμό σφαλμάτων
- Να είναι μεταφέρσιμος

1.6 Βήματα υλοποίησης Μεταφραστή

Οργάνωση Μεταγλωττιστή

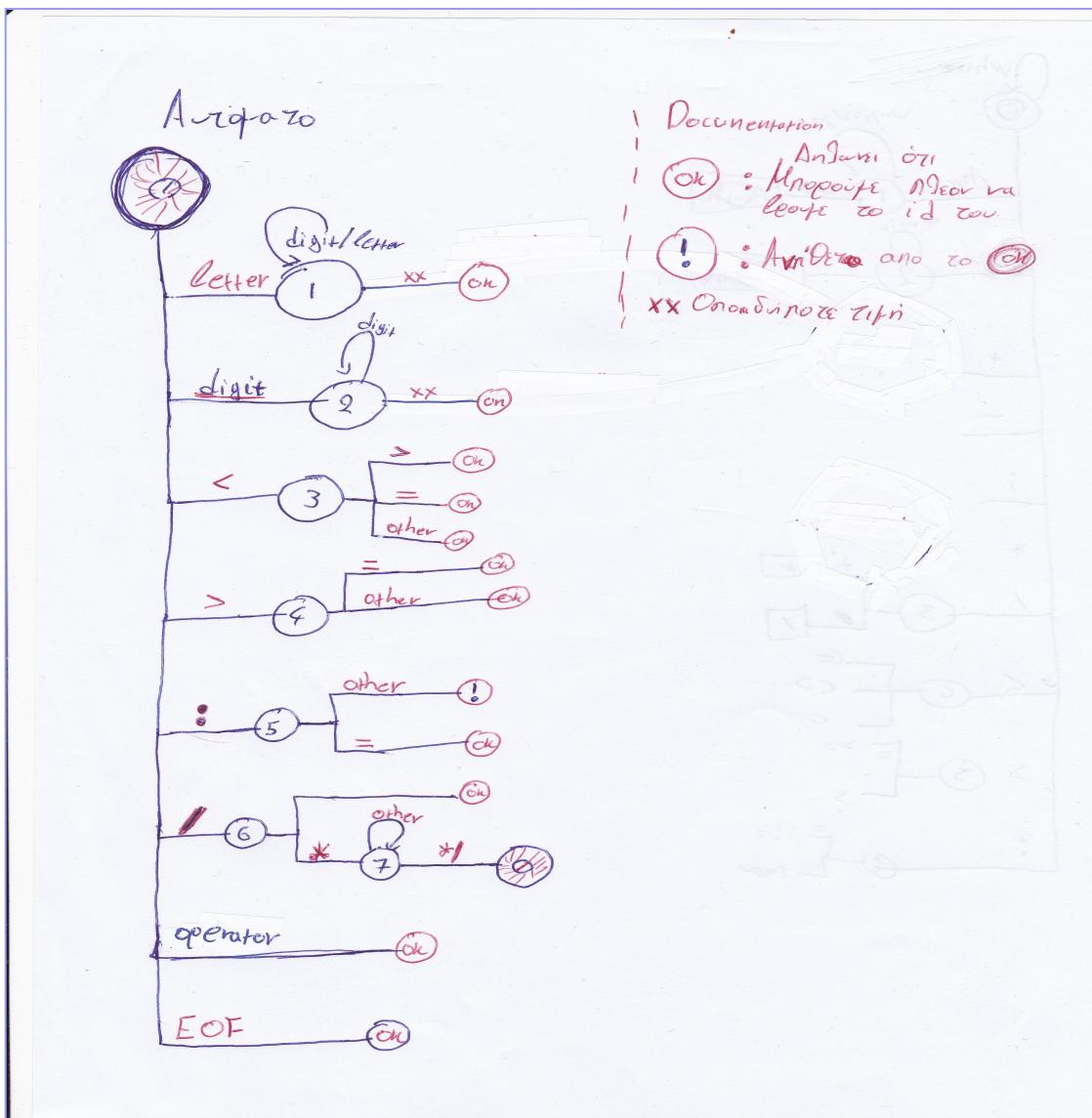


Εικόνα από τις σημειώσεις του καθηγητη.

2. Λεκτικός αναλυτής

2.1 Στόχος

Η εργασία του λεκτικού αναλυτη ονομάζεται παραγωγή μιας λεκτικής μονάδας στο αρχείο πηγηαίου κώδικα με βάση το αυτόματο της Εικ 2.1 και η επιστορή ενός ακαίρεου ειδικού κωδικού (TOKEN) που χαρακτηρίζει την λεκτική μονάδα που βρήκαμε.



Εικ.2.1: Το αυτόματο NFA

2.2 Μεθοδοι και πεδία της κλάσης *LexicalAnalyzer*

private:

```

fstream* fileInput;    (Πηγηαίο αρχειό προγραμματος MiniPascal)
string lexis;          (Λεκτική μονάδα που βρήκαμε)
stringstream errors;   (Διαγνωστικά λάθη)
bool parseToken();
```

int getTokenID(); (Υπολογίζει το TOKEN της λεκτικής μονάδας)

public:

(Κανει parse στο αρχείο και επιστρέφει το TOKEN της λεκτικής μονάδας)

int getToken();

(Επιστρεφει το πεδίο lexis)

string getLexis();

int line; (Κραταει την γραμμή που βρισκόμαστε τώρα στο πηγηαίο αρχειο.

κατα τη διαρκεια του parsing)

2.3 *parseToken()* - Μετατροπής Αυτοματού NFA σε C++

Βοηθητικές μεταβλητές:

int state: αποθηκεύει προσωρινά σε ποια κατασταση βρισκόμαστε κατα τη διαρκεια του parcing ωστε να μας βοηθήσει αργότερα να αποφανθούμε σε ποιά κατασταση θα μεταβούμε.

char c: κρατάει τον χαρακτήρα που διαβασαμε απο το αρχείο.

char temp: κραταει τον χαρακτήρα που βρίσκετε αμέσως μετά τον c χωρίς να τον εξάγει.

*Για κάθε κατάσταση του αυτοματου και για κάθε λεκτική μονάδα έχουμε δηλώσει ένα ειδικό κωδικό που τα χαρακτηρίζει.

(Αρχειο token.h enum states , enum tokens).

Λειτουργία:

Γεμίζει το πεδίο lexis με την έγκυρη λεκτική μονάδα που βρήκαμε.

1. Αρχικά μεταβένουμε στην αρχική κατάσταση, βάζουμε το state = state0.
2. Γεμίζουμε το πεδίο c με τον χαρακτήρα που διαβασαμε και εξάγαμε απο το αρχειο (μεδοδος **getChar()**)
3. Γεμίζουμε το temp με τον διπλανό χαρακτήρα χώρις να τον εξάγουμε απο το αρχειο (μεθοδος **peekChar()**)
4. Επειτα αναλογα με την πληροφορίες που έχουμε μεταβαίνουμε στην κατάσταση που θέλουμε μέχρι να φτάσουμε σε μία τελική κατασταση εξόδου
5. Η μετάβαση γίνεται με την δομή **if else**.

Κομάτι κώδικα απο το αρχειο LexicalAnalyzer.cpp method parseToken():

```
State=state0;
C= 'a';
If( (isLetter(c) || isDigit(c) )&& state==state0){
    State=state1;
    Lexis+=c;
}
```

Όντως αν κοιτάξουμε το αυτοματό μας (εικόνα 2.1)

Οταν είμαστε στην κατασταση 0 και

διαβάσουμε γράμμα τοτε θα μεταβούμε στην κατασταση 1. Δηλαδή το state μας θα γίνει state = state1;

2.3 *gettokenId()* - Υπολογισμός λετκικού κωδικού(TOKEN)

Οπως αναφέραμε και πριν για κάθε λεκτική μονάδα υπάρχει και ενας κωδικός
(Αρχείο tokens.h)

Λειτουργία:

Αυτο που κάνουμε είναι απλα να ελέγχουμε αν η λεκτική μοναδα μας είναι μία ειδικής σημασιας λέξη ή ένα απλο ID Ή Number και επιστρέφουμε τον ανάλογο κωδικό.

Κομμάτι ψευδο-κώδικα απο το αρχείο LexicalAnalyzer.cpp method gettokenId():

Lexis == "program" //Εστω οτι έχει αυτην την τιμή

```
If(lexis == "program"){
    Return tokens->PROGRAM;
}
Else if(lexis == "function"){
    .
    .
}
```

}else if{.....}....else...

Άν έοχουμε λοιπόν διαβασει την ειδική λέξη program τότε πρέπει να επιστρέψουμε το αντιστοιχο token ωστε αργότερα ο συντακτικός αναλυτής να το διαχειριστεί ανάλογα.

3. Συντακτικός Αναλυτής

3.1 Στόχος

Οπως έιπαμε και πριν καθε τι έχει και μια δουλειά να κάνει, ωστε να καταφέρουμε απο το πηγηαιο αρχείο να φτάσουμε στο τελίκο και αναγνωσιμο απο τον υπολογιστη εκτελεσιμο αρχείο π.χ *.exe.

Έτσι λοιπόν και ο συντακτικός αναλυτής παίζει ένα απο τον σημαντικό ρόλο στην κατασκεύη του μεταφραστή μας, αποτελόντας τη γραμματική και τον <<σκελετο>> της γλώσσας μας.

Εμείς θα υλοποιήσουμε την **αναδρομική κατάβαση** με την γραμματική τύπου (**LL1**).

Λειτουργία

- Ελέγχει αν το πηγιαίο αρχείο ανήκει ή όχι στην γλώσσα μας.
- Δημιουργεί το κατάλληλο περιβάλλον ώστε αργότερα να μπουν κομμάτια κώδικα που μας παράγουν το τελίκο κώδικα.

3.2 Υλοποίηση γραμματικής LL(1) σε κώδικα C++

Η γραμματική LL(1) αναγνωρίζει από αριστερά στα δεξιά, την αριστερότερη δυνατή παραγωγή και όταν βρίσκεται σε δίλλημα ποιον κανόνα να ακολουθησει τότε αρκεί να κοιτάξει το αμεσως επόμενο συμβόλο στην συμβολοσειρά εισόδου.

Πανεπιστήμιο Ιανουάριον Τμήμα Πληροφορικής Επίκουρη Καθηγήστρια Διδάσκων: Γ. Μανής Φεβρουάριος 2014		Πανεπιστήμιο Ιανουάριον Τμήμα Πληροφορικής Επίκουρη Καθηγήστρια Διδάσκων: Γ. Μανής Φεβρουάριος 2014	
Γραμματική της mini-Pascal			
<PROGRAM>	::= program ID <PROGRAMBLOCK>	<PROCORFUNC>	::= procedure ID <PROCORFUNCBODY> function ID <PROCORFUNCBODY>
<PROGRAMBLOCK>	::= <DECLARATIONS> <SUBPROGRAMS> <BLOCK>	<FORMALPARS>	::= <FORMALPARLIST> ϵ
<BLOCK>	::= begin <SEQUENCE> end	<FORMALPARLIST>	::= <FORMALPARITEM> (, <FORMALPARITEM>)*
<DECLARATIONS>	::= (<CONSTDECL>)* (<VARDECL>)*	<FORMALPARITEM>	::= ID var ID
<CONSTDECL>	::= const <ASSIGNLIST> ; ϵ	<SEQUENCE>	::= <STATEMENT> (; <STATEMENT>)*
<ASSIGNLIST>	::= <ASSIGNCONST> (, <ASSIGNCONST>)*	<STATEMENT>	::= <ASSIGNMENT-STAT> <IF-STAT> <WHILE-STAT> <FOR-STAT> <CALL-STAT> <PRINT-STAT> <INPUT-STAT> <RETURN-STAT>
<ASSIGNCONST>	::= ID := CONSTANT	<ASSIGNMENT-STAT>	::= ID := <EXPRESSION>
<VARDECL>	::= var <VARLIST> ; ϵ	<IF-STAT>	::= if <CONDITION> then <BLOCK-OR-STAT> <ELSEPART>
<VARLIST>	::= ID (, ID)*	<ELSEPART>	::= ϵ else <BLOCK-OR-STAT>
<SUBPROGRAMS>	::= (<PROCORFUNC>) *		
<WHILE-STAT>	::= while <CONDITION> do <BLOCK-OR-STAT>	<RELATIONAL-OPR>	::= == < > >= <= <>
<FOR-STAT>	::= for <ASSIGNMENT-STAT> to <EXPRESSION> <STEP-PART> <BLOCK-OR-STAT>	<EXPRESSION>	::= <OPTIONAL-SIGN> <TERM> (<ADD-OPR> <TERM>)*
<STEP-PART>	::= step <EXPRESSION> ϵ	<TERM>	::= <FACTOR> (<MUL-OPR> <FACTOR>)*
<CALL-STAT>	::= call ID <ACTUALPARS>	<FACTOR>	::= CONSTANT (<EXPRESSION>) ID <OPTACTUALPARS>
<PRINT-STAT>	::= print (<EXPRESSION>)	<OPTACTUALPARS>	::= <ACTUALPARS> ϵ
<INPUT-STAT>	::= input (ID)	<ADD-OPR>	::= + -
<RETURN-STAT>	::= return (<EXPRESSION>)	<MUL-OPR>	::= * /
<ACTUALPARS>	::= (<ACTUALPARLIST> ϵ)	<OPTIONAL-SIGN>	::= ϵ <ADD-OPR>
<ACTUALPARLIST>	::= <ACTUALPARITEM> (, <ACTUALPARITEM>)*		
<ACTUALPARITEM>	::= <EXPRESSION> var ID		
<CONDITION>	::= <BOOLTERM> (or <BOOLTERM>)*		
<BOOLTERM>	::= <BOOLFACTOR> (and <BOOLFACTOR>)*		
<BOOLFACTOR>	::= not [<CONDITION>] [<CONDITION>] <EXPRESSION> <RELATIONAL-OPR> <EXPRESSION>		

Εικ. 3.1: Γραμματική της MiniPascal

Έτσι λοιπόν ψευτοαλγορίθμικά κάνουμε το εξής για να μετατρέψουμε την Γραμματική μας σε κώδικα.(Εικόνα 3.1)

Βήμα 1: Για κάθε κανόνα αριστερά φτιαχνούμε και μία συνάρτηση/μέθοδο με το όνομα του κανόνα.

Βήμα 2: Ζητάμε από τον λεκτικό αναλυτή να μας επιστέψει την λεκτική μονάδα που βρήκε.

Βήμα 2: Όταν συναντάμε ένα μη τερματικό συμβολο εκτελόμε την αντιστοιχη συνάρτηση/μέθοδο.

Βήμα 3: Αν συναντήσουμε ένα τερματικό σύμβολο, ελέγχουμε αν ταυτίζεται με την λεκτική μονάδα που υπολογίσαμε στο Βήμα 2, και εκτελούμε την μέθοδο/συνάρτηση αλλιώς σταματάμε και τυπώνουμε το κατάλληλο μηνυμα λάθους.

Βήμα 4: Όταν αναγνωρισθεί και η τελευταία λεκτική μονάδα τότε το πιγηαίο αρχείο μας ανήκει στην γλώσσα μας και μπορούμε να προχωρήσουμε περεταίρω.

Παράδειγμα 3.1

Έστω ο κανόνας από την Εικ. 3.1 <PROGRAM> ::= program ID <PROGRAM_BLOCK>

Υποθέτοντας ότι ήδη έχουμε υλοποιησει-δηλώσει το <PROGRAM_BLOCK> σε μέθοδο programBlock() θα πάρουμε το εξής:

```
Void SyntaxAnalyzer::program(){
    //Βήμα 1
    cuToken = lex->getToken();
    //Βήμα 2
    If(cuToken == PROGRAM_TOKEN_ID){
        //Βήμα 1
        cuToken = lex->getToken();
        //Βήμα 2
        If(cuToken == ID_TOKEN_ID){
            //Βήμα 3
            programBlock();
        }
        else{
            Cout << "exepted program-name-id after program word" << endl; exit();
        }
    }
    else{
        Cout << "expected program word << endl; exit(); } }
```

Παράδειγμα 3.2

Εστω ο κανόνας από την Εικ. 3.1 <DECLARATION> ::= (<CONSTDECL>)* (<VARDECL>)*

Υποθέτοντας ότι ήδη έχουμε υλοποιησει-δηλώσει το <CONSTDECL> και το <VARDECL> σε μέθοδο constDecl() και varDecl() θα πάρουμε το εξής:

Σημείωση: Βρήκαμε * ;;; αυτό σημαίνει while και οχιτι if !!

```
void SyntaxAnalyzer::declaration() {  
    cuToken = this->lex->getToken();  
    while (cuToken == CONST || cuToken==VARIABLE){  
        if(cuToken ==CONST){  
            this->constDecl();  
            cuToken = this->lex->getToken();  
        }  
        else if(cuToken == VARIABLE){  
            this->varDecl();  
            cuToken = this->lex->getToken();  
        }  
    }  
}
```

3.3 Μεθόδοι και Πεδια της κλάσης SyntaxAnalyzer

Επειδή στα αρχεια της εργασίας θα βρούμε και άλλες μεθόδους, παραμέτρους και πεδία εδω θα αναφέρουμε καθαρα τις μεθόδους και τα πεδία που αφορούν την συντακτική ανάλυση.

public:

```
LexicalAnalyzer* lex;           (Λεκτικός Αναλυτής)  
SyntaxAnalyzer(LexicalAnalyzer *); (constructor με όρισμα Λεκτικό Αναλυτή)  
virtual ~SyntaxAnalyzer();  
void checkSyntax();             (Καλείτε για να ελένξει αν το πηγηατο αρχείο ανήκει  
στην γλώσσα μας)  
string getProgramName();        (Επιστρέφει το όνομα του προγραμματος)
```

Private:

```
int cuToken;                   (Αποθηκεύουμε την Λεκτική Μονάδα)  
string PROGRAM_NAME;          (Αποθηκευουμε το όνομα προγράμματος)  
void program();                (Συναρτήσεις γραμματικής Εικ. 3.1 με την  
μέθοδο LL1 σε C++)  
void programBllblock(string);  
void block();  
void decleration();  
void constDecl();
```

```
void assignList();
void assignConst();
void varDecl();
void varList();
void subPrograms(LabelQuad*);
void procOrFunc();
void procOrFuncBody();
void formalPars();
void formalParList();
void formalParItem();
void selectStat();
void sequence();
void blockOrStat();
void statement();
void assignmentStat();
void ifStat();
void elsePart();
void whileStat();
void forStat();
void stepPart(string*);
void callStat();
void printStat();
void inputStat();
void returnStat();
void actualPars();
void actualParList();
void actualParItem();
void condition(LQList**,LQList**);
void boolTerm(LQList**,LQList**);
void boolFactor(LQList**,LQList**);
void relationalOpen(string*);
void expression(string*);
void term(string*);
void factor(string*);
void optActualPars(string*);
void addOper();
void mulOper();
void optionalSign();
```

3.4 Ιδιαιτερότητες της C++

Κατά την υλοποίηση ξεφεύγουμε λίγο από την ιδέα της κλήσης του λεκτικού αναλυτη, για να επιστρέψει μια λεκτική μονάδα πριν από ένα μη-τερματικό σύμβολο. Αυτό εντοπίζεται όταν έχουμε while και ζητάμε να γεμίσουμε το cuToken πριν κάνουμε τον ελεγχο.. Και άν δεν ισχύει η συνθήκη η οποια δεν είναι συντακτικό προβλημα , θα ξανακαλέσουμε μετα το τέλος της while το cuToken με αποτέλεσμα να παραβλέψουμε μία λεκτική μονάδα οπου είναι πολύ σημαντική για να είναι σωστά συντακτίκα το προγραμμα μας.

Από το παράδειγμα 3.2

Εστω μετα την declaration() εκτελέστε το κομμάτι κώδικα ή μεθόδου που περιέχει

Αφου η declaration έχει while αυτό σημαίνει πως θα φτάσει σε ένα σημείο όπου η συνθήκη της θα είναι false άρα εμείς θα πρέπει να αξιοποιήσουμε την λεκτική μονάδα που για το declaration while είναι λάθος αλλα για το επόμενο κομμάτι ίσως είναι ζωτικής σημασίας.

Καλλόντας εδώ λοιπόν τον λεκτικό να μας επιστρέψει μια λεκτική μονάδα δεν του λέμε φερε μας την προηγούμενη αλλα μία καινούρια.

Έστω οτι το cuToken στο declaration είναι το SOMETHING_ID

Οπότε θα αγνοήσουμε την while.

Ζητώντας εδώ λοιπον ξανα για λεκτική μονάδα χάνουμε το SOMETHING_ID με XXXX_ID

Και άρα θα θα έχουμε μήνυμα λάθους.

Η λύση έιναι μέτα από δομές while να μην καλούμε τον λεκτικό αναλυτη να μας δώσει μια νέα λεκτική μονάδα.

cuToken = lex->getToken(); //Διγραφή ωστε να δουλέψει σωστά

If(cuToken == SOMETHING_ID)

cuToken = les->getTokenId();

Foo();

4. Ενδιαμεσος κώδικας

4.1 Στόχος

Ο ενδιάμεσος κώδικας εφαρμόζετε πάνω στον κώδικα του συντακτικού αναλυτή και στην πληροφορία που μας προσφέρει ο λεκτικός αναλυτής. Τι έιναι όμως ο ενδιάμεσος κώδικας;;

Είναι κώδικας που αποτελεί μια γλώσσα πιο προσιτή στην λογική λειτουργίας του μηχανήματος και λιγότερης απλότητας όπως μιας γλώσσας υψηλού επιπέδου.

Ας δουμέ τον ενατο μας σαν μία μηχανή είναι ευκολό να μας πει κάποιος διαβασε αυτο το κέιμενο 10 φορές αφου το γράψεις . Με την λογική σου πολύ απλά θα το γραψεις μόνο μία φόρα και όχι 10 και θα το διαβάζεις απο την αρχή ως το τέλος μέχρι να φτάσεις τις 10 φορές.

Έτσι <<σκέφτεται >> και μια μηχανή. Λογικά..

```
Αν γραψουμε εμείς δηλαδη  
writeFile();  
for (i=0 to 10 step 1)  
    readFile(); //Απο την αρχή ως το τέλος
```

Το μηχάνημα θα κάνει το εξής σε μονό 6 βήματα:

```
Βημα 1: Κλήση writeFile();  
Βημα 2: Αν το i<10 Αλμα Βημα 4  
Βημα 3: ΑΛΜΑ ΣΤΟ ΤΕΛΟΣ  
Βημα 4: readFile() ;  
Βημα 5: Ανξησε το i++  
Βημα 6: Αλμα στο Βήμα 2
```

Αν ακολουνούσαμε τον χαζό τροπο θα έιχαμε $1+5*10$ Βήματα... Πολυ περισσότερα απο τα 6 ε;;

Το ερώτημα που προκύπτει λοιπόν είναι πως θα καταφέρω να υλοποιήσω την γλώσσα μου ώστε να είναι πιο κατανοητη στο μηχανημά μου ;;;

4.2 Μεθόδοι και πεδία του Ενδιάμεσου κώδικα

Ο κώδικα αυτός βρίσκετε υλοποιημένος στην κλαση SyntaxAnalyzer και για αυτό τον λόγο θα αναφέρουμε μόνο μεθόδους και πεδία που αφορούν τον ενδιάμεσο κώδικα.

Για εμάς κάθε Βήμα είναι μία τετράδα η οποία μας αναφέρει τον αυξοντα αριθμό της για αν την βρισκουμε, την λειτουργία που πραγματοποιει, και τα αντιστοιχα ορισματα της.

Public:

void printList();

Private:

L QList mainList; (Η λίστα που αποθηκένουμε τις Τετράδες-τα Βήματα)

int temp_num; (Αυξοντας αριθμός των προσωρινών μεταβλητών που δημιουργούμε)

L QList* emptyList(); (Δημιουργεί μία άδεια λίστα τετράδων)

L QList* makeList(LabelQuad*); (Δημιουργεί μια λίστα τετράδων και προσθέτει το όρισμα της στην λίστα)

L QList* merge(L QList*,L QList*); (Ενώνει την 2η λίστα στην πρώτη)

LabelQuad* genQuad(string,string,string,string); (Παράγει μία νέα τετράδα και την προσθέτει στην Main λίστα τετράδων)

string newTemp(); (Δημιουργεί ένα όνομα προσωρινής μεταβλητή)

bool backpatch(L QList*,int); (Αλλάζει το πεδίο Z των τετράδων που περιέχοντε στην λίστα)

int nextQuad(); (Επιστρέφει τον αριθμό της επομενης τετράδας που προκειτε να παραχθεί)

4.2.1 Κλάση LabelQuad (Βήμα)

private:

```
int pos;                                (Κρατάει τον ανξόντα αριθμό βημάτος)  
string op,x,y,z;                        (Κρατάει το oparation και αντιστοιχα ορίσματα)
```

public:

```
LabelQuad(int,string,string,string,string);  
void print();                            (Εκτυπώνει την τετράδα-το Βήμα στην οθόνη)  
virtual ~LabelQuad();  
int getPos();                           (Μεθόδοι getters & setters)  
void setZ(int);  
const string& getOp() const;  
void setOp(const string& op);  
const string& getX() const;  
void setX(const string& x);  
const string& getY() const;  
void setY(const string& y);  
const string& getZ() const;
```

4.2.2 Κλάση LQList

public :

```
list<LabelQuad*> m_list; //Λιστα με Τετράδες (Κώδικας Βημάτων )  
LQList();
```

4.3 Μετατροπή δομών σε <<Κώδικα Βημάτων>>

Για κάθε δομή υπάρχει και ένας αντίστοιχος <<κώδικας βημάτων>> ή ενδιάμεσος κώδικας.

Αναλογα με την γλώσσα που φτιάχνουμε τον μεταφραστή μας υπάρχουν και περιορισμοί.

Παρακάτω για συντομία αναφερόμαστε απευθπίας στην γραμματική και όχι στον κώδικα C++.

Αρχή και τέλος block

```
<PROGRAM> ::= program ID <PROGRAMBLOCK (ID) > genquad("halt", "_", "_", "_")
<PROGRAMBLOCK (name)> ::=
    <DECLARATIONS>
    <SUBPROGRAMS>
    genquad("begin_block", name, "_", "_")
    <BLOCK>
    genquad("end_block", name, "_", "_")
```

Αριθμητικές Παραστάσεις

Pos: op,x,y,z Kάνει το Operation μεταξύ x & y και το αποτέλεσμα το αποθηκεύει στο z

Παράδειγμα:

$x + (y + z) \times w$

Ενδιάμεσος κώδικας βημάτων

1: +,y,z,T_1

2: ×,T_1,w,T_2

3: +, x, T_2, T_3

```
Expression(place) -> Term1(place)( + Term2(place){P1})* {P2}
{P1}: w = newTemp()           //Νέο όνομα προσωρινής μεταβλητής
genquad("+",Term_1.place, Term_2.place, w)
Term_1.place=w
{P2}: Expression.place =Term_1.place      //Επιστρέφουμε δηλαδή που αποθηκεύτηκε το αποτέλεσμα
```

Term(place) -> Factor1(place)(x Factor2(place) {P1})* {P2}

{P1}: w = newTemp()
genquad("x",Factor1.place,Factor2.place,w)
Factor1.place=w
{P2}: Term.place=Factor1.place

Factor-> (Expression(place)) {P1}
{P1}: Factor.place=Expression.place
Factor -> id {P1}
{P1}: Factor.place=id

Κομάτι κώδικα από το SyntaxAnalyzer C++

```
void SyntaxAnalyzer::expression(string* e) {
    this->optionalSign();
    if(cuToken == PLUS || cuToken == MINUS)
        cuToken = this->lex->getToken();
    string Term_one = this->lex->getLexis();
    this->term(&Term_one);
    while(cuToken == PLUS || cuToken == MINUS){
        string symbol = this->lex->getLexis();
        string Term_two;
        this->term(&Term_two);
        string w = this->newTemp();
        genQuad(symbol,Term_two,Term_one,w);
        Term_one = w;
    }
    *e = Term_one;
}
```

Οι λογικές παραστάσεις έχουν μια ιδιαιτερότητα καθώς δεν ξέρουμε ακομά όταν ισχνει η όχι η συνθήκη σε ποια τετράδα πρέπει να πάμε. Για αυτό τον λόγο φτιάχνουμε λίστες που αποθηκένουμε αντες τις τετράδες που θα χρειαστούν γέμισα τους όταν γνωρίζουμε 100% που θα πρέπει να κανουν άλμα.

➤ Λογικές Παραστάσεις - OR

Condition(true_list,false_list)->
BoolTerm1(true_list,false_list){P1}(or {P2}BoolTerm2(true_list,false_list) {P3})*

{P1}:Condition.true_list= BoolTerm1.true_list
Condition.false_list= BoolTerm1.false_list
{P2}:backpatch(Condition.false_list nextquad())
{P3}:Condition.true_list= merge(Condition.true_list, BoolTerm2.true_list)

Condition.false_list= BoolTerm2.false_list

➤ *Λογικές Παραστάσεις - AND*

BoolTerm(true_list,false_list)->

BoolFactor1(true_list,false_list){P1}(and {P2}BoolFactor2(true_list,false_list) {P3})*

{P1}:BoolTerm.true_list= BoolFactor1.true_list

BoolTerm.false_list= BoolFactor1.false_list

{P2}:backpatch(BoolTerm.true_list, nextquad())

{P3}:BoolTerm.false_list= merge(BoolTerm.false_list, BoolFactor2.false_list)

BoolTerm.true_list= BoolFactor2.true_list

➤ *Λογικές Παραστάσεις*

BoolFactor(true_list,false_list) ->(RealOp(true_list,false_list)) {P1}

{P1}: RealOp.true_list=BoolFactor.true_list

RealOp.false_list=BoolFactor.false_list

RealOp(true_list,false_list) ->Expression1(place)relopExpression2(place){P1}

{P1}: RealOp.true_list=makelist(nextquad())

genQuad(relop, Expression1.place, Expression2.place, “_”)

RealOp.false_list=makelist(nextquad())

genQuad(“jump”, “_”, “_”, “_”)

Κλήση Υποπρογραμμάτων

Κλήση διαδικασίας:

```
call assign_v(in a, inoutb)
genQuad(par, a, CV, _)
genQuad(par, b, REF, _)
genQuad(call, assign_v, _, _)
```

Κλήση συνάρτησης:

```
error = assign_v(in a, inoutb)
genQuad(par, a, CV, _)
genQuad(par, b, REF, _)
w = newTemp()
genQuad(par, w, RET, _)
genQuad(call, assign_v, _, _)
```

Εντολή return

S -> return (Expression(place)) {P1}
{P1}:genquad("retv",Expression.place,"_","_")

Εκχώρηση

S-> id := Expression(place) {P1}
{P1} : genQuad(":=",Expression.place,"_",id)

Δομή while

S -> while {P1}Condition(true_list,false_list) do {P2}S1{P3}
{P1}:Bquad:=nextquad()
{P2}:backpatch(Condition.true_list,nextquad())
{P3}:genquad("jump","_","_","Bquad")
backpatch(Condition.false_list,nextquad())

Δομή if

S -> if Condition(true_list,false_list) then {P1} S1{P2}TAIL {P3}

{P1}: backpatch(Condition.true,nextquad())
{P2}:ifList=makelist(nextquad())
genquad("jump","_","_","_")
backpatch(Condition.false,nextquad())
{P3}:backpatch(ifList,nextquad())

TAIL -> else S2| TAIL -> ε

Εισοδος Εξοδος

S -> input (id) {P1}
{P1}:genquad("inp",id,"_","_")

S -> print (Expresion(place)) {P2}
{P2}:genquad("out",Expression.place,"_","_")

5. Πίνακας Συμβόλων

5.1 Στόχος

Μέχρι στιγμής αυτό που καταφέραμε είναι να παράγουμε μία ενδιαμεση γλώσσα με βηματα ,αναγνώσιμη από τον υπολογιστή. Μα ο υπολογιστής για να θυμάται ονόματα μεταβλητών και συναρτήσεων αλλα και που βρίσκονται μεσα στο προγραμμα πρέπει να χρησημοποιησει μνήμη. Για αυτό θα χρησημοποιήσουμε έναν <>Πίνακα συμβόλων>>. Ειναι στην ουσία μία λίστα με σκοπιές που καθε μία σκοπιά(πεδίο πρόσβασης) έχει και μεταβλήτες ή συναρτήσεις που είναι σε θέση να εκτελεσει. Μια τέτοια δομή θα φτιάξουμε παρακάτω..

Παράδειγμα: Αν καλέσουμε μία συνάρτηση foo() ο υπολογιστής θα πει.. Τι ειναι αυτο;; για να μάθει τι είναι αυτό που καλείτε αλλα και που θα το βρει στην μνήμη του, το αναζήτα στον πίνακα συμβόλων.

Στον κώδικα στο SyntaxAnalyzer.cpp Ο πίνακας συμβόλων είναι μια **list<Scope*> scopeList;**

5.2 Μέθοδοι & Πεδία της κλάσης Entity

Tι είναι ένα αντικέιμενο Entity;;

Ένα Entity είναι μια δομή οπου κρατα πληροφορίες για την μεταβλητή, συναρτηση, παραματερους ή σταθερές. Όπως και για το που θα την βρόυμε μέσα στην μνήμη **offset**.

Μεθόδοι & Πεδία

public:

```
Entity(string,int,int); (Δημιουργεί το entity ώστε να περιγράφει μεταβλητή/παράμετρο)
//Function
Entity(string,int,int,list<Argument*>,int); (Περιγράφει μια Συνάρτηση/Διαδικασία)
Entity(string,string); (Περιγράφει μία σταθερά)
Entity(string,int); (Περιγράφει μια προσωρινή μεταβλητή)
```

...

Getterss/setters πεδίων

private:

```
string name; (Όνομα μεταβλητής/συναρτησης κλπ..)
//Variable,Function
int type; (Πέρνει τίμες VARIABLE/FUNCTION/PROCEDURE)
//Variable,Function,Parametres,tempVar
int offset; (Περιγράφει πόσες θέσεις στην μήμη θα βρούμε την μεταβλητή/συναρτηση μας κλπ.)
//Function
int startQuad; (Αν το entity είναι function μας κρατάει πιά τετράδα της συνάρτησης εκτελείτε πρώτη)
list<Argument*> argList; (Λιστα με παραμέτρους συνάρτησης)
int frameLength; (Αν προκειτε για συνάρτηση μας λέει πόσο χώρο πιάνει στην μνήμη)
//Const Var
string value;
//Parametres
int parMode;
```

5.3 Μεθόδοι & Πεδία της Κλάσης Scope

Tι είναι ένα Scope;;;

Είναι μία Λίστα από Entities που περιγράφουν μία συνάρτηση ή το πρόγραμμά μας.
Το Προγραμμά μας έχει ένα Scope, αλλα και κάθε αλλη συνάρτηση που συναντάμε έχουμε Scope με πληροφορίες για τις μεταβλητές/συναρτήσεις που χρησιμοποιει(Entities).Κάθε Scope μικρότερου level έχει προσβαση στα scope με μεγαλύτερο βάθος φωλιάσματος.

Μεθόδοι & Πεδία

private:

```
list<Entity*> entList;    (Λιστα με τα Entities)
int nesting_level;        (Βάθος Φωλιάσματος)
Scope *enclosingScope;   (Δεν θυμάμαι;;)
int frameLength;          (Αυτήν την πληροφία την θέλουμε και εδώ αν πρόκειτε για συνάρτηση)
```

public:

```
string name;            (Όνομα προγράμματος/συνάρτησης/διαδικασίας)
Scope(string);          (Constructor)
Scope* getEnclosingScope();
void setEnclosingScope(Scope* enclosingScope);
int getNestingLevel();
void setNestingLevel(int nestingLevel);
void add(Entity*);      (Προσθήκη ενός Entity στην λίστα)
Entity* getEntity(string); (Επιστροφή Entity βάση ονόματος)
void print();
```

5.4 Ενέργειες στον Πίνακα Συμβόλων

- **Προσθήκη νέου Scope:** όταν ξεκινάμε τη μετάφραση μιας νέας συνάρτησης
scopeList.push_back(SCOPE*);
- **Διαφραφή Scope:** όταν τελειώνουμε τη μετάφραση μιας συνάρτησης - με τη διαγραφή διαγράφουμε την εγγραφή (record) του Scope και όλες τις λίστες με τα Entity και τα Argument που εξαρτώνται από αυτήν
scopeList.erase(i);
- **Προσθήκη νέου Entity current_scope->add(new Entity(.....))**
 - όταν συναντάμε δήλωση μεταβλητής
 - όταν δημιουργείται νέα προσωρινή μεταβλητή
 - όταν συναντάμε δήλωση νέας συνάρτησης
 - όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης
- **Προσθήκη νέου Argument:** όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης
- **Αναζήτηση: current_scope->getEntity(name)**

μπορεί να αναζητηθεί κάποιο entity με βάση το όνομά του.

Η αναζήτηση ενός entity γίνεται ξεκινώντας από την αρχή του πίνακα και την πρώτη του γραμμή. Αν δε βρεθεί πηγαίνουμε στην επόμενη γραμμή έως ότου βρεθεί το entity ή τελειώσουν όλα τα entities οπότε επιστρέφουμε και μήνυμα λάθους. Αν με το ζητούμενο όνομα υπάρχει πάνω από ένα entity τότε επιστρέφουμε το πρώτο που θα συναντήσουμε.

5.5 Μέθοδοι & Πεδια του Πίνακα Συμβόλων στην κλάση SyntaxAnalyzer

Public:

list<Scope*> scopeList; (Πίνακας συμβόλων)
Scope *current; (Κραταει το Scope στο οποιο κάνουμε ενέργειες τωρα)

Private:

list<Argument*> tempArgList; (Προσωρινά Arguments ωστε να γεμίσουμε μετα το κατάληγο Entity)
int offset; (Κρατάει το offset προσωρινά του Current Scope)
int Main_Program_Framelength;
int nesting_level; (Κρατάει το βάθος φωλιάσματος που έχουμε φτάσει)

5.5 Τα περίεργα της C++

Όπως είδαμε παραπάνω κρατάμε προσωρινά κάποιες τιμές όπως π.χ τα arguments..

Arguments

Όταν καλούμε την συναρτησή μας πρέπει να αποθηκεύσουμε αυτες τις παραμέτρους κάπου οποτε χρησιμοποιούμε την λίστα αυτή.

Offset

Το offset αυτό γίνεται 0 όταν μεταφράζουμε μία συνάρτηση μα απαιτήται όταν τελειώσει η συνάρτηση να επαναφέρουμε το offset με το offset του γονέα Scope. Επίσης πειδή οι μεταβλητές πιάνουν χώρο 4 στην μνήμη/μεταβλητή το offset το προσαυξάνουμε κατα 4 και αυτό.

*Scope * current*

Το χρειαζόμαστε καθώς θα πρέπει να κάνουμε ενέργειες πάνω στο scope που εργαζόμαστε χωρίς να ψάχνουμε μέσα στην λίστα... Δεν είναι λίγο χρονοβόρο;;

6. Τελικός κώδικας

6.1 Στόχος

Όπως λέγαμε και προηγουμένως πριν περάσουμε στον τελικό κώδικα χρειαζόμασταν έναν ενδιαμεσο κώδικα παραπλήσιο με τον τελικό και κατανοητός από τον ανθρωπο αλλα και πληροφορία για το που θα αποθηκέυσουμε τις μεταβλητές και συναρτήσεις στην μνήμη. Στον τελικό κώδικα αυτό που έχουμε να κάνουμε είναι να μεταφράσουμε σε γλώσσα assembly τον ενδιαμεσο μας κώδικα χωρίς ομως να κάνουμε χρήση των μεταβλητών ως οναματα αλλα ως διευθυνσεις στην μνήμη.

Για τον συγκεκριμένο μεταφραστή θα χρησημοποιήσουμε τον metasim assembler (developed in Univercity of Ioannina).

Λόγω ελειψης documentation ίσως υπάρχουν σημαντικές ελειψεις κατα την λειτουργία.

6.2 Εισαγωγή στον Metasim

Με λίγα λόγια

Ως assemblers δεν μπορεί να χρησημοποιησει ονόματα των μεταβλητών μας αλλα τις διευθυνσεις τους στην μνήμη. Ο metasim περιέχει καταχωρητές R[0]...R[255] για την αποθηκευση προσωρινων αποτελεσματων απο πράξεις διευθυνσεων μεταβλητων αλλα και συναρτήσεων.

Και για την σύγκριση τιμών περιέχει καταχωρητές ονοματι SR που με τις ήδη υπαρχουσες εντολές αυτοι αλλάζουν αυτοματα και μπορούμε να αποφανθούμε για το αποτέλεσμα ένκολα.

Στην κυρίως Μνήμη του προγράμματος έχουμε προσβαση γραφοντας στο assembly αρχειο μας M[R[..]],M[9999],M[R[0]+5]

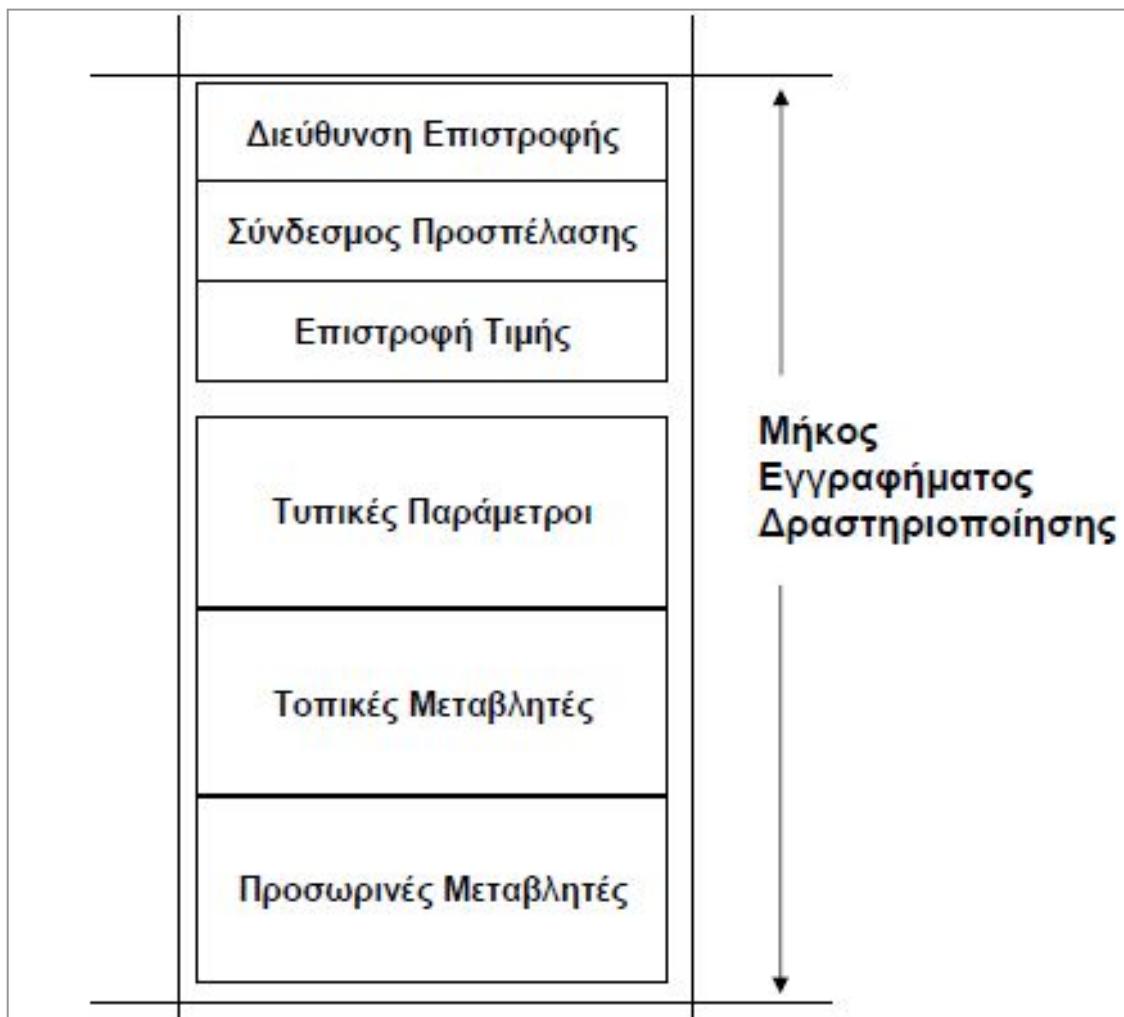
Ας δούμε την μνήμη ως μία στοίβα.. Ο καταχωρητής R[0] δείχνει στην κορυφή της.. Κατα σύμβαση έχουμε θέσει να δέιχνει $R[0] = M[R[0] + 600]$ και αλλάζει αναλογα με το αν πρέπει να ανατρέξουμε στο **εγγράφημα δραστηριοποιησης** μιας συναρτησης. Ο R[1] R[2] για την αποθηκευση τιμών μεταβλητών για πράξεις όπως προσθεση,αφαιρεση κλπ..

Και ο R[254],R[255] για ότι άλλο.

Κάθε εντολή μας στον ενδιάμεσο ειναι μια αντιστοιχή στον assemblers όπως αναφέρουμε παραάτω οπότε κάνουμε το πρόγραμμα πιο εύκολο. Το μόνο θέμα μας είναι το πως θα αναζητούμε τις μεταβλητές σωστά ειτε περνανε με τιμή είτε με αναφορά.

6.3 Εγγράφημα δραστηριοποιησης

- ◆ Δημιουργείται για κάθε συνάρτηση από αυτήν που την καλεί
- ◆ Όταν αρχίζει η εκτέλεση της συνάρτησης ο δείκτης στοίβας $R[0]$ μεταφέρεται στην αρχή του εγγρασήματος δραστηριοποίησης
- ◆ Περιέχει πληροφορίες που χρησιμένουν για την εκτέλεση και τον τερματισμό της συνάρτησης καθώς και πληροφορίες που σχετίζονται με τις μεταβλητές που χρησιμοποιεί.
- ◆ Όταν τερματίζεται η συνάρτηση ο χώρος που καταλαμβάνει το εγγράφημα δραστηριοποίησης επιστρέφεται στο σύστημα.



Εικ. 6.1 Αναπαράσταση Εγγραφηματος Δραστηριοποιησης στην Μνήμη

- ◆ **Διεύθυνση επιστροφής:** η διεύθυνση στην οποία θα μεταβεί η ροή του προγράμματος όταν ολοκληρωθεί η εκτέλεση της συνάρτησης.
- ◆ **Σύνδεσμος Προσπέλασης:** δείχνει στο εγγράφημα δραστηριοποίησης που πρέπει να αναζητηθούν μεταβλητές οι οποίες δεν είναι τοπικές αλλά η συνάρτηση έχει δικαίωμα να χρησιμοποιήσει.
- ◆ **Επιστροφή τιμής:** η διεύθυνση στην οποία θα γραφεί το αποτέλεσμα της συνάρτησης όταν αυτό υπολογιστεί.
- ◆ Χώρος αποθήκευσης παραμέτρων συνάρτησης
 - αποθηκεύεται η τιμή, αν πρόκειται για πέρασμα με τιμή
 - αποθηκεύεται η διεύθυνση, αν πρόκειται για πέρασμα με αναφορά
- Χώρος αποθήκευσης τοπικών μεταβλητών
- Χώρος αποθήκευσης προσωρινών μεταβλητών

6.4 Εντολές Metasim

Επειδή κάποιοι μεθόδοι ευρέσης μεταβλητης επαναλαμβάνονται για αυτό τον λόγο φτιάχνουμε συναρτησεις που θα τις προσθέσουμε σε μία κλάση.

Εντολή => Meaning

ini tr => *tr register*

outi so1 =>*so1 register*

addit,so1,so2 =>*tr=so1 + so2*

subi tr,so1,so2 =>*tr=so1 - so2*

muli tr,so1,so2 =>*tr=so1 * so2*

divi tr,so1,so2 =>*tr=so1 / so2*

movi tr,so1 =>*tr := so1*

jmp addr => *addr label, M[..], R[..], δ/ση(αριθμός)*

cmpi so1,so2 =>*so1,so2 registers*

jb addr =>*addr label, M[..], R[..], δ/ση(αριθμός)*

jbe addr =>*addr label, M[..], R[..], δ/ση(αριθμός)*

ja addr =>*addr label, M[..], R[..], δ/ση(αριθμός)*

jae addr =>*addr label, M[..], R[..], δ/ση(αριθμός)*

je addr =>*addr label, M[..], R[..], δ/ση(αριθμός)*

jne addr =>*addr label, M[..], R[..], δ/ση(αριθμός)*

6.5 Από τετράδες σε τελικό κώδικα

```

jump,_,_w
    jmp Lw

relop,x,y,w           =>relop={=,<>,>,>=,<,<=}
    loadvr(x,1)R[1]=x
    loadvr(y,2)R[2]=y
    cmpi R[1], R[2]
    condjmp Lw           =>comdjmp={je,jne,jb,jbe,ja,jae}

:=,x,_z
    loadvr(x,1)R[1]=x
    storerv(1,z)z=R[1]

op,x,y,z             =>op={+,-,*,/}
    loadvr(x,1)           =>R[1]=x
    loadvr(y,2)           =>R[2]=y
    op R[3], R[1], R[2]   =>op={addi,subi,muli,divi}
    storerv(3,z)          =>z=R[3]

out,x,_
    loadvr(x,1)           =>R[1]=x
    outi R[1]

in,x,_
    ini R[1]               =>x=R[1]
    storerv(1,x)

:=,x,_,return_value
    loadvr(x,1)           =>R[1]=x
    movi R[255],M[8+R[0]]
    movi M[R[255]],R[1]

Αποθηκένεται ο χ στην διευθυνση που βρισκετε αποθηκευμένη στην 3η θέση του εγγραφήματος δραστηριοποιησης.

```

•
•
•
•

Τα υπόλοιπα στις σημειώσεις Μεταφραστών στο ecourse.(No time)

6.5.1 Η συνάρτηση *gnlvcode()*

Μεταφέρει στον R[255] την διεύθυνση μια μή τοπική μεταβλητής

Τα υπόλοιπα στις σημειώσεις Μεταφραστών στο ecourse.(No time)

6.5.2 Η συνάρτηση $load(v,r)$

Φωρτώνει την διεύνυνση της μεταβλητής v ή την τιμή της στον καταχωρητή $R[r]$

Τα υπόλοιπα στις σημειώσεις Μεταφραστών στο ecourse.(No time)

6.5.2 Η Συνάρτηση $storev(r,v)$

Αποθηκένει την τιμή,διευθυνση της τιμής του καταχωρητη $R[r]$ στην μεταβλητή v (θέση στην Μνήμη π.χ $M[600+v.offset]$)

Τα υπόλοιπα στις σημειώσεις Μεταφραστών στο ecourse.(No time)

6.6 Η κλάση *EndCodeEngine*

Η Κλάση *EndCodeEngine* κάνει ακριβώς την διαδικάσία που περιγράφαμε στην παράγραφο 6.5.Δημιουργεί ένα αρχείο στο οποίο και αποθηκένει τον τελικό κώδικα αν φτάσουμε σε επιτυχία.Και δημιουργείτε ως αντικείμενο στην κλάση *SyntaxAnalyzer* και χρησιμοποιείται αμέσως μετά το τέλος μια συνάρτησης ή τον προγράμματος μεταφράζοντας μια μια τις τετράδες. Επειδή όπως αναφέραμε πρέπει αν στεφει με επιτυχία η διαδικασία χρησημοποιούμε μία δομή *stringstream* της *c++* για την προσωρινή αποθήκευση του αρχείου μας στην μνημη *ram*.

6.6.1 Μεθόδοι & Πεδία της κλάσης

private:

SyntaxAnalyzer* syntax;	(Δεικτής σε αντικέιμενο συντακτικού αναλυτή)
stringstream assemblyStream;	(Προσωρινό αρχείο στην μνημη)
fstream assemblyFile;	(Τελικό αρχείο στον δίσκο)
int parametr_number;	(Αριθμος Παραμετρων στην περιπτωση κλήσης συνάρτησης)

public:

EndCodeEngine(SyntaxAnalyzer*);	
virtual ~EndCodeEngine();	
void gnlvcode(string);	
void loadvr(string,int);	
void storevr(int,string);	
void generateEndCode(LabelQuad*);	(Παράγει τελικό κώδικα με βάση την τετράδα που έχει ως παράμετρο)
void generateParEndCode(LabelQuad*);	(Παραγωγή κώδικα αν έχουμε τετράδα παραμέτρου)
void tryGenerateRealOperator(LabelQuad*);	(Παραγωγή κώδικα αν έχουμε τετράδα RealOp >,<,=, etc)
void tryGenerateOperator(LabelQuad*);	(Παραγωγή κώδικα αν έχουμε τετράδα Operator +,- etc)
void printFile(string);	(Τυπώνει στο τελικό αρχειο)
void printConsole();	(Τυπώνει τον κώδικα στην κονσολα)

7. Η Συνάρτηση Main()

Παίρνει σαν όρισμα από την κονσόλα την διελυθυνσή και το ανοιγει αν υπάρχει.

Δημιουργεί τον Λεκτικό αναλυτή.

Δημιουργει τον Συντακτικό αναλυτή με όρισμα μια αναφόρα στον Λεκτικό αναλυτή.

Καλέι την μέθοδο checkSyntax για τον έλεγχο του πηγηαιου αρχειου αν ανήκει στην γλώσσα και την παραγωγή τελικού κώδικα.

Τέλος.